Deep Network Development

Practice Memorandum

Course Instructors
Tamás Takács, Imre Molnár

Course Demonstrators
Arpan Ekka, Máté Badó Miklós

Course Supervisor
Kristian Fenech

Department of Artificial Intelligence
Faculty of Informatics, ELTE
2026

Preface

This volume is the course memorandum for *Deep Network Development* (course code `IPM-22fmiDNDEG`), a Master's-level course offered by the Department of Artificial Intelligence, Faculty of Informatics, Eötvös Loránd University (ELTE), under the academic supervision of Dr. Kristian Fenech.

Scope of the course

Deep Network Development introduces students to the full landscape of modern deep learning through a weekly lecture paired with a hands-on practical session. This memorandum focuses on the practical side of the course. The semester covers the foundations of neural networks (linear and logistic regression, multi-layer perceptrons, backpropagation, optimizers), convolutional architectures for image classification and transfer learning, the four classical computer-vision tasks (object detection, semantic and instance segmentation, pose estimation) together with their modern deep-learning solutions, recurrent networks and attention mechanisms for sequence modelling, the Transformer architecture with its vision and language variants, and finally generative models and inverse problems such as depth estimation, optical flow, and text-guided image inpainting.

Content of this memorandum

Thirty notebook-based practical sessions, one per topic, covering everything from Python and NumPy foundations through to text-guided image inpainting. Each chapter is a cleaned and typeset reproduction of the Colab notebook delivered in class, with code preserved, outputs abridged where volume would impair readability, and a revised homework set appended so that the reader can continue practising independently.

The graded midterm and final examinations, together with the quiz example collections, are reproduced in the companion volume *Deep Network Development Lecture Memorandum* that accompanies the lecture slide decks.

The accompanying assignments and coding entry examinations are reproduced in the companion volume *Deep Network Development Assignments and Coding Exams Memorandum*.

Authorship and attribution

The course content collected here was co-developed by Tamás Takács and Imre Molnár on an approximately equal basis. Two practice chapters, NMT with Transformers and Text-Guided Image Inpainting, were authored by Máté Badó Miklós and are attributed to him in their respective chapter headers.

The editorial work required to convert the Colab notebooks and exam documents into a single cohesive typographic volume was produced specifically for this grant submission.

Accompanying lecture slides

The weekly lecture slide decks are submitted alongside this memorandum as separate PowerPoint (`.pptx`) files. Their typeset reproduction forms the companion volume *Deep Network Development Lecture Memorandum*. The deck titles are:

1. Introduction
2. Linear Regression and Artificial Neural Networks
3. Image Classification and Convolutional Neural Networks
4. Convolutional Neural Network Architectures and Transfer Learning
5. Object Detection and Image Segmentation
6. Natural Language Processing Basics and Recurrent Neural Networks
7. Transformers and Vision Transformers
8. Deep Learning Tools for Computer Vision, Part 1
9. Deep Learning Tools for Computer Vision, Part 2

Contents

Preface **2**

I. Practices **19**

1. Python and Google Colaboratory **20**

1.1. **Using Google Colaboratory** . . . 20
1.2. **Colab runtimes and mounting Google Drive** . . . 21
1.3. **Running Python scripts** . . . 21
1.4. **Basics Comments and Print** . . . 22
1.4.1. **Numbers and variables** . . . 22
1.4.2. **Strings** . . . 23
1.4.3. **Logical expressions** . . . 23
1.4.4. **`NoneType` type** . . . 24
1.5. **Lists** . . . 24
1.6. **If/else statements, loops, the `pass` keyword** . . . 27
1.7. **Homework (Optional)** . . . 28

2. Python Fundamentals **29**

2.1. **Functions** . . . 29
2.1.1. **Simple functions, `return`** . . . 29
2.1.2. **Optional parameters** . . . 30
2.1.3. **Recursion** . . . 30
2.1.4. **Nested function** . . . 31
2.1.5. **Passing function as an argument** . . . 31
2.1.6. **Anonymous functions (`lambda`)** . . . 32
2.2. **Data structures** . . . 32
2.2.1. **`tuple` type** . . . 32
2.2.2. **Packing and unpacking tuples** . . . 33
2.2.3. **`set` type** . . . 34
2.2.4. **Set operations** . . . 35
2.2.5. **`dict` type: dictionary (not only for words...)** . . . 36
2.3. **Important built-in functions** . . . 37
2.3.1. **The `enumerate()` function** . . . 37
2.3.2. **The `zip()` function** . . . 37
2.3.3. **The `min()`, `max()`, `abs()`, `round()`, `all()`, `any()` functions** . . . 38
2.3.4. **The `map()` and `filter()` functions** . . . 38
2.4. **List, Set, Dict comprehension** . . . 39

2.5. **Classes** . . . 42
2.6. **Type hints and dataclasses** . . . 43
2.7. **Handling errors, the `assert` statement** . . . 43
2.8. **Iterators and generators** . . . 44
2.9. **Homework (Optional)** . . . 44

3. NumPy Introduction **45**
3.1. **Creating arrays** . . . 45
3.2. **Numpy basics - the `ndarray` type** . . . 47
3.3. **Changing the shape of arrays** . . . 48
3.4. **Conversion between datatypes** . . . 50
3.5. **Basic indexing, multiple views** . . . 50
3.6. **Simple arithmetic operations, treating arrays of different shapes (broadcasting)** . . . 53
3.7. **Memory layout, views vs. copies** . . . 60
3.8. **Homework (Optional)** . . . 61

4. NumPy Fundamentals **62**
4.1. **Operations performed along certain axes** . . . 62
4.2. **Matrix and vector operations, linear algebra** . . . 63
4.3. **Broadcasting in practice** . . . 66
4.4. **Concatenating arrays, inserting, deleting.** . . . 66
4.5. **Advanced indexing** . . . 68
4.6. **Other useful Numpy operations** . . . 73
4.7. **Homework (Optional)** . . . 73

5. PyTorch GPU Integration **74**
5.1. **Necessary Imports** . . . 74
5.2. **GPU** . . . 74
5.3. **PyTorch** . . . 75
5.3.1. **Numpy vs Torch** . . . 75
5.4. **Homework (Optional)** . . . 79

6. PyTorch Autograd **80**
6.1. **Necessary Imports** . . . 80
6.2. **Computation graphs and Autograd** . . . 80
6.3. **Lecture example** . . . 81
6.3.1. 1. Create the directed acyclic graph of the operations . . . 82
6.3.2. 2. Substitution . . . 82
6.4. **Substitution with other values** . . . 87
6.5. **Update parameters** . . . 88
6.6. **How different optimisers traverse the loss landscape** . . . 88
6.7. **Detaching, `no_grad`, and gradient accumulation** . . . 89
6.8. **Homework (Optional)** . . . 90

7. Linear Regression **91**
7.1. **Necessary Imports** . . . 91
7.2. **Reading the Data** . . . 91
7.3. **Exploring the Data** . . . 91
7.4. **Data Preprocessing** . . . 93
7.5. **Correlation Heatmap** . . . 93
7.6. **Handling data** . . . 94
7.7. **Handling missing values** . . . 95
7.8. **The bias-variance tradeoff** . . . 96
7.9. **Linear Regression** . . . 96
7.10. **Linear Regression with Tim** . . . 97
7.10.1. **Visualizing Life expectancy vs BMI plot** . . . 97
7.11. **Preprocessing data** . . . 98
7.12. **Linear Regression with Tom** . . . 99
7.13. Visualzing Life expectancy vs GDP plot . . . 99
7.14. **Preprocessing data** . . . 100
7.15. **Multiple Regression** . . . 101
7.16. **Multiple Regression with Tim** . . . 102
7.17. **Multiple Regression with Tom** . . . 103
7.18. **A note on activation functions** . . . 104
7.19. **Solving the problem using a Deep Neural Network** . . . 104
7.20. **Preprocessing** . . . 104
7.20.1. **Selecting features** . . . 104
7.20.2. **Normalizing values** . . . 106
7.20.3. **Shuffling and splitting the dataset** . . . 107
7.21. **Defining DataLoader and Model** . . . 107
7.21.1. **Make a prediction for Tim and Tom:** . . . 109
7.22. Tim: . . . 109
7.22.1. Tom: . . . 110
7.23. **Standardizing the inputs and running a prediction** . . . 110
7.24. **Interpreting the fitted coefficients** . . . 110
7.25. **Implement Early Stopping** . . . 111
7.26. **Homework (Optional)** . . . 112

8. Image Classification with CNNs **114**
8.1. **Necessary Imports** . . . 114
8.2. **Data - CIFAR10** . . . 114
8.2.1. **Geometric Transformations** . . . 115
8.2.2. **Color Transformations** . . . 115
8.2.3. **Miscellaneous Transformations** . . . 115
8.2.4. **Key Features of `DataLoader`** . . . 116
8.3. **Create and Train a Multi-Layered Perceptron (MLP)** . . . 120
8.3.1. **Let's create the neural network, set up the device and get a simple vizualisation** 121

8.4. **Loss function** . . . 121
8.4.1. **Optimizers and their options** . . . 121
8.5. **Train** . . . 121
8.5.1. **Data augmentation for classification** . . . 126
8.5.2. **Why CNN depth gives larger receptive fields** . . . 126
8.6. **Create and Train a Convolutional Neural Network (CNN)** . . . 127
8.7. **Model** . . . 129
8.7.1. **Let's create the neural network, set up the device and get a simple vizualisation** 130
8.8. **Visualisation with Tensorboard** . . . 132
8.9. **Loss function** . . . 132
8.9.1. **Optimizers and their options** . . . 133
8.10. **Train** . . . 133
8.11. **Confusion Matrix** . . . 137
8.12. **Homework (Optional)** . . . 138

9. Spectrogram Classification with CNNs **139**
9.0.1. **1. Raw Waveform** . . . 139
9.0.2. **2. Mel-Frequency Cepstral Coefficients (MFCC)** . . . 139
9.0.3. **3. Mel Spectrogram** . . . 140
9.0.4. **4. Tempogram** . . . 140
9.0.5. **5. Chromogram** . . . 140
9.0.6. **Comparison of Representations** . . . 141
9.1. **Necessary Imports** . . . 142
9.1.1. **Downloading the GTZAN Dataset** . . . 142
9.1.2. **Loading the Data** . . . 143
9.2. **Looking at Different Representations** . . . 143
9.3. **Sticking with Short-Time Fourier Transform** . . . 147
9.4. **Looking at our Mel Spectrograms in the Dataset** . . . 149
9.4.1. **Creating Datasets and Dataloaders** . . . 149
9.4.2. **Key Features of `DataLoader`** . . . 150
9.5. **Checking Class Distribution** . . . 151
9.6. **Train / Validation / Test splits and data leakage** . . . 153
9.7. **Creating our Model** . . . 154
9.7.1. **Architecture Overview:** . . . 154
9.8. **Training our Model** . . . 155
9.8.1. **Why Use This Approach?** . . . 155
9.8.2. **Tracking Performance & Early Stopping** . . . 155
9.9. **Loss function** . . . 156
9.9.1. **Optimizers and their options** . . . 157
9.9.2. **Plotting our Losses and Accuracies** . . . 159
9.10. **Testing our Model** . . . 160
9.10.1. **Data Augmentation for Spectrograms** . . . 161
9.11. **Training a New Model** . . . 163

9.12. **Visualizing Feature Vectors** . . . 165
9.13. **Music Genre Similarities** . . . 167
9.13.1. **Euclidean Distance** . . . 168
9.13.2. **Dot Product Similarity** . . . 168
9.13.3. **Cosine Similarity** . . . 168
9.14. **Integrated Gradients** . . . 170
9.14.1. **Task – Integrated Gradients Visualization** . . . 170
9.15. **Integrated Gradients Visualization** . . . 170
9.16. **Uploading your Own Music** . . . 172
9.17. **Homework (Optional)** . . . 175

10. Transfer Learning **176**
10.1. **Necessary Imports** . . . 176
10.2. **Loading the Data** . . . 177
10.3. **Data Augmentation** . . . 178
10.3.1. **Geometric Transformations** . . . 178
10.3.2. **Color Transformations** . . . 179
10.3.3. **Miscellaneous Transformations** . . . 179
10.3.4. **Visualizing a few images** . . . 180
10.4. **Training and Testing the Model** . . . 181
10.4.1. **Visualizing the model predictions** . . . 183
10.5. **Model overview: ResNet18** . . . 184
10.6. **Reading training curves** . . . 186
10.7. **Training a CNN from scratch** . . . 186
10.8. **Saving our Model** . . . 187
10.9. **SHAP Analysis** . . . 189
10.10. **When transfer learning works (and when it does not)** . . . 191
10.11. **Learning-rate schedules** . . . 191
10.12. **Finetuning the ResNet18 Model** . . . 192
10.13. **Keeping the ResNet18 Backbone Intact** . . . 195
10.14. **Another Way to Finetune the Entire Network** . . . 198
10.15. **Homework (Optional)** . . . 201

11. Quantization Techniques **202**
11.1. **Choosing between dynamic, static, and quantization-aware training** . . . 202
11.2. **Quantization in PyTorch** . . . 202
11.3. **Necessary Imports** . . . 203
11.4. **Loading the Data** . . . 204
11.5. **Data Augmentation** . . . 205
11.5.1. **Geometric Transformations** . . . 205
11.5.2. **Color Transformations** . . . 206
11.5.3. **Miscellaneous Transformations** . . . 206
11.5.4. **Visualizing a few images** . . . 207

11.6. **Model overview: ResNet18** . . . 208
11.7. **Finetuning the ResNet18 Model** . . . 210
11.8. **Quantization Aware Training (QAT)** . . . 215
11.8.1. **Steps to Perform QAT**: . . . 215
11.8.2. **Example of Preparing a Model for QAT:** . . . 215
11.8.3. **Conclusion on Quantization Methods** . . . 220
11.9. **Homework (Optional)** . . . 221

12. Object Detection with YOLO11 **222**
12.1. **Models** . . . 222
12.1.1. Key Points: . . . 222
12.2. **Installing the Ultralytics Package** . . . 223
12.3. **YOLO versions, heads, and model sizes** . . . 223
12.4. **Train** . . . 224
12.5. Building a New Model **(CLI)** . . . 226
12.6. Loading a Pretrained Model **(CLI)** . . . 227
12.7. **Validation** . . . 228
12.8. Validate an Official Model **(CLI)** . . . 228
12.9. **Inference** . . . 229
12.10. Infer an Official Model **(CLI)** . . . 229
12.11. Displaying the output . . . 230
12.12. Infer an Official Model **(Python)** . . . 231
12.13. **Train Results** . . . 235
12.14. **Homework (Optional)** . . . 236

13. Faster R-CNN for Object Detection **237**
13.1. **Necessary Imports** . . . 237
13.2. **The Penn-Fudan Pedestrian Detection Dataset** . . . 238
13.3. **Showcasing our Dataset** . . . 238
13.4. **Creating a Dataset and Transformations** . . . 241
13.5. **Building the Dataset** . . . 244
13.6. **Visualizing our Transformed Samples** . . . 245
13.7. **Two-stage vs. single-stage detectors** . . . 247
13.8. **Building the Fast R-CNN Model** . . . 247
13.9. **Training** . . . 251
13.10. **Testing our Model** . . . 257
13.11. **Homework (Optional)** . . . 262

14. Object Detection from Scratch **264**
14.1. **Necessary Imports** . . . 264
14.2. **A synthetic multi-object dataset** . . . 265
14.2.1. **Intersection over Union, visually** . . . 267
14.3. **Bounding box representations and Intersection over Union** . . . 268
14.4. **Anchor boxes** . . . 268

14.5. **Model architecture** . . . 269
14.6. **Target assignment** . . . 270
14.7. **Multi-task loss** . . . 271
14.8. **Training** . . . 272
14.8.1. **NMS at a glance** . . . 274
14.9. **Non-Maximum Suppression** . . . 275
14.10. **Inference pipeline** . . . 275
14.11. **Precision, recall, and mean Average Precision** . . . 278
14.12. **Homework (Optional)** . . . 279

15. Image Segmentation with U-Net **280**
15.1. **Necessary Imports** . . . 280
15.1.1. **Loading the data** . . . 280
15.2. **Looking at Our Dataset** . . . 281
15.3. **Loss functions for semantic segmentation** . . . 282
15.4. **Preparing our Data for Training** . . . 282
15.5. **Visualizing our Dataset Again using DataLoaders** . . . 284
15.5.1. **Createing a Neural Network with PyTorch** . . . 286
15.5.2. **Setting Up the Model** . . . 289
15.5.3. **Model and its Parameters** . . . 290
15.5.4. **Optimizers and their options** . . . 291
15.5.5. Loss Functions and Their Options . . . 291
15.6. **Training** . . . 291
15.6.1. **Saving and Loading the Model** . . . 293
15.6.2. **Testing our Model** . . . 293
15.7. **Visualizing our Predictions** . . . 294
15.8. **Homework (Optional)** . . . 296

16. Instance Segmentation with YOLO11 **298**
16.1. **Models** . . . 298
16.1.1. Key Points: . . . 299
16.2. **Installing the Ultralytics Package** . . . 299
16.3. **Masks versus boxes: what instance segmentation adds** . . . 299
16.4. **Train** . . . 300
16.5. **Building a New Model (CLI)** . . . 302
16.6. **Loading a Pretrained Model (CLI)** . . . 303
16.7. **Validation** . . . 304
16.8. **Validate an Official Model (CLI)** . . . 304
16.9. **Inference** . . . 305
16.10. **Infer an Official Model (CLI)** . . . 305
16.11. **Displaying the output** . . . 306
16.12. **Infer an Official Model (Python)** . . . 306
16.13. **Mask Plotting and Coverage Calculation** . . . 310

16.14. **IMPORTANT** . . . 311
16.15. **Homework (Optional)** . . . 313

17. Pose Estimation with YOLO11 **315**
17.1. **Models** . . . 315
17.1.1. Key Points: . . . 316
17.2. **Installing the Ultralytics Package** . . . 316
17.3. **Keypoints and visibility flags** . . . 317
17.4. **Train** . . . 317
17.5. **Building a New Model (CLI)** . . . 318
17.6. **Loading a Pretrained Model (CLI)** . . . 319
17.7. **Validation** . . . 320
17.8. **Validate an Official Model (CLI)** . . . 320
17.9. **Inference** . . . 321
17.10. **Infer an Official Model (CLI)** . . . 321
17.11. **Displaying the output** . . . 321
17.12. **Infer an Official Model (Python)** . . . 322
17.13. **Angle Calculation** . . . 327
17.14. **Homework (Optional)** . . . 329

18. Segment Anything Model 2 **331**
18.1. **SA-V Dataset** . . . 332
18.2. **SAM 2 Architecture** . . . 332
18.3. **Installing the Ultralytics Package** . . . 332
18.4. **Downloading images and videos for testing** . . . 332
18.5. **Prompt types and how to combine them** . . . 333
18.6. **SAM 2 Prediction Example (Ultralytics)** . . . 333
18.7. **Plotting our test images** . . . 334
18.8. **Run Inference on SAM 2** . . . 335
18.9. **SAM 2 Prediction Example (Meta)** . . . 337
18.10. **Environment Set-up** . . . 337
18.11. **Set-up Device and Configs** . . . 338
18.11.1. **Loading the SAM 2 video predictor** . . . 338
18.12. **Setup Helper functions for plotting** . . . 339
18.13. **Plotting test images** . . . 339
18.13.1. **Printing out the shape of image embeddings** . . . 342
18.14. **Specifying a specific object with a box** . . . 346
18.15. **Combining points and boxes** . . . 347
18.16. **Video segmentation with SAM 2 (Meta)** . . . 349
18.16.1. Example 1: Segment & track one object . . . 350
18.17. **Homework (Optional)** . . . 353

19. Sentiment Analysis **354**
19.1. **Necessary Imports** . . . 354

19.2. **Data** . . . 355
19.3. **Minimal Data Exploration** . . . 355
19.3.1. **Spilitting the Data into Train and Validation Sets** . . . 357
19.3.2. **Check Data Imbalance after Splitting** . . . 357
19.3.3. **Tokenization and Text Preprocessing** . . . 358
19.3.4. **Visualizing our Inputs as Text and Tokens** . . . 360
19.4. **Visualizing Tokenized Data** . . . 360
19.4.1. **Adding Padding** . . . 361
19.4.2. **Batching and Creating a DataLoader** . . . 362
19.5. **Padding, truncation, and bidirectional LSTMs** . . . 363
19.6. **LSTM-Based Model** . . . 364
19.7. **Training our Neural Network** . . . 365
19.7.1. **Saving and Loading the Model** . . . 369
19.8. **Plotting our Results** . . . 369
19.9. **Running Inference on our Model** . . . 370
19.9.1. Try an example from our data . . . 370
19.10. **Testing our Model with Arbitrary Reviews** . . . 371
19.11. **Homework (Optional)** . . . 372

20. Time Series Prediction **373**
20.1. **Necessary Imports** . . . 373
20.2. **Data** . . . 373
20.2.1. **Minimal Data Exploration** . . . 374
20.2.2. **Spilitting the Data into Train and Validation Sets** . . . 375
20.3. **Train, validation and test splits for time series** . . . 376
20.4. **Creating a Dataset and a Dataloader** . . . 376
20.5. **Models** . . . 377
20.6. **Train** . . . 380
20.7. **Evaluating our Model** . . . 383
20.7.1. **LSTM** . . . 384
20.7.2. **Vanilla RNN** . . . 386
20.7.3. **GRU** . . . 387
20.7.4. **ANN** . . . 388
20.7.5. **CNN** . . . 390
20.8. **Homework (Optional)** . . . 391

21. Neural Machine Translation **392**
21.1. **Necessary Imports** . . . 392
21.2. **Data** . . . 393
21.3. **Data Exploration** . . . 394
21.4. **Creating a Joint Dataset of Translations** . . . 395
21.5. **Save the New Subset** . . . 397
21.6. **Let's Load it Back in** . . . 397

21.7. **Creating our Vocabulary** . . . 397
21.8. **Tokenization** . . . 399
21.8.1. **Eliminating Duplicate Entries** . . . 401
21.9. **Splitting the Date** . . . 403
21.10. **Padding Sequences and Converting them to Tensors** . . . 405
21.11. **Creating a Dataset and a Dataloader** . . . 407
21.12. **Model** . . . 409
21.13. **Encoder** . . . 410
21.14. **Decoder** . . . 411
21.15. **Seq2Seq** . . . 413
21.16. **Teacher forcing, free running, and exposure bias** . . . 415
21.17. **Training** . . . 415
21.18. **Main Training Loop** . . . 417
21.19. **Plotting our Results** . . . 424
21.20. **Testing and Evaluating our Model** . . . 426
21.21. **Homework (Optional)** . . . 428

22. The Attention Mechanism **429**
22.1. **Necessary Imports** . . . 429
22.2. **Downloading Model Weights** . . . 430
22.2.1. **What attention looks like in practice** . . . 430
22.3. **The Attention Mechanism** . . . 431
22.4. **Why scaled dot-product attention divides by $\sqrt{d_k}$** . . . 432
22.4.1. **Scaled Dot-Product Attention** . . . 433
22.4.2. **How Does It Work?** . . . 433
22.4.3. **Why Do We Scale by $\frac{1}{\sqrt{d_k}}$?** . . . 434
22.4.4. **Masking in Attention** . . . 434
22.4.5. **Causal (Look-Ahead) Masking** . . . 435
22.4.6. **Multi-Head Attention** . . . 439
22.4.7. **How is Multi-Head Attention Applied in Neural Networks?** . . . 440
22.4.8. **Transformer Encoder** . . . 442
22.4.9. **Structure of the Transformer Encoder** . . . 443
22.4.10. **Why Residual Connections?** . . . 443
22.4.11. **Importance of Layer Normalization** . . . 444
22.4.12. **Feed-Forward Network (FFN) in Transformers** . . . 444
22.4.13. **Implementation Considerations** . . . 445
22.4.14. **Positional Encoding** . . . 447
22.4.15. **PyTorch Lightning Module** . . . 450
22.5. **Experiments** . . . 453
22.5.1. **Sequence-to-Sequence Learning** . . . 454
22.6. **Training our Model** . . . 457
22.7. **The Attention Mask** . . . 459
22.8. **Homework (Optional)** . . . 461

23. Transformer from Scratch **463**
23.0.1. **Notebook Overview** . . . 463
23.1. **Necessary Imports** . . . 463
23.2. **Masking in self-attention** . . . 464
23.3. **The Self-Attention Module** . . . 464
23.4. **Sinusoidal positional encoding** . . . 466
23.5. **The Encoder** . . . 466
23.6. **The Decoder** . . . 468
23.7. **Putting it All Together** . . . 471
23.7.1. **Testing on Dummy Data** . . . 472
23.8. **Training the Transformer Model** . . . 473
23.8.1. **Utility Functions** . . . 474
23.9. **Creating our PyTorch Dataset and DataLoader** . . . 475
23.9.1. **Model Hyperparameters** . . . 475
23.9.2. **Testing our Model with a Dummy Tensor** . . . 476
23.10. **Training Loop** . . . 476
23.11. **Testing our Model** . . . 477
23.12. **Homework (Optional)** . . . 477

24. NMT with Transformers **479**
24.1. **Necessary Imports** . . . 479
24.2. **Data** . . . 480
24.3. **Data Exploration** . . . 481
24.4. **Creating a Joint Dataset of Translations** . . . 482
24.5. **Save the New Subset** . . . 484
24.6. **Let's Load it Back in** . . . 484
24.7. **Creating our Vocabulary** . . . 484
24.8. **Tokenization** . . . 486
24.8.1. **Eliminating Duplicate Entries** . . . 488
24.9. **Splitting the Date** . . . 490
24.10. **Padding Sequences and Converting them to Tensors** . . . 492
24.11. **Creating a Dataset and a Dataloader** . . . 494
24.12. **From RNN seq2seq to transformer seq2seq** . . . 496
24.13. **Model** . . . 497
24.14. **Encoder** . . . 497
24.15. **Decoder** . . . 498
24.16. **Seq2Seq** . . . 500
24.17. **Training** . . . 504
24.18. **Main Training Loop** . . . 507
24.19. **Plotting our Results** . . . 508
24.20. **Testing and Evaluating our Model** . . . 509
24.21. **Homework (Optional)** . . . 511

25. Hugging Face Transformers and Gradio **512**
25.1. **Necessary Imports** . . . 512
25.2. **The Hugging Face ecosystem** . . . 513
25.3. **Exploring the Hugging Face Website** . . . 513
25.4. **Exploring a Model** . . . 513
25.5. **Hugging Face Datasets** . . . 523
25.6. **Hugging Face Spaces** . . . 524
25.7. **Background Removal Tool** . . . 524
25.7.1. **Downloading a Test Image** . . . 525
25.8. **Adding a Gradio Layer on Top** . . . 527
25.9. **Gradio UI Setup** . . . 527
25.10. **Text-to-Image with Diffusion Upscaler** . . . 528
25.11. **QR Code Generator with Diffusion** . . . 532
25.12. **Let's Do that in Gradio as well** . . . 536
25.13. **Homework (Optional)** . . . 537

26. The Vision Transformer (ViT) **539**
26.1. **Necessary Imports** . . . 539
26.2. **Loading Our Dataset** . . . 540
26.3. **Patch embedding and the [CLS] token** . . . 542
26.4. **Transformers for Image Classification** . . . 542
26.4.1. **How does the Vision Transformer work?** . . . 543
26.4.2. **Building the Transformer Model** . . . 544
26.4.3. **Pre-LN Transformer Block Implementation** . . . 545
26.4.4. **Building the Vision Transformer (ViT)** . . . 546
26.4.5. **Integrating Vision Transformer into PyTorch Lightning** . . . 547
26.5. **Experiments** . . . 548
26.5.1. **Training Function** . . . 548
26.5.2. **Hyperparameter Selection & Intuition** . . . 549
26.5.3. **Patch Size** . . . 549
26.5.4. **Embedding & Hidden Dimensionality** . . . 549
26.5.5. **Learning Rate & Regularization** . . . 550
26.5.6. **Exploring Hyperparameters** . . . 550
26.5.7. **Vision Transformer vs. CNN Performance on CIFAR10** . . . 551
26.5.8. **Key Observations** . . . 551
26.5.9. **Training Curve Comparison** . . . 551
26.5.10. **Comparing Vision Transformer and ResNet on CIFAR10** . . . 552
26.5.11. **Key Observations from Training Curves** . . . 552
26.5.12. **Why Does This Happen?** . . . 552
26.5.13. **Why Do Vision Transformers Work So Well on Large Datasets?** . . . 553
26.6. **Testing Our Model** . . . 553
26.7. **Homework (Optional)** . . . 554

27. Superpixel Tokenization for ViTs **556**
27.1. **Necessary Imports** . . . 556
27.2. **Loading Our Dataset** . . . 557
27.3. **Why superpixels? Grid patches vs. semantic regions** . . . 558
27.4. **Transformers for Image Classification** . . . 558
27.4.1. **Limitations** . . . 558
27.4.2. **Smarter Tokenization** . . . 559
27.5. **Grid-Based Patching Function** . . . 559
27.6. **Soft Assignment Clustering for Superpixel Tokenization** . . . 565
27.7. **Simple Linear Iterative Clustering (SLIC)** . . . 572
27.7.1. **1. Regular Grid Initialization** . . . 572
27.7.2. **2. Local Search for Compact Clusters** . . . 572
27.7.3. **3. Distance Metric for Pixel-Cluster Association** . . . 572
27.7.4. **4. Iterative Refinement for Precise Clustering** . . . 572
27.8. **An End-2-End Solution (Soft Assignment Differentiable Clustering for Superpixel Tokenization)** . . . 577
27.9. **Soft Assignment Differentiable Clustering for Superpixel Tokenization (diffSLIC)** 577
27.9.1. `compute_stride_and_padding` . . . 577
27.9.2. `spixel_upsampling` and `spixel_downsampling` . . . 577
27.9.3. `compute_elem_to_center_assignment` and `compute_center_to_elem_assignment` . . . 579
27.9.4. `update_clst_feats` . . . 579
27.9.5. `DiffSLIC` Module . . . 582
27.9.6. **Utility Functions for Visualization** . . . 584
27.10. **Homework (Optional)** . . . 587

28. Monocular Depth Estimation **588**
28.1. **1. Setup & Data Download** . . . 588
28.1.1. **1.1 Mount Drive & Install Libraries** . . . 588
28.1.2. **1.2 Download PASCAL VOC 2012 [6]** . . . 589
28.1.3. **1.3 Quick Data Inspection** . . . 589
28.2. **2. Monocular Depth Estimation** . . . 591
28.2.1. **2.1 Understanding the Challenge** . . . 591
28.3. **Relative vs. absolute depth** . . . 591
28.3.1. **2.2 MiDaS Architecture** . . . 591
28.3.2. **2.3 Inference and Visualization** . . . 592
28.4. **Homework (Optional)** . . . 594

29. Optical Flow **595**
29.1. **1. Setup & Data Download** . . . 595
29.1.1. **1.1 Mount Drive & Install Libraries** . . . 595
29.1.2. **1.2 Download PASCAL VOC 2012 [6]** . . . 596
29.1.3. **1.3 Quick Data Inspection** . . . 596

29.2. **2. Optical Flow with RAFT** . . . 598
29.2.1. **2.1 Introduction** . . . 598
29.2.2. **2.2 Prepare Video and Frames** . . . 598
29.3. **What RAFT learned that Lucas-Kanade could not** . . . 599
29.4. **2.3 Load RAFT Model and Preprocessing** . . . 599
29.5. **2.4 Compute and Visualize Optical Flow** . . . 600
29.6. **2.5 Visualizing Optical Flow Results** . . . 600
29.7. **Homework (Optional)** . . . 602

30. Text-Guided Image Inpainting **603**
30.1. **1. Setup & Data Download** . . . 603
30.1.1. **1.1 Mount Drive & Install Libraries** . . . 603
30.1.2. **1.2 Download PASCAL VOC 2012 [6]** . . . 604
30.1.3. **1.3 Quick Data Inspection** . . . 604
30.2. **2. Text-Guided Inpainting with Stable Diffusion** . . . 606
30.3. **2.1 Introduction** . . . 606
30.4. **Mask quality and prompt engineering** . . . 606
30.5. **2.2 Loading the Pretrained Inpainting Pipeline** . . . 607
30.6. **2.3 Preprocessing** . . . 609
30.7. **2.4 Hyperparameters** . . . 610
30.8. **2.5 Running the pipeline** . . . 610
30.9. **2.6 Visualization** . . . 611
30.10. **Homework (Optional)** . . . 612

31. Generative Modelling **613**
31.1. **Necessary Imports** . . . 613
31.2. **Loading Our Dataset** . . . 614
31.3. **2. Autoencoders** . . . 615
31.3.1. **2.1 Understanding the Architecture** . . . 615
31.3.2. **2.2 Autoencoder Implementation** . . . 616
31.3.3. **2.3 Training the Autoencoder** . . . 619
31.3.4. **2.4 Visualization & Latent Space Analysis** . . . 626
31.3.5. **2.5 Latent Space Interpolation** . . . 628
31.4. **3. Variational Autoencoders (VAE)** . . . 629
31.4.1. **3.1 Key Concepts** . . . 629
31.4.2. **3.2 VAE Implementation** . . . 630
31.4.3. **3.3 Training the VAE** . . . 633
31.4.4. **3.4 VAE Visualization & Comparison** . . . 638
31.5. **4. Generative Adversarial Networks (GANs)** . . . 641
31.5.1. **4.1 DCGAN Architecture** . . . 641
31.5.2. **4.2 GAN Training Loop** . . . 643
31.5.3. **4.3 Training Visualization** . . . 652

31.6. **5. Diffusion Models & Stable Diffusion (Optional)** . . . 655
31.6.1. **5.1 DDPM Architecture** . . . 655
31.6.2. **5.2 Gaussian Diffusion Process** . . . 658
31.6.3. **5.3 Model Initialization** . . . 660
31.6.4. **5.4 Training the DDPM** . . . 661
31.6.5. **5.5 Visualizing the Forward Diffusion** . . . 665
31.6.6. **5.6 Training Progress** . . . 665
31.6.7. **5.7 Generating Samples with DDPM** . . . 666
31.6.8. **5.8 Accelerated Sampling with DDIM** . . . 667
31.6.9. **5.9 Visualizing the Reverse Diffusion** . . . 668
31.6.10. **HOMEWORK (Optional)** . . . 669

32. Neural Rendering **671**
32.1. **Necessary Imports** . . . 671
32.2. **Loading Our Dataset** . . . 671
32.2.1. **1.3 Core Reconstruction Functions** . . . 672
32.2.2. **1.4 Incremental SfM Pipeline** . . . 674
32.2.3. **1.5 Visualization of the Reconstructed Scene** . . . 676
32.3. **2. Neural Radiance Fields (NeRF)** . . . 680
32.3.1. **2.1 Introduction to NeRF** . . . 680
32.3.2. **2.2 NeRF Setup** . . . 681
32.3.3. **2.3 The Core NeRF Model + Positional Encoding** . . . 683
32.3.4. **2.4 Ray Generation & Volumetric Rendering** . . . 684
32.3.5. **2.5 NeRF Training Loop** . . . 685
32.3.6. **2.6 Training Plots and Final Visualization** . . . 699
32.4. **3. Gaussian Splatting** . . . 701
32.4.1. **3.1 Introduction to 3D Gaussian Splatting** . . . 701
32.4.2. **3.2 Setup for Gaussian Splatting** . . . 701
32.4.3. **3.3 Interactive Web Viewer** . . . 702
32.4.4. **3.4 Analysis: Real-Time vs. Offline Rendering** . . . 702
32.4.5. **HOMEWORK (Optional)** . . . 703

II. Appendices **704**

33. Datasets and Pretrained Models **705**

References **714**

Part I.

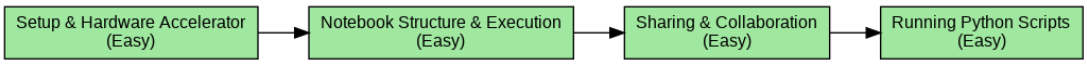
Practices

Chapter 1

Python and Google Colaboratory

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1PCTkAYD_D-EB0VnQjxYjivGSuscXBKxl?usp=sharing

Google Colab is a cloud-based platform that allows you to write and execute Python code in interactive Jupyter notebooks, eliminating the need for local installations. It seamlessly integrates with popular Python libraries and offers free access to computational resources like GPUs and TPUs.



1.1. Using Google Colaboratory

We are using Python version 3.11

Setting Google Colab to the correct version: Edit –> Notebook Settings –> Runtime type –> Python 3

Google Colab edits Jupyter notebooks which have a .ipynb extension. A Jupyter notebook is made up of code blocks and text blocks. The code blocks can be run one at a time or all at once.

Running a single block: done by clicking the button that appears in the top left corner of the block after selecting it (or by selecting the block and pressing Ctrl+Enter).

Running the entire notebook: done by clicking 'Run all' under the Runtime menu (or pressing Ctrl+F9).

If a code block has any printable/plottable output (by explicit print function usage, plotting, or the result of the last standalone expression) it will appear under the block. If an error occurs during the run of a block, the notebook's evaluation stops, and the error is printed as well.

The code's execution is done in a runtime environment that is automatically connected to the notebook. Its status can be seen in the top right corner, eg. "Connected".

Much like Google Docs, a notebook can be shared with either **view only** or **can edit** permissions. If the notebook is shared with someone using the **view only** permission, then they cannot make any edits to or run the notebook. They can only see the text and code blocks with the result of the last run under each code block. Someone with the **view only** permission can, of course, copy the notebook to their own Drive and edit or run it there (File –> Save a copy in Drive). A **view only** notebook can be opened in **playground mode**: in this case, a temporary copy is made of the notebook that is editable, however, if we do not save it to our Drive before quitting, then all changes will be lost.

The notebooks can be downloaded and used with a personal Jupyter runtime environment. The .ipynb files can be converted to .html files with Jupyter to be viewed offline without any runtime environment.

Running a Jupyter notebook

Jupyter notebooks work similarly to the Python interpreter. After the notebook runs, the declared variables in its code blocks will remain as long as the runtime environment is connected. If we were to rerun a previously run block, its result can be different than when we ran it the first time. For example, if in one block we declare and define a variable with a starter value, and then in the next block we raise this value, and the latter one runs twice, the value of this variable will be different than it would be if it was run only once. The "Runtime" menu's "Restart runtime" option will **restart the runtime environment* so we can avoid such problems.

Google Colab runtime environment settings:

Google provides a graphics card for free to speed up notebooks. The graphics card acceleration however only works in libraries that use it, such as Tensorflow or PyTorch. A simple Python/Numpy code's runtime will not be changed by this function in Colab. This graphics card acceleration can be activated at: Edit –> Notebook Settings –> Hardware accelerator –> GPU.

Python cheat sheet:

https://perso.limsi.fr/pointal/_media/python:cours:mementopython3-english.pdf

1.2. Colab runtimes and mounting Google Drive

Google Colab notebooks run on an ephemeral virtual machine. Anything written to `/content/` is deleted when the runtime disconnects. For course work and experiments you will want two things: a GPU runtime and persistent storage on Google Drive.

Selecting a GPU: *Runtime -> Change runtime type -> Hardware accelerator -> GPU*. The free tier usually provides a Tesla T4 with 16 GB of memory, plenty for the exercises in this course.

Mounting Drive: the snippet below binds your Google Drive to `/content/drive/MyDrive/`. You will be asked to authenticate once per runtime. Files placed there persist across sessions and are available to every Colab notebook you open.

```
from google.colab import drive
drive.mount('/content/drive')

# Convention used throughout the course: store datasets and checkpoints under
# a per-course folder so paths stay stable across notebooks.
COURSE_DIR = '/content/drive/MyDrive/DeepNetworkDevelopment'
import os
os.makedirs(COURSE_DIR, exist_ok=True)
print('Ready:', COURSE_DIR)
```

1.3. Running Python scripts

Although on these lectures we will only use the interpreter style Google Colaboratory, the simplest way of using Python is running Python script files. Script files conventionally have the '.py' extension. Running them requires a local or remote machine with an installed Python environment. On windows, WinPython and Anaconda distributions are the most common ones, and the latter is recommended. On Linux, Python

2.7 is typically part of the system, but installing Python 3 is done by using the Linux distribution's package manager (For example on Ubuntu using apt-get). A script can be run on Windows using cmd or Anaconda's command line. On Linux it can be run using a bash, like so:

```
python3 script.py
```

This will run the script, after which the run of the Python runtime environment will end too, unlike with an interpreter (such as Jupyter/Google Colab), so in this case, defined variables will not remain.

1.4. Basics Comments and Print

```
# this is a comment

# Ctrl+/ for multi line comment

'''
A block comment.
'''

print("Hello World!")
4+7
3+5  # result of the last standalone expression is automatically printed
```

```
Hello World!
```

```
8
```

1.4.1. Numbers and variables

Variables in Python: In Python, the type of variables is not bound and does not need an explicit declaration. Variables always store the memory address of an object, so its type is the type of that object. The memory space needed to store a new object is reserved automatically, there is no need for an explicit declaration to reserve that memory.

```
# Simple arithmetic with variables you'll see throughout the course.
batch_size = 32
num_classes = 10
learning_rate = 1e-3

print('Batch size:', batch_size)
print('Classes per batch (upper bound):', batch_size * num_classes)
print('Learning rate (float division):', 1 / 1000)
print('Integer division on epochs:', 100 // 8)         # how many batches of 8 fit in 100 samples
print('Modulo for batch remainder:', 100 % 8)
print('Power, weights in a 2**k layer:', 2 ** 10)
```

```
print('Variables with meaningful names for ML:')

batch_size = 32        # int
learning_rate = 1e-3   # float
decay = 0.99           # float
epoch = 0              # int
model_name = 'resnet18'  # str

# Mixed-type arithmetic: int + float -> float, same as before
effective_lr = learning_rate * decay
```

```
print('learning_rate * decay:', effective_lr, '->', type(effective_lr).__name__)

# Bigint: Python has no overflow, which matters for large factorial-like counts
num_combinations = 2 ** 128
print('2**128 fits natively:', num_combinations)
```

```
print("\nPython integer implementation is practically unlimited (bigint), no overflow happens:")

s = 1234567787978
print(s ** 10)
```

```
Python integer implementation is practically unlimited (bigint), no overflow
happens:
822525579429579674081821836294642674122858351968982549868076061993423511789502178148295905757434450711811492789236183142 4
```

1.4.2. Strings

```
# In Python strings can be specified in many ways,
#   but they all instantiate objects of the same str class

print("string 1") # double quoted string
print('string 2') # single quoted string, same as the two above

print('''
poem line#1
poem line#2
''')  # multiple lined string

print('''hey''' + "ho") # the usual string operations work, e.g. concatenation, more on them later

j = 'string are just "strings",'
h = " also strings are 'immutable'" # you can't change a string's value, only the reference to the string
f = j + h # String concatenation
print(f)

print("\nMultiplication: ")
ff = j*2
print(ff)
```

```
string 1
string 2

poem line#1
poem line#2

heyho
string are just "strings", also strings are 'immutable'

Multiplication:
string are just "strings",string are just "strings",
```

1.4.3. Logical expressions

```
print("Boolean operators: ")
print("True and False:", True and False) # Python Booleans are True and False
print("True or False:", True or False) # or keyword
print("not False:", not False) # not keyword
```

```
print("The type of True: ", type(True), '\n')

print("Three is greater than two:", 3 > 2) # value inequality check
print("More complex expression:", 1 < 2 <= 3) # a<b<c => (a<b) and (b<c)
# a and b and c (a and b) and (b and c); a or b or c => (a or b) and (b or c)
print("3 equals 3:", 3 == 3) # value equality check
print("3 does not equal 3:", 3 != 3, '\n') # value inequality check

string1 = 'this is the comparable string'
string2 = string1   # string2 and string1 variables refer to the same object from now
string3 = 'this is the comparable string'
print("String operator '==' compares contents: ", string1 == string2)
print("String operator '==' compares contents: ", string1 == string3)
print("Operator 'is' tells whether they are the same object: ", string1 is string2)
print("Operator 'is' tells whether they are the same object: ", string1 is string3)
```

```
Boolean operators:
True and False: False
True or False: True
not False: True
The type of True:  <class 'bool'>

Three is greater than two: True
More complex expression: True
3 equals 3: True
3 does not equal 3: False

String operator '==' compares contents:  True
String operator '==' compares contents:  True
Operator 'is' tells whether they are the same object:  True
Operator 'is' tells whether they are the same object:  False
```

1.4.4. NoneType type

```
a = None
print(a)
print(a is None)  # the None literal is a singleton object
print(a is not None)  # the 'is not' expression is a single operator (negation of 'is')

print(type(None))
```

```
None
True
False
<class 'NoneType'>
```

1.5. Lists

The list can contain any arbitrary objects, even with mixed types.

Documentation: https://docs.python.org/3/tutorial/datastructures.html

Features of the Python list type:

The Python list in the default Python implementation (CPython) is a pointer array written in C. Its size grows dynamically: if the list's size reaches the size of the allocated memory, then a new, larger memory space will be allocated and the old pointers will be copied over to the new space.

Since the list is implemented as an array, the time complexity of accessing and overwriting an element is O(1), so it takes constant time. Adding an element to the end (`append()`, `extend()`) and deleting from the end (`pop(-1)`) takes constant time, not counting the case in which the allocated memory space gets full. Inserting or deleting the first item or one somewhere in the middle will have a time complexity of O(n) because these are not linked lists: every item, behind the inserted or deleted one in the middle, will need to be moved forwards or backwards.

```
class_names = ['cat', 'dog', 'bird', 'fish']
print('Class labels used by the dataset:', class_names)

empty_batch = []
print('An empty batch:', empty_batch, '\n')

copied = list(class_names)
print('Copy of class_names:', copied)
print('Is the copy the same object? ', copied is class_names)

batch = [[1.2, 3.4], [2.1, -0.5], [0.0, 1.0]]   # a 3-sample minibatch, 2 features each
print('A mini-batch as a list of feature vectors:', batch)
```

```
print("\nIndexing a list: ")
basic_list = [1,2,3,4,5]
print("A basic list:", basic_list)
print("Element at index 2 in the basic list (third element):", basic_list[2])
                                # Indexing starts with ZERO and goes to length-1

print("2nd element counting from the back of the basic list:", basic_list[-2])
                                # Negative index ''-k' corresponds to length-k

length = len(basic_list) # this is a function returning an int, the length of the given list
print(length, 'is the length of the jagged list: ', basic_list)
```

```
Indexing a list:
A basic list: [1, 2, 3, 4, 5]
Element at index 2 in the basic list (third element): 3
2nd element counting from the back of the basic list: 4
5 is the length of the jagged list:  [1, 2, 3, 4, 5]
```

```
print("\nMethods of the list class:")

basic_list.append(3)
print("Appended one element to the end: ", basic_list)

basic_list.append([3, 4])
print("Appended one element (a list with a length of two) to the end: ", basic_list)

basic_list.extend([3, 4])
print("Extended the list with two elements: ", basic_list)

basic_list.insert(0, None)
print("Inserted an element to the front of the list (index 0): ", basic_list)

basic_list.insert(-1, None)
print("Inserted an element to the back of the list (index -1): ", basic_list)
```

```
Methods of the list class:
Appended one element to the end:  [1, 2, 3, 4, 5, 3]
Appended one element (a list with a length of two) to the end:  [1, 2, 3, 4, 5,
```

```
3, [3, 4]]
Extended the list with two elements:  [1, 2, 3, 4, 5, 3, [3, 4], 3, 4]
Inserted an element to the front of the list (index 0):  [None, 1, 2, 3, 4, 5,
3, [3, 4], 3, 4]
Inserted an element to the back of the list (index -1):  [None, 1, 2, 3, 4, 5,
3, [3, 4], 3, None, 4]
```

Other list methods:

`pop()`, `remove()`, `clear()`, `sort()`, ...

```
print("\nSlice and stride: ")
# Format of list slicing: my_list[from:to:step]
#   iteration goes from 'from' (inclusive) to 'to' (exclusive) with a step size of 'step'
#   second colon (':') and any of 'from', 'to' or 'step' can be omitted

my_list = [0,1,2,3,4,5]
print("The original list: ", my_list)
print("From idx 2 to idx 4 (exclusive): ", my_list[2:4])
print("From idx 2 to idx 4 (exclusive) with step size 2: ", my_list[2:4:2])
print("From idx 2 to the end: ", my_list[2:])
print("From the beginning till idx 2 (exclusive): ", my_list[:2], '\n')

print("Whole list with step size 2: ", my_list[::2])
print("Reversed list: ", my_list[::-1])
print("Reversed list with step size 2: ", my_list[::-2])
print("Reversed list with step size 2, starting"\
            " from second but last element towards the beginning: ", my_list[-2::-2])

# when slicing the reversed list, the 'from' marks the latter index and
# 'to' is the smaller one, closer to the front as we iterate in the opposite direction
```

```
Slice and stride:
The original list:  [0, 1, 2, 3, 4, 5]
From idx 2 to idx 4 (exclusive):  [2, 3]
From idx 2 to idx 4 (exclusive) with step size 2:  [2]
From idx 2 to the end:  [2, 3, 4, 5]
From the beginning till idx 2 (exclusive):  [0, 1]

Whole list with step size 2:  [0, 2, 4]
Reversed list:  [5, 4, 3, 2, 1, 0]
Reversed list with step size 2:  [5, 3, 1]
Reversed list with step size 2, starting from second but last element towards
the beginning:  [4, 2, 0]
```

```
print("When slicing, out of range indices will not raise an error: ", my_list[:8])
print("When slicing, out of range indices will not raise an error: ", my_list[7:8])

# indexing with a single out of range index would of course cause an error

print("\nString is also a sequence type:")
s = 'abcdef'
print(s[2])
print(s[-1])
print(s[:4])
print(s[-2:1:-1])
```

```
When slicing, out of range indices will not raise an error:  [0, 1, 2, 3, 4, 5]
When slicing, out of range indices will not raise an error:  []

String is also a sequence type:
c
f
abcd
edc
```

1.6. If/else statements, loops, the pass keyword

Python blocks are indicated using whitespace characters: after `for`, `while`, `if`, `else`, `elif` an indented block must follow. Indentation means adding to the number of whitespace characters at the beginning of the line. Each statement in a block must start its line with the same amount of whitespace characters. A whitespace character can be a tabulation or space, but they cannot be mixed. (Google Colab replaces tabs with spaces).

```
print("\nIf statements: ")
ch = 'a'
if ch == 'a':
    print(1)

if ch == 'a':
  print(2)
elif ch == 'a':
  print(3)

if ch == 'b':
  print(4)
else:
  print(5)

if ch == 'd':
  print(6)
elif ch == 'a':
  print(7)
else:
  print(8)
```

```
If statements:
1
2
5
7
```

```
print('\nTraining loop skeleton, iterate over epochs:')
for epoch in range(0, 5):
    print(f'  epoch {epoch+1}/5')

total_samples_seen = 0
for step in range(20):
    total_samples_seen += 32   # each step processes a batch of 32
print('Total samples seen after 20 steps:', total_samples_seen)

# range(a, b, step), every second epoch for evaluation
```

```
print('\nEvaluate every 2 epochs:')
for eval_epoch in range(0, 10, 2):
    print('  evaluating at epoch', eval_epoch)
```

```
print("\nWhile loop: ")
i = 0
while(i < 5):
    print('  ', i)
    i +=1

for i in range(3):
    print('\nPass is the empty statement')
    pass    # do nothing: empty indented block results in a syntax error
            #    this is why 'pass' exists
```

```
While loop:
   0
   1
   2
   3
   4

Pass is the empty statement

Pass is the empty statement

Pass is the empty statement
```

1.7. Homework (Optional)

1. **Mount Drive and persist a file.** Open a fresh Colab notebook, mount Google Drive, and write a short text file to `/content/drive/MyDrive/DeepNetworkDevelopment/hello.txt`. Disconnect the runtime, reconnect, and confirm the file is still there.
2. **Training-loop skeleton.** Write a `for epoch in range(n_epochs)` loop that prints a decaying learning rate at each epoch using `lr * (0.95 ** epoch)`. Use `n_epochs = 10` and `lr = 1e-3`.
3. **String formatting with f-strings.** Given `model_name = 'resnet18 '`, `accuracy = 0.9123`, and `step = 450`, print a single line of the form `[step 450] resnet18 acc=91.23%` using an f-string.
4. **List slicing on a feature vector.** Create `features = [round(i * 0.1, 2) for i in range(20)]`. Extract every third element starting from index 2. Then reverse the result.
5. **`if`/`else` with boolean operators.** Given `accuracy = 0.87` and `patience_left = 2`, write an `if` statement that prints "stop early" when accuracy is below 0.9 and patience is zero, "continue" otherwise.

Chapter 2

Python Fundamentals

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/11Gb2R5vjR7_u1v0EQ27ILEW5tNggQ-kr?usp=sharing

This notebook provides a comprehensive overview of `Python fundamentals`, covering topics such as defining functions (with return values, optional parameters, recursion, nested functions, and lambda expressions), as well as techniques for passing functions as arguments. It also explores essential data structures like `tuples, sets, and dictionaries, shows data manipulation using list, set, and dictionary` comprehensions, and introduces error handling with assertions.



2.1. Functions

Documentation: https://docs.python.org/3.6/tutorial/controlflow.html#defining-functions - Chapters 4.6 and 4.7

2.1.1. Simple functions, `return`

```
def procedure1(my_param): # if no return statement, returns None in the end
    print(my_param)

def function_double(my_param): # type of 'my_param' is not restricted
    return my_param*2

def function_swap(arg1, arg2): # Multiple returns: returns a tuple with a length of 2
    return arg2, arg1

print("What do they return? ")
print("If no return statement, then returns None: ", procedure1("print this!"))
print("Return number argument*2: ", function_double(2))
print("Return string argument*2: ", function_double("two"))
print("Swap arguments: ", function_swap("first", "second"))

result = function_swap("first", "second")
print(type(result))
```

```
What do they return?
print this!
If no return statement, then returns None:  None
Return number argument*2:  4
Return string argument*2:  twotwo
```

```
Swap arguments:  ('second', 'first')
<class 'tuple'>
```

2.1.2. Optional parameters

```
for i in range (0,10):
    print(str(i).zfill(3))
```

```
000
001
002
003
004
005
006
007
008
009
```

```
# Default-argument pattern shows up in every training loop:
def decay_lr(lr, factor=0.95):
    return lr * factor

print('Start LR:', 1e-3)
print('After one epoch:', decay_lr(1e-3))
print('Custom factor:', decay_lr(1e-3, factor=0.5))

# Stepping a learning rate schedule
def step_lr(lr, step, milestone=10, gamma=0.1):
    return lr * gamma if step >= milestone else lr

print('\nStep LR at step 5 (no decay yet):', step_lr(0.1, step=5))
print('Step LR at step 12 (decayed):', step_lr(0.1, step=12))
```

```
def append_1(arr=[]):
    arr.append(1)
    return arr

a = [2, 3, 5]
append_1(a)
```

```
[2, 3, 5, 1]
```

```
append_1()
```

```
[1]
```

```
append_1()
```

```
[1, 1]
```

2.1.3. Recursion

```
def factorial(n):
    product = 1
    for i in range(1,n+1):
        product *= i
    return product
```

```
print("Factorial implemented with a loop: ", factorial(6))

def factorial_rec(n):
    if n <= 0:
        return 1
    else:
        return n*factorial_rec(n-1)

print("Factorial implemented in a recursive way: ", factorial_rec(6))

def factorial_rec2(n):
    return 1 if n <= 0 else n*factorial_rec(n-1)

print("Factorial implemented in a recursive way, shorter version: ", factorial_rec2(6))
```

```
Factorial implemented with a loop:  720
Factorial implemented in a recursive way:  720
Factorial implemented in a recursive way, shorter version:  720
```

2.1.4. Nested function

```
def absolute(x):
    def absolute_positive(x):  # functions can be defined in functions, to any level of depth
        return x

    def absolute_negative(x):
        return -x

    if x < 0:
        return absolute_negative(x)
    else:
        return absolute_positive(x)

print("Nested absolute value function: ", absolute(-3.5))
print("Nested absolute value function: ", absolute(2))
```

```
Nested absolute value function:  3.5
Nested absolute value function:  2
```

2.1.5. Passing function as an argument

```
def addition(a, b):
    return a+b

def multiplication(a, b):
    return a*b

def apply_twice(func, a, b, c):
    return func(func(a, b), c)

a = addition
print(a(3,4))
print("Applying addition() twice: ", apply_twice(addition, 2, 3, 4))
print("Applying multiplication() twice: ", apply_twice(multiplication, 2, 3, 4))
print("Applying built-in max() function twice: ", apply_twice(max, 2, 3, 4))
```

```
7
Applying addition() twice:  9
```

```
Applying multiplication() twice:  24
Applying built-in max() function twice:  4
```

2.1.6. Anonymous functions (`lambda`)

```python
# lambdas are common in ML code for tiny loss/metric functions.
relu        = lambda x: x if x > 0 else 0
square      = lambda x: x * x
mse_single  = lambda y_hat, y: (y_hat - y) ** 2

print('relu(-2.5):', relu(-2.5))
print('relu( 3.7):', relu(3.7))
print('square(4):', square(4))
print('MSE(y_hat=3.0, y=2.5):', mse_single(3.0, 2.5))

# Passing a lambda as an argument, the apply_fn pattern
def apply_to_pairs(fn, pairs):
    return [fn(a, b) for a, b in pairs]

predictions = [(3.0, 2.5), (1.0, 1.0), (0.5, 0.9)]
print('\nMSE per sample:', apply_to_pairs(mse_single, predictions))
```

2.2. Data structures

We discussed the `list` type in the previous chapter.

We will discuss the `tuple`, `set`, `dict` types below.

Documentation: https://docs.python.org/3.6/tutorial/datastructures.html

Other built-in container types can be found in the `collections` module: https://docs.python.org/3/library/collections.html

2.2.1. `tuple` type

Immutable type (its state cannot be modified after it's created).

```python
print("Tuples:")

t1 = (2,3,"a",(4,5))
t2 = ("abc",)   # tuple with a length of 1: a comma is needed
not_a_tuple = ("abc")    # NOT a tuple, but a string in a pair of parentheses
t3 = ()    # zero length tuple

print("A tuple with four elements: ", t1)
print("A tuple with a single element: ", t2)
print("This is not a tuple: ", not_a_tuple)
print("A zero length tuple: ", t3)

# Invalid operation: t1[0] = 1  -> 'TypeError: 'tuple' object does not support item assignment'
#    -> tuple is IMMUTABLE, it cannot be changed after creation

t4 = (2,3,[],t2)
print("\nThe tuple: ", t4)
t4[2].append(1) # the 2. elements is a list it can changed but the elements can be changed
t4[2].append("abc") #
t4[2].remove(1)
```

```
t4[2].insert(0,0)
print("Adding/removing/inserting an element to the list inside the tuple: ", t4)
```

```
Tuples:
A tuple with four elements:  (2, 3, 'a', (4, 5))
A tuple with a single element:  ('abc',)
This is not a tuple:  abc
A zero length tuple:  ()

The tuple:  (2, 3, [], ('abc',))
Adding/removing/inserting an element to the list inside the tuple:  (2, 3, [0,
'abc'], ('abc',))
```

2.2.2. Packing and unpacking tuples

```
# t5 = 1, 2, "asd", [34,54], (2,(1,))
# print("Packed 5 elements into a tuple: ", t5)

def func_swap(x, y):
  return y, x

t6 = func_swap(2,7)
print("\nThe function returned two elements which were automatically packed into a tuple: ", t6)

aa, bb = func_swap(3,8)
print("The function returned two elements which were put in two separate variables: ")
print("    ", aa)
print("    ", bb)

a1, b1 = t6
print("\nUnpacked a 2 long tuple into two variables: ")
print("    ", a1)
print("    ", b1)

print("\nLength of a tuple: ", len(t6))

print("\nIterating a tuple the same way as a list: ")
for item in t6:
  print("    ", item)
```

```
The function returned two elements which were automatically packed into a tuple:
(7, 2)
The function returned two elements which were put in two separate variables:
     8
     3

Unpacked a 2 long tuple into two variables:
     7
     2

Length of a tuple:  2

Iterating a tuple the same way as a list:
     7
     2
```

```
my_list = ['abc', -1.5, [None]]
a1, a2, a3 = my_list
print("Any sequence can be unpacked this way (if length is known), not only a tuple: ")
print("    ", a1)
print("    ", a2)
print("    ", a3)
```

```
Any sequence can be unpacked this way (if length is known), not only a tuple:
     abc
     -1.5
     [None]
```

If we want to pass the elements of a tuple when calling a function as different arguments of that function then the **unpack operator** might help.

More details at https://realpython.com/python-kwargs-and-args/

2.2.3. `set` type

In a set, every element appears once. It can contain mixed types of objects as well but only immutable ones (such as numbers, strings, NoneType, tuple, . . .).

Checking if a set contains an element by the implementation of a hash function has a time complexity of **O(1)**. This same operation on a list would have a time of **O(n)**.

```
print("Set objects: ")

# s1 = {}  # -> this creates an empty dict object, not an empty set
s1 = set()  # to create an empty set, the constructor must be used
print("An empty set: ", s1)

s2 = {1,2,3,2,("a",),1,None,3,3,None,3,"adab",("a",),"a"}
print("Duplicate elements are stored once in the set: ", s2)

# s3 = {2, [2]}: 'TypeError: unhashable type: 'list'' -> only immutable types can be put in a set.

list1 = [7,1,1,1,2,5,7,8]
tup1 = (3,1,1,2,2,3,3)
str1 = "abcabd"
s6 = set(list1)
s7 = set(tup1)
s8 = set(str1)

print("\nA set constructed from a list: ", s6)
print('A set constructed from a tuple: ', s7)
print('A set constructed from the characters of a string (as string is iterable): ', s8)

# NO GUARANTEE about the order of iteration!
```

```
Set objects:
An empty set:  set()
Duplicate elements are stored once in the set:  {1, 'adab', 3, 2, ('a',), 'a',
None}

A set constructed from a list:  {1, 2, 5, 7, 8}
A set constructed from a tuple:  {1, 2, 3}
```

```
A set constructed from the characters of a string (as string is iterable):
{'c', 'a', 'd', 'b'}
```

2.2.4. Set operations

```
print("Length of a set: ", len(s2))

# It is much faster to check a large set for an element than to check a list
print("Is 'adab' in the set? ", 'adab' in s2)
print("Is 'adab' not in the set? ", 'adab' not in s2)   # 'not in' is a single operator!
print("Is 'asd' in the set? ", 'asd' in s2)

print('\nUnion: ', s6 | s7)
print('Intersection: ', s6 & s7)
print('Symmetric difference (items present in exactly one set): ', s6 ^ s7)
print('Subtraction (tems in set#1 that are not in set#2)', s6 - s7)

print("\nModifying a set: ")
print("The set: ", s6)
s6.add('adab')
print("The set after 'adab' was added: ", s6)
s6.add(1)  # already present, not added
print("The set after 1 was added (unchanged since 1 was already in the set): ", s6)
s6.remove('adab')
print("The set after 'adab' was removed: ", s6)
s6.update(['23', 23, 23, (23,)])   # adding all elements from a list, set, or other sequence
print("The set after adding multiple elements: ", s6)

print("\nIterating a set, order is not guaranteed:")
for my_item in s8:
    print('    ', my_item)
```

```
Length of a set:  7
Is 'adab' in the set?  True
Is 'adab' not in the set?  False
Is 'asd' in the set?  False

Union:  {1, 2, 3, 5, 7, 8}
Intersection:  {1, 2}
Symmetric difference (items present in exactly one set):  {3, 5, 7, 8}
Subtraction (tems in set#1 that are not in set#2) {8, 5, 7}

Modifying a set:
The set:  {1, 2, 5, 7, 8}
The set after 'adab' was added:  {1, 2, 'adab', 5, 7, 8}
The set after 1 was added (unchanged since 1 was already in the set):  {1, 2,
'adab', 5, 7, 8}
The set after 'adab' was removed:  {1, 2, 5, 7, 8}
The set after adding multiple elements:  {1, 2, 5, 7, 8, '23', (23,), 23}

Iterating a set, order is not guaranteed:
     c
     a
     d
     b
```

```
test_set = {"a","b","c","d","e",1,"f",2,"g",3,"h"}
test_set
```

```
{1, 2, 3, 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'}
```

2.2.5. `dict` type: dictionary (not only for words...)

Associates unique keys to values. It is best to think of it as a set of key-value pairs, where the keys must be unique. Keys can only be of an immutable type (for hashing purposes).

```
dict1 = {}   # an empty dictionary
dict2 = {1:['bla','bla'], 'asd':['blab'], ():[123, 456], (1,2,3):2}   # keys and values of different type are
  possible

print("An empty dictionary: ", dict1)
print("A dictionary: ", dict2)

print("An iterator over the keys of the dictionary: ", dict2.keys())
print("An iterator over the values of the dictionary: ", dict2.values())
print("An iterator over the (key, value) pairs of the dictionary: ", dict2.items())

dict4 = {'John':25, 'Eliza':32, 'Sam':59}
print("\nIteration through keys, order is not guaranteed:")
for key in dict4:
  print('    ', key)

print("\nIteration through (key, value) pairs, order is not guaranteed:")
for key, value in dict4.items():   # <-- unpacking (key, value) tuple into two variables
  print("\t", key, " -> ", value)
```

```
An empty dictionary:  {}
A dictionary:  {1: ['bla', 'bla'], 'asd': ['blab'], (): [123, 456], (1, 2, 3):
2}
An iterator over the keys of the dictionary:  dict_keys([1, 'asd', (), (1, 2,
3)])
An iterator over the values of the dictionary:  dict_values([['bla', 'bla'],
['blab'], [123, 456], 2])
An iterator over the (key, value) pairs of the dictionary:  dict_items([(1,
['bla', 'bla']), ('asd', ['blab']), ((), [123, 456]), ((1, 2, 3), 2)])

Iteration through keys, order is not guaranteed:
     John
     Eliza
     Sam

Iteration through (key, value) pairs, order is not guaranteed:
         John  ->  25
         Eliza  ->  32
         Sam  ->  59
```

Dictionary operations

```
print("Length of dict: ", len(dict4))
print("Get value for key 'John' from dict: ", dict4['John'])

print("Is 'Mary' among the dict keys? ", 'Mary' in dict4)
```

```
print("Is 'Eliza' not among the dict keys? ", "Eliza" not in dict4)

# modified the value of a key in dict
dict4['John'] += 10
print("\nIncrease value for key 'John' by 10: ", dict4)

# added new key (and value) to dict
dict4['Robert'] = 1
print("Added a new key to the dict with a value of 1: ", dict4)

# deleting key-value pair:
del dict4['John']     # the del statement is used for deletion of variables as well, we will discuss this later
print("Deleted 'John' from the dict: ", dict4)

# adding all elements of a dict to another one:
dict4.update(dict2)   # if keys are present in both dicts, the corresponding value in dict2 overwrites the
   value in dict4
print("Added all elements from another dict (values for existing keys were overwritten: ", dict4)

# !!! ORDER OF ITERAION through the dict items is NOT guaranteed to be either sorted by
#          keys or values or by item insertion date
# however, OrderedDict from collections module guarantees iteration order
#          to be the same as the order of item insertions to the OrderedDict
```

```
Length of dict:  3
Get value for key 'John' from dict:  25
Is 'Mary' among the dict keys?  False
Is 'Eliza' not among the dict keys?  False

Increase value for key 'John' by 10:  {'John': 35, 'Eliza': 32, 'Sam': 59}
Added a new key to the dict with a value of 1:  {'John': 35, 'Eliza': 32, 'Sam':
59, 'Robert': 1}
Deleted 'John' from the dict:  {'Eliza': 32, 'Sam': 59, 'Robert': 1}
Added all elements from another dict (values for existing keys were overwritten:
{'Eliza': 32, 'Sam': 59, 'Robert': 1, 1: ['bla', 'bla'], 'asd': ['blab'], ():
[123, 456], (1, 2, 3): 2}
```

2.3. Important built-in functions

2.3.1. The `enumerate()` function

Iterates through a sequence returning the elements and their indexes in order.

```
# enumerate() is the idiomatic way to iterate minibatches with an index
batch = [0.1, -0.3, 0.8, 0.2, -0.7]
print('Iterating a batch with indices:')
for i, value in enumerate(batch):
    print(f'  sample #{i}: {value:+.2f}')
```

2.3.2. The `zip()` function

Creates a sequence of pairs by aggregating two sequences (list, iterator, etc.). Its result is an iterator.

```
# zip() pairs features with labels, essential for supervised learning
features = [[1.2, 0.5], [0.3, -0.1], [-0.6, 2.1]]
labels   = ['cat', 'dog', 'cat']
```

```
print('Iterating (features, label) pairs:')
for feat, lab in zip(features, labels):
    print('  ', feat, '->', lab)

# A dictionary from two aligned sequences
model_scores = dict(zip(['acc', 'f1', 'precision', 'recall'],
                        [0.91, 0.89, 0.92, 0.87]))
print('\nModel scores as a dict:', model_scores)
```

2.3.3. The `min()`, `max()`, `abs()`, `round()`, `all()`, `any()` functions

```
my_list2 = [15, 25, 1.26, -56, 2., .3]
print("The minimum of a sequence: ", min(my_list2))
print("The maximum of a range iterator: ", max(range(3,11)))

print("\n1.5 rounded: ", round(1.5))

my_bools = [2 < 3, 4 < 5 <= 1, True]
print("\nA sequence of boolean expressions: ", my_bools)
print("Logical AND operator applied to a sequence of boolean expressions: ", all(my_bools))
print("Logical OR operator applied to a sequence of boolean expressions: ", any(my_bools))
```

```
The minimum of a sequence:  -56
The maximum of a range iterator:  10

1.5 rounded:  2

A sequence of boolean expressions:  [True, False, True]
Logical AND operator applied to a sequence of boolean expressions:  False
Logical OR operator applied to a sequence of boolean expressions:  True
```

2.3.4. The `map()` and `filter()` functions

`map()`: applies a function to all elements of a sequence

`filter()`: selects the elements of a sequence for which the given function returns true

```
my_list = [1,2,3]
my_doubled_sequence = map(lambda x: 2*x, my_list)  # named functions can be given as well
print("Iterate my doubled sequence:")
for item in my_doubled_sequence:
  print('    ', item)

print("\nThe iterators can be converted into lists: ", list(map(lambda x: 2*x, my_list)))

filtered_sequence = filter(lambda x: x > 2, my_list)  # only keep elements greater than 2
print("Convert the filtered sequence iterator into a list: ", list(filtered_sequence))
```

```
Iterate my doubled sequence:
     2
     4
     6

The iterators can be converted into lists:  [2, 4, 6]
Convert the filtered sequence iterator into a list:  [3]
```

2.4. List, Set, Dict comprehension

A concise one-line way of creating new lists/sets/dictionaries. Syntactic sugar.

```
print("List comprehensions: ")
# EXAMPLE#1, LONG VERSION:

my_list = [1.2, 2.2, -3., .5, 0.]
print("\nWe will double each item in the following list: ", my_list)

new_list = []
for item in my_list:
    new_list.append(item*2)

print("The result list created with a for loop: ", new_list)

# EXAMPLE#1, SHORT VERSION:
new_list = [item*2 for item in my_list]
print("The result list created with a list comprehension: ", new_list)

# EXAMPLE#2, LONG VERSION:
print("\nWe will double each positive number, set zeros to one, ...")
print("    ... and triple all negative numbers in the following list: ", my_list)

new_list = []
for item in my_list:
    if item > 0:
        new_list.append(item*2)
    elif item == 0.:
        new_list.append(1.)
    else:
        new_list.append(item*3)

print("The result list created with a for loop: ", new_list)

# EXAMPLE#2, SHORT VERSION:
new_list = [item*2 if item > 0 else 1. if item == 0. else item*3 for item in my_list]
print("The result list created with a list comprehension: ", new_list)
```

```
List comprehensions:

We will double each item in the following list:  [1.2, 2.2, -3.0, 0.5, 0.0]
The result list created with a for loop:  [2.4, 4.4, -6.0, 1.0, 0.0]
The result list created with a list comprehension:  [2.4, 4.4, -6.0, 1.0, 0.0]

We will double each positive number, set zeros to one, ...
    ... and triple all negative numbers in the following list:  [1.2, 2.2, -3.0,
0.5, 0.0]
The result list created with a for loop:  [2.4, 4.4, -9.0, 1.0, 1.0]
The result list created with a list comprehension:  [2.4, 4.4, -9.0, 1.0, 1.0]
```

```
# WHEN USING 'IF' WITH ONLY ONE BRANCH

# EXAMPLE#1, LONG VERSION:

my_list = [1.2, -2.2, -3., .5, 0.]
print("\nWe will select each negative item and double them from the following list: ", my_list)
```

```
new_list = []
for item in my_list:
  if item < 0:
    new_list.append(item*2)

print("The result list created with a for loop: ", new_list)

# EXAMPLE#1, SHORT VERSION:
new_list = [item*2 for item in my_list if item < 0]
print("The result list created with a list comprehension: ", new_list)
```

```
We will select each negative item and double them from the following list:
[1.2, -2.2, -3.0, 0.5, 0.0]
The result list created with a for loop:  [-4.4, -6.0]
The result list created with a list comprehension:  [-4.4, -6.0]
```

Two cases of using if statements in list comprehension: * If the if statement has an else branch and all branches, including the else branch, add a new element to the list/set/dictionary, then inside the comprehension all cycles that contain an if statement have their **for keyword after the if/elif/else keywords** e.g.:

```
new_list = []
for item in my_list:
  if item > 0:
    new_list.append(item*2)
  else:
    new_list.append(1.)
```

Rewritten:

```
new_list = [item*2 if item > 0 else 1. for item in my_list]
```

- If the if statement has no else branch, or not all of its branches add new elements to the list/set/dictionary, then inside the comprehension the **if keyword is after the for keyword of the cycle it is contained in**.

```
new_list = []
for item in my_list:
  if item > 0:
    new_list.append(item*2)
```

Rewritten:

```
new_list = [item*2 for item in my_list if item > 0]
```

```
# JAGGED LISTS (lists in lists)

print("\nJagged lists:")
my_jlist = [[2,3,-8],[4,-7,2],[],[5,6,-3],[1,5]]
print("\nWe double each negative item in the following list: ", my_jlist)

# EXAMPLE#1: modify jagged list: double negative elements

# LONG VERSION:
new_list = []
```

```
for item_list in my_jlist:
  new_sublist = []
  for item in item_list:
    if item < 0:
       new_sublist.append(item*2)
    else:
       new_sublist.append(item)
  new_list.append(new_sublist)

print("The result list created with a loop: ", new_list)

# SHORT VERSION:

new_list = [[item*2 if item < 0 else item for item in sublist] for sublist in my_jlist]
print("The result list created with a (nested) list comprehension: ", new_list)

# EXAMPLE#2: jagged list to flat list
print("\nWe create a flat list of items from the following jagged list: ", my_jlist)

# LONG VERSION:
new_list = []
for item_list in my_jlist:
  for item in item_list:
    new_list.append(item)

print("The result list created with a loop: ", new_list)

# SHORT VERSION:

new_list = [item for itemlist in my_jlist for item in itemlist]
print("The result list created with list comprehension: ", new_list)
```

```
Jagged lists:

We double each negative item in the following list:  [[2, 3, -8], [4, -7, 2],
[], [5, 6, -3], [1, 5]]
The result list created with a loop:  [[2, 3, -16], [4, -14, 2], [], [5, 6, -6],
[1, 5]]
The result list created with a (nested) list comprehension:  [[2, 3, -16], [4,
-14, 2], [], [5, 6, -6], [1, 5]]

We create a flat list of items from the following jagged list:  [[2, 3, -8], [4,
-7, 2], [], [5, 6, -3], [1, 5]]
The result list created with a loop:  [2, 3, -8, 4, -7, 2, 5, 6, -3, 1, 5]
The result list created with list comprehension:  [2, 3, -8, 4, -7, 2, 5, 6, -3,
1, 5]
```

Set and dictionary comprehension:

```
print("\nSet comprehension: ")

print("\nWe create a set from the absolute values of items in this list: ", new_list)
new_set = {abs(item) for item in new_list}
print("The result set created with set comprehension: ", new_set)

print("\nDict comprehension: ")

print("\nWe create a dict from item:'item' key-value pairs from this set: ", new_set)
```

```
new_dict = {k:str(k) for k in new_set} ## {1:str(1), 2:str(2),...}
print("The result dict created with dict comprehension: ", new_dict)
```

```
Set comprehension:

We create a set from the absolute values of items in this list:  [2, 3, -8, 4,
-7, 2, 5, 6, -3, 1, 5]
The result set created with set comprehension:  {1, 2, 3, 4, 5, 6, 7, 8}

Dict comprehension:

We create a dict from item:'item' key-value pairs from this set:  {1, 2, 3, 4,
5, 6, 7, 8}
The result dict created with dict comprehension:  {1: '1', 2: '2', 3: '3', 4:
'4', 5: '5', 6: '6', 7: '7', 8: '8'}
```

2.5. Classes

The concept of classes appears in many programming languages, in most cases as part of the object-oriented programming paradigm.

In an ideal case, a class contains the required fields and the methods for defining the behaviour of one implemented data structure or functionality. This way by initializing the class we get one instance of the object.

As an example, we can take a class that represents a car. The class is the prototype of the car as a concept. The instance is a concrete materialization of the car's prototype, meaning that it can be managed as a physical car (e.g.: Lazlo's car). The fields of the car class are the car's attributes, such as its current speed, weight, number of seats, number of passengers, etc. By creating an instance these fields get values, just how a real car has a speed that can be described by a number, has a weight and a number of seats and passengers. The current values of these fields make up the object's state. The methods of the car class used on the concrete instance can change the values of the fields (and thus change the state of the instance) or return them.

The documentation of advanced concepts such as inheritance, static variables/functions can be found here: https://docs.python.org/3/tutorial/classes.html

```
class Sample:
    '''A single labelled training example.'''

    def __init__(self, features, label):
        self.features = features
        self.label = label

    def flipped_label(self, num_classes):
        # Non-destructive label transformation (useful for data augmentation tricks)
        return (self.label + 1) % num_classes

    def norm(self):
        return sum(x * x for x in self.features) ** 0.5

s = Sample(features=[0.3, -0.4, 1.1], label=2)
```

```
print('Features:', s.features)
print('Label:', s.label)
print('Flipped label (10 classes):', s.flipped_label(num_classes=10))
print('L2 norm of features:', round(s.norm(), 4))
```

2.6. Type hints and dataclasses

Modern Python code uses *type hints* to communicate what functions expect and return. They are not enforced at runtime, they exist for readers, IDEs, and static checkers, but they make ML research code dramatically easier to follow, especially when tensors of different shapes and dtypes are passed around. The `@dataclass` decorator builds on type hints: annotating fields is enough to get a constructor, a readable `__repr__`, and equality comparison for free. Virtually every PyTorch-based codebase you will read in this course leans on `@dataclass`.

```
from typing import List, Tuple

def mean(xs: List[float]) -> float:
    '''Return the arithmetic mean of a list of floats.'''
    return sum(xs) / len(xs)

def split(xs: List[float], k: int) -> Tuple[List[float], List[float]]:
    '''Split xs into a k-sized head and the remainder.'''
    return xs[:k], xs[k:]

losses = [1.2, 0.9, 0.7, 0.5, 0.4]
print('Mean loss:', mean(losses))

first, rest = split(losses, 2)
print('First two:', first)
print('Rest:', rest)
```

```
from dataclasses import dataclass

@dataclass
class TrainingConfig:
    model_name: str
    batch_size: int = 32
    learning_rate: float = 1e-3
    epochs: int = 10

cfg = TrainingConfig(model_name='resnet18')
print(cfg)                          # auto-generated __repr__
print('Batch size:', cfg.batch_size)

# @dataclass also provides equality for free
cfg2 = TrainingConfig(model_name='resnet18')
print('Same config?', cfg == cfg2)

cfg2.learning_rate = 1e-4
print('Same after LR change?', cfg == cfg2)
```

2.7. Handling errors, the `assert` statement

Error handling works in Python much like it does in, for example, C++ and Java. Documentation: https://docs.python.org/3.6/tutorial/errors.html

The `assert` keyword can be used as a debugging tool. If the statement after the `assert` keyword does not return true, an `AssertionError` exception will be thrown.

2.8. Iterators and generators

One of the tools for lazy evaluation is the iterator. The old Python version 2 has many built-in operations that return one or more lists. This is not ideal in the case when large lists need to be generated, but we only required their first few elements. In Python version 3 most of these built-in functions will return an iterator. The iterator only creates the next element if it is needed. Generators are a special type of iterator. The following link contains some more information about generators:

https://wiki.python.org/moin/Generators

2.9. Homework (Optional)

1. **Typed mean function.** Write a function `weighted_mean(xs: List[float], ws: List[float]) -> float` that computes the weighted mean of `xs` with weights `ws`. Validate with `assert` that the two lists have the same length.
2. **Model-metadata dataclass.** Define a `@dataclass` named `ModelInfo` with fields `name: str`, `num_params: int`, `accuracy: float`, and `epochs_trained: int = 0`. Instantiate it and print the auto-generated repr.
3. **Comprehension for confusion-matrix cells.** Given `true = ['cat', 'dog', 'cat', 'cat']` and `pred = ['cat', 'cat', 'dog', 'cat']`, use a dict comprehension to count occurrences of each (true, pred) pair.
4. **Lambdas for loss selection.** Create a dict that maps loss names to lambdas: `'mse'`, `'mae'`, `'squared_log'`, each taking `(y_hat, y)` and returning a single float. Apply each to a small list of prediction/target pairs.
5. **Custom iterable class.** Write a `class Batches` whose instances can be iterated with a `for` loop to yield successive chunks of a source list. The constructor should take `data` and `batch_size`. Use `__iter__` and `__next__` (or `yield` in a generator method).
6. **Catching specific exceptions.** Write a function that receives a list of strings and tries to convert each to `float`. Collect successful conversions in one list and the failing strings in another, using a `try/except ValueError`.

Chapter 3

NumPy Introduction

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook: https://colab.research.google.com/drive/11lGGnduGU9m1QnyNhbrl1bdFN4wuItSZ?usp=sharing#scrollTo=Ee33Tyx-7aXh

Numpy is one of Python's most important packages: it allows for quick numerical computations. The default interpreter/compiler of Python is CPython. This implementation, compared to other languages, results in largely slower programs. The reasons for this lie in Python a being dynamically typed language, its automatic memory allocation, and CPython working almost entirely as an interpreter, which leaves very few options for optimizing code.

The speed difference truly shows when many Python statements need to be run, for example, while doing a matrix multiplication by using loops and multiplying the scalars together. This is exactly why Numpy contains functions that are capable of performing the same operation vectorized, many times over, without needing to use a Python loop. Numpy functions call code that was written in more efficient languages (such as C, Fortran), so much higher efficiency is possible by using Numpy than by using a native Python implementation.



3.1. Creating arrays

The type of the elements inside the `ndarray` array must be given at its creation. Every element in the array will have the same type, with the exception of rarely used structured arrays and recarrays, but we won't be using those.

Documentation: https://docs.scipy.org/doc/numpy/reference/routines.array-creation.html

```
a = np.zeros((2,3), dtype=np.float32)
print("New floating point array, filled with zeros:\n", a)

b = np.ones((2,2), dtype=np.int32)
print("New integer array, filled with ones:\n", b)

c = np.empty((1,3), dtype=np.float32)  # NO INITIALIZATION of elements, values are undefined
print("New EMPTY (uninitialized) array (elements may have any starting value):\n", c)

d = np.array([[1.,.2,-2.8],[2.6,3.2,-.2]], dtype=np.float32)
print("New floating point array, elements are given explicitly:\n", d)
print("    ... it's shape is ", d.shape)
```

```
New floating point array, filled with zeros:
 [[0. 0. 0.]
 [0. 0. 0.]]
New integer array, filled with ones:
 [[1 1]
 [1 1]]
New EMPTY (uninitialized) array (elements may have any starting value):
 [[ 0.000000e+00  0.000000e+00 -5.630583e+20]]
New floating point array, elements are given explicitly:
 [[ 1.   0.2 -2.8]
 [ 2.6  3.2 -0.2]]
    ... it's shape is  (2, 3)
```

Main datatypes (`dtype`): * `np.bool_`: Boolean datatype (True or False). True is represented by a 1 and False is represented by a 0. * `np.int8, int16, int32, int64`: integer datatypes contained in 1, 2, 4, 8 bytes, I.e. int8's values are inside the [-128, 127] interval. * `np.float16, float32, float64`: real number datatypes contained in 2, 4, 8 bytes. Graphics cards today are optimized for using the float32 (single precision) datatype.

Creating arrays from equal step size sequences:

```
# ranges, interavals

a = np.arange(5, dtype=np.int64)   # end value is given
print("Array with elements from range(5): ", a)
print("    ... shape of the array is:", a.shape)

b = np.arange(5, 10, dtype=np.float16)   # start & end value is given
print("\nArray with elements from range(5, 10): ", b)

c = np.arange(10, -4, -2, dtype=np.float64)   # start, end & step value is given
print("\nArray with elements from range(10, 4, -2): ", c)

b2 = np.arange(10, 5, dtype=np.float16)   # start & end value is given
print("\nArray with elements from (empty) range(10, 5): ", b2)
print("    ... shape of the empty array is:", b2.shape)

d = np.linspace(10., 13., num=7, endpoint=False, dtype=np.float64)
print("\nArray with equally spaced numbers from interval:\n", d)
```

```
Array with elements from range(5):  [0 1 2 3 4]
    ... shape of the array is: (5,)

Array with elements from range(5, 10):  [5. 6. 7. 8. 9.]

Array with elements from range(10, 4, -2):  [10.  8.  6.  4.  2.  0. -2.]

Array with elements from (empty) range(10, 5):  []
    ... shape of the empty array is: (0,)

Array with equally spaced numbers from interval:
 [10.         10.42857143 10.85714286 11.28571429 11.71428571 12.14285714
 12.57142857]
```

Generating random numbers:

```
# random numbers
r1 = np.random.rand(1, 4, 1)
print("(1,4,1) shaped array with random elements from [0,1) interval (uniform):\n ", r1)
print("    ... shape of the array:", r1.shape, "; datatype of the array:", r1.dtype)

r2 = np.random.randn(2)
print("\n(2,) shaped array with random elements from standard normal (Gaussian) distribution:\n ", r2)

r3 = np.random.randint(5, size=(2,5), dtype=np.uint16)
print("\n(2,5) shaped array with random integer elements from [0,4] interval (uniform):\n ", r3)
```

```
(1,4,1) shaped array with random elements from [0,1) interval (uniform):
  [[[0.44428035]
  [0.56448829]
  [0.43426523]
  [0.42845164]]]
    ... shape of the array: (1, 4, 1) ; datatype of the array: float64

(2,) shaped array with random elements from standard normal (Gaussian)
distribution:
  [0.02795003 0.36478077]

(2,5) shaped array with random integer elements from [0,4] interval (uniform):
  [[0 3 1 3 4]
 [2 3 0 3 0]]
```

3.2. Numpy basics - the ndarray type

The most important type in the package, the ndarray is an n-dimensional array implementation. Let's create a 2 dimensional, 2x3 size array containing integers. The dimensions of an array are not to be confused with the mathematical dimensions of a vector.

```
import numpy as np  # import numpy package using the 'np' alias

a = np.array([[2,3,4],[1,5,8]], dtype=np.int32)   # create array
b = np.array([[3,2,9],[2,6,5]], dtype=np.int32)   # create array

print("The 'a' array:\n", a)
print("The 'b' array:\n", b)
print("Sum of the two arrays, elementwise:\n", a + b)
print("Multiplying all elements of 'a' by 2:\n", 2*a)
print("Adding up all elements of array 'a'. The sum is:", np.sum(a))
```

```
The 'a' array:
 [[2 3 4]
 [1 5 8]]
The 'b' array:
 [[3 2 9]
 [2 6 5]]
Sum of the two arrays, elementwise:
 [[ 5  5 13]
 [ 3 11 13]]
Multiplying all elements of 'a' by 2:
 [[ 4  6  8]
 [ 2 10 16]]
Adding up all elements of array 'a'. The sum is: 23
```

As you can see, it's possible to perform the same operation on the entire array (or a slice of it) without needing to write any loops in our code.

The `ndarray` array is represented of course as a sequence of bytes in the memory, so it is stored as a 1-dimensional vector. Once we created such an array, the number of elements cannot change, so if we wish to raise the size, we must create a new array. However, the shape of the array can change: an array with 6 elements can take up a 2x3, 3x2, 6, 6x1, 6x1x1, 1x3x1x1x2x1, etc. shape as well. In this case, the place of the elements in memory remains unchanged. The possibility of seeing the array in 6 dimensions and index it as such is provided by special iterators of the `ndarray` type. For example, a 1x3x1x1x2x1 shaped array can be indexed as: `my_array[0,2,0,0,1,0]`. The array's implementation can resolve this indexing by knowing how many steps must be taken in the memory to take a step along each axis (`ndarray.strides`).

Let's look at the `ndarray` type's attributes:

```
# An ML-style feature matrix: 3 samples, 4 features each.
X = np.array([[0.1, 0.5, -0.2, 1.0],
              [0.4, 0.3,  0.0, 0.8],
              [0.7, 0.1, -0.6, 0.2]], dtype=np.float32)
print('shape:', X.shape)            # (n_samples, n_features)
print('ndim :', X.ndim)
print('dtype:', X.dtype)
print('itemsize (bytes):', X.itemsize)
print('total bytes:', X.nbytes)
print(X)
```

3.3. Changing the shape of arrays

As mentioned before, the shape of arrays can change freely, assuming that in the new shape the number of elements is unchanged. In this case, no copy of the original array gets made, instead, a new view is created over the old array. The type of this view is also `ndarray`, so it's seen as a simple array. It's important to understand that the view references the elements of the original array, so if the elements of the original array change, we will see these altered elements in the view as well. Also, if we change elements of the view, the original array changes too.

```
a = np.arange(6, dtype=np.int32)
print("The 'a' array:\n", a)
print("   ... its shape is", a.shape)   # a tuple

b = a.reshape((2, 3)).copy()   # reshape to 2D shape: (2, 3)
print("\nThe 'a' array reshaped to 2D shape (2, 3):\n", b)
print("   ... its shape is", b.shape)

# We modify an item in the original array
a[0] = 42
print("\nThe modified 'a' array:\n", a)
print("The 2D 'b' view of the 'a' array is also modified:\n", b)

# Now we modify an item in the view array
b[1,1] = 99
print("\nThe content of the 'a' array after 2nd modification:\n", a)
print("The 'b' view after 2nd modification:", b)
```

```
The 'a' array:
 [0 1 2 3 4 5]
   ... its shape is (6,)

The 'a' array reshaped to 2D shape (2, 3):
 [[0 1 2]
 [3 4 5]]
   ... its shape is (2, 3)

The modified 'a' array:
 [42  1  2  3  4  5]
The 2D 'b' view of the 'a' array is also modified:
 [[0 1 2]
 [3 4 5]]

The content of the 'a' array after 2nd modification:
 [42  1  2  3  4  5]
The 'b' view after 2nd modification: [[ 0  1  2]
 [ 3 99  5]]
```

Since the number of elements in the array is fixed, while reshaping it we can set the length of an axis to -1, which calculates the correct length automatically.

```
a = np.arange(6, dtype=np.int32).reshape((3, 2))    # ndarray.reshape()
print("The 'a' array:\n", a)
print("   ... its shape is", a.shape)

b = a.reshape(-1)   # reshape to 1D
print("\nThe 'b' array:\n", b)
print("   ... its shape is", b.shape)

c = b.reshape((1,-1,3,1))   # reshape to 4D: (1,2,3,1)
print("\nThe 'c' array:\n", c)
print("   ... its shape is", c.shape)

d = np.reshape(a, (-1,))   # np.reshape() works similarly to ndarray.reshape()
print("\nThe 'd' array:\n", d)
print("   ... its shape is", d.shape)
```

```
The 'a' array:
 [[0 1]
 [2 3]
 [4 5]]
   ... its shape is (3, 2)

The 'b' array:
 [0 1 2 3 4 5]
   ... its shape is (6,)

The 'c' array:
 [[[[0]
   [1]
   [2]]

  [[3]
   [4]
   [5]]]]
   ... its shape is (1, 2, 3, 1)

The 'd' array:
```

```
[0 1 2 3 4 5]
  ... its shape is (6,)
```

The order of the array's axes can be switched up as well (`np.swapaxes()`, `np.transpose()`, `ndarray.T`). These operations are the ones that should be used for transposing matrices as well.

3.4. Conversion between datatypes

Later we will need to be able to convert an array of a certain datatype into another datatype. This will create a new array.

Here's a common use case: images and videos are often stored in uint8 format (a pixel's brightness in each color channel is represented by a value between 0 and 255), but in neural networks, we like to use real numbers, because gradient and hidden representations are not necessarily integers, so we convert them to the float32 format.

During the conversion process, we must be aware of possible data loss that may result from loss of precision, or under/overflow.

```
a = np.array([[2,3,125], [255, 0, 45]], dtype=np.uint8)
print("Data type of the 'a' array:", a.dtype)
print("The 'a' array:\n", a)

b = a.astype(np.float32)  # type conversion
print("\nData type of the 'b' array:", b.dtype)
print("The 'b' array:\n", b)
```

```
Data type of the 'a' array: uint8
The 'a' array:
 [[  2   3 125]
 [255   0  45]]

Data type of the 'b' array: float32
The 'b' array:
 [[  2.   3. 125.]
 [255.   0.  45.]]
```

3.5. Basic indexing, multiple views

Multiple elements of a Numpy array can be indexed at once to perform the same operation on all of them without using any loops.

We differentiate two different methods of indexing: **basic** and **advanced indexing**.

Basic indexing manages elements of the array like using a range on a Python list, but along multiple dimensions at once. For example, if array `a` is a one-dimensional array with a minimum of 10 elements, `a[5:10:2]` references its elements with the index 5,7,9. `a[5:10:2]` itself is of the ndarray type and includes the aforementioned three elements. In **basic indexing**, which is very similar, the created array is only a *view* of the original array, so if the original array's elements are changed, the view's elements will also change accordingly.

We will discuss **advanced indexing** in the next notebook.

Accessing and printing an element:

```
# Indexing a small feature batch (4 samples, 3 features).
batch = np.array([[0.1, 0.3, 0.9],
                  [0.4, 0.2, 0.7],
                  [0.8, 0.1, 0.5],
                  [0.6, 0.4, 0.3]], dtype=np.float32)

# Single row (one sample): integer index on axis 0
print('sample #2:', batch[2])

# Single column (one feature across all samples)
print('feature #1 across samples:', batch[:, 1])

# Scalar element at (row, col)
print('batch[0, 2] =', batch[0, 2])

# Negative indices: -1 is the last sample
print('last sample:', batch[-1])
```

Negative index (indexing from the back):

```
# indexing with negative numbers: -1: last, -2: last but one

print("\nAccessing a single element:", a[0,-1])
print("Accessing a single element:", a[-2,-1])
```

```
Accessing a single element: 42
Accessing a single element: 3
```

Range indexing (the syntax similiar to Python lists):

```
a = np.arange(6, dtype=np.int32)
print("The 'a' array:\n", a)
print("   ... its shape is", a.shape)

print("\nA slice of array 'a':", a[2:4])  # syntax -> start:stop:step_size

print("Another slice of array 'a':", a[2:])   # from #2 till end

print("A third slice of array 'a':", a[:4])   # from start till #4 (exclusive)

print("\nEvery second item in array 'a':", a[::2])

print("Reversed 'a':", a[::-1])

print("Custom slice 'a':", a[4:0:-2])
```

```
The 'a' array:
 [0 1 2 3 4 5]
   ... its shape is (6,)

A slice of array 'a': [2 3]
Another slice of array 'a': [2 3 4 5]
A third slice of array 'a': [0 1 2 3]

Every second item in array 'a': [0 2 4]
Reversed 'a': [5 4 3 2 1 0]
Custom slice 'a': [4 2]
```

Range indexing on multiple axes at the same time:

```
# A small grayscale image patch, 8x8 pixels.
img = np.arange(64, dtype=np.float32).reshape((8, 8))
print('image shape:', img.shape)
print(img)

# Extract the top-left 4x4 quadrant
top_left = img[:4, :4]
print('\ntop-left 4x4 quadrant:')
print(top_left)

# Every second row, full width (downsample vertically)
print('\nevery second row:')
print(img[::2, :])
```

Empty range. It's possible for an array's size to be 0, in this case, its shape along at least one axis is 0.

```
b = a[8:,:]
print("\nAn empty slice of 'a': ", b)  # no error if RANGE is out of bounds
print("   ... its shape is", b.shape)

# b = a[8,:]   # would result in an error, because axis#0 SINGLE index is out of bounds
```

```
An empty slice of 'a':  []
   ... its shape is (0, 6)
```

Printing slices of arrays:

```
a = np.arange(12, dtype=np.int32).reshape((4,3))
print("The 2D 'a' array:\n", a)
print("   ... its shape is", a.shape)

a[::2,1:3] = 42
print("\nThe modified 2D 'a' array:\n", a)

b = a[:,:2]   # 'b' is a view of array 'a', if we write 'b', we also write 'a'
print("\nThe 'b' array (a view of 'a'):\n", b)
print("   ... its shape is", b.shape)

b[:] = 99   # with [:] (or [:,:], ...) we refer to all elements in 'b'
print("\nThe 'b' array (a view of 'a'):\n", b)
print("\nThe 'a' array:\n", a)
```

```
The 2D 'a' array:
 [[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
   ... its shape is (4, 3)

The modified 2D 'a' array:
 [[ 0 42 42]
 [ 3  4  5]
 [ 6 42 42]
 [ 9 10 11]]

The 'b' array (a view of 'a'):
 [[ 0 42]
 [ 3  4]
```

```
 [ 6 42]
 [ 9 10]]
   ... its shape is (4, 2)

The 'b' array (a view of 'a'):
 [[99 99]
 [99 99]
 [99 99]
 [99 99]]

The 'a' array:
 [[99 99 42]
 [99 99  5]
 [99 99 42]
 [99 99 11]]
```

3.6. Simple arithmetic operations, treating arrays of different shapes (broadcasting)

Arithmetics performed element-to-element (elementwise) on same-size arrays. Adding the elements of two arrays is done as shown below. No need for python loops.

```
# Element-wise ops work on same-shaped arrays, the workhorse of ML.
a = np.array([[1.2, 0.5, -0.3],
              [0.0, 2.1, -1.4]], dtype=np.float32)
b = np.array([[0.3, 0.3,  0.3],
              [0.1, 0.1,  0.1]], dtype=np.float32)

print('a + b:')
print(a + b)
print('\na * b (element-wise):')
print(a * b)
print('\na ** 2:')
print(a ** 2)
```

```
# full_like / zeros_like / ones_like are handy when building companion arrays
# with matching shape and dtype (e.g. a mask or a gradient buffer).
a = np.array([[1.2, 0.5, -0.3],
              [0.0, 2.1, -1.4]], dtype=np.float32)

mask  = np.full_like(a, 1.0)          # all ones, same shape and dtype as a
grad  = np.zeros_like(a)               # zero-initialised gradient
count = np.ones_like(a, dtype=np.int32)  # different dtype, same shape

print('mask shape:', mask.shape, 'dtype:', mask.dtype)
print('grad:')
print(grad)
print('count:')
print(count)
```

So far all is clear, we can add same-size arrays, multiply them, etc. elementwise. However, Numpy makes it possible, under certain conditions, to perform elementwise operations on arrays of different sizes. To do this, Numpy automatically synchronizes the sizes of the arrays, repeating elements along certain axes if needed. This is **broadcasting**.

What exactly happens if we try to raise every element of an array by a constant?

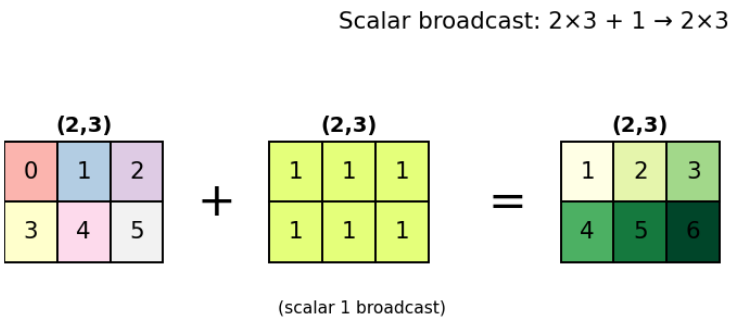
```
a = np.arange(6, dtype=np.int32).reshape((2,3))
print("The 2D 'a' array:\n", a)
print("   ... its shape is", a.shape)

a = a+1
print("\nThe modified 'a' array:\n", a)
```

```
The 2D 'a' array:
 [[0 1 2]
 [3 4 5]]
   ... its shape is (2, 3)

The modified 'a' array:
 [[1 2 3]
 [4 5 6]]
```

Numpy first packs the Python int type '1' literal into a (1,) shaped Numpy array with one element which is the 1. After which, as seen on the image, raises its size to a (2,3) shape by repeating the elements along both axes, so that it exactly matches the 'a' array's shape.



In the case of broadcasting, the **extended size array is really just a view over the original array**, no additional memory space is allocated. There is no actual repeating of elements, instead, the same memory space is accessed repeatedly.

Let's look at other cases of broadcasting. We can think of a 2D array like a matrix. Let's add a vector to each row of this matrix!

```
a = np.arange(6, dtype=np.int32).reshape((2,3))
print("The 2D 'a' array:\n", a)
print("   ... its shape is", a.shape)

vec = np.arange(10,40,10, dtype=np.int32)
print("\nThe 1D 'vec' array:\n", vec)
print("   ... its shape is", vec.shape)

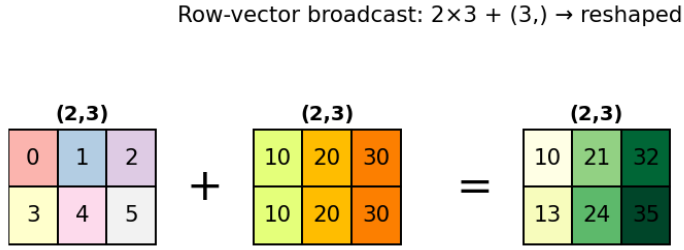
a = a+vec
print("\nThe modified 'a' array:\n", a)
```

```
The 2D 'a' array:
 [[0 1 2]
 [3 4 5]]
   ... its shape is (2, 3)
```

```
The 1D 'vec' array:
 [10 20 30]
   ... its shape is (3,)

The modified 'a' array:
 [[10 21 32]
 [13 24 35]]
```

In this case, the vector with the shape of (3,) gets a new axis: so (1,3) will be its new shape, then by repeating elements along the axis with index #0, it will have a shape of (2,3).



Now let's try adding a column vector to the matrix!

```
a = np.arange(6, dtype=np.int32).reshape((2,3))
print("The 2D 'a' array:\n", a)
print("   ... its shape is", a.shape)

vec = np.arange(10,30,10, dtype=np.int32)
print("\nThe 1D 'vec' array:\n", vec)
print("   ... its shape is", vec.shape)

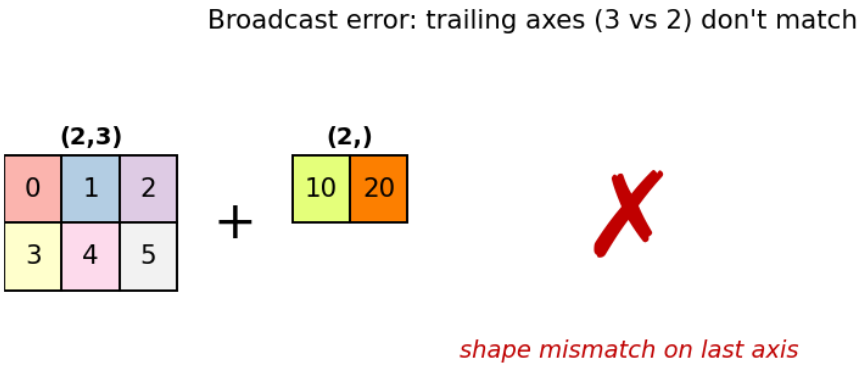
# a = a+vec    # ValueError: operands could not be broadcast together with shapes (2,3) (2,)
```

```
The 2D 'a' array:
 [[0 1 2]
 [3 4 5]]
   ... its shape is (2, 3)

The 1D 'vec' array:
 [10 20]
   ... its shape is (2,)
```

This would cause Numpy to throw an error, as it can't complete the operation. Why?

During broadcasting, Numpy tries comparing the length of each axis of the arrays moving backwards from the last dimension. In the last example, these were equal, however, this time the matrix's last axis with its length of three and the vector's length of 2 will not be compatible with each other.



What can we do?

We need to let Numpy know that we want to compare the vector to the axis with index #0 from the matrix, instead of the axis with index #1. Because the axis comparison begins from the back, the vector must be manually made to be 2 dimensional, by the addition of an extra (length 1) last axis. So, we'll manually reshape the (2,) shaped vector into a (2, 1) shape and after that, perform the addition.

```
a = np.arange(6, dtype=np.int32).reshape((2,3))
print("The 2D 'a' array:\n", a)
print("   ... its shape is", a.shape)

vec = np.arange(10,30,10, dtype=np.int32)
print("\nThe 1D 'vec' array:\n", vec)
print("   ... its shape is", vec.shape)

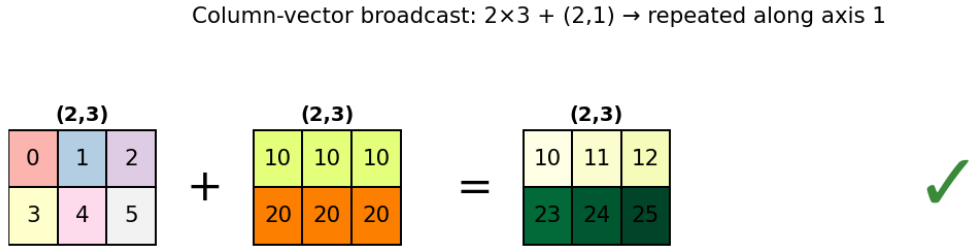
a = a + vec.reshape((2, 1))
print("\nThe modified 'a' array:\n", a)
```

```
The 2D 'a' array:
 [[0 1 2]
 [3 4 5]]
   ... its shape is (2, 3)

The 1D 'vec' array:
 [10 20]
   ... its shape is (2,)

The modified 'a' array:
 [[10 11 12]
 [23 24 25]]
```

The operation was successful, the following happened: the (2,1) shaped vector that we made to be 2 dimensional repeated along axis #1 three times. This is of course just a view because actually, it's only the iteration over the vector that changed, so it still takes up 2 elements of space in memory.



We can create a multiplication table using broadcasting:

```
v1 = np.arange(3, dtype=np.int32)
print("The 'v1' vector:\n", v1)
print("   ... its shape is", v1.shape)

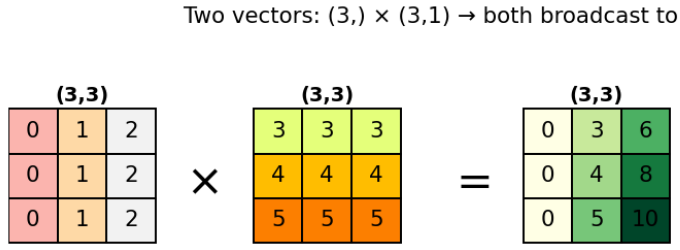
v2 = (np.arange(3, dtype=np.int32)+3).reshape(3,1)
print("The 'v2' vector:\n", v2)
print("   ... its shape is", v2.shape)

r = v1 * v2
print("\nThe result array:\n", r)
```

```
The 'v1' vector:
 [0 1 2]
   ... its shape is (3,)
The 'v2' vector:
 [[3]
 [4]
 [5]]
   ... its shape is (3, 1)

The result array:
 [[ 0  3  6]
 [ 0  4  8]
 [ 0  5 10]]
```

This time the shape of both arrays changed in order to match them together. The first, (3,) shaped vector got a new axis at index #0, so its dimensions match the second vector, making its shape (1,3). Now the first vector's 3 long axis is matched with the other vector's 1 length axis, while the second actor's 3 length axis is matched with the first vector's new axis by Numpy, so along both axes, the length of 1 is extended to a length of 3 by virtually repeating elements. So, in the end, two (3,3) shaped matrices will have their elements multiplied together in pairs, which will return the final result.



So what are the exact rules of broadcasting?

- If the dimensions of two arrays do not match, the array with fewer dimensions gets new (length of 1) first axes. Just enough so that their dimensions match.
- Then the two arrays axes get paired and compared: the first array's axis with index #i has to have a length which is compatible with the second arrays #i axis. The two axes are compatible if their length is equal or either has a length of 1. If their length is equal, there is no need for repetition along the given axis. If their length is not equal and one of the arrays' axis has a length of 1, then along that array the entire array is repeated, enough times that the axis' length matches the length of the other array's axis.
- If two arrays are incompatible along any of the axes, then Numpy throws an error and the broadcasting cannot be performed.

Documentation: https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html

The `None` index

If we want to add a new (length 1) axis to the shape of the array, the `None` (or `np.newaxis`) index is something we can use.

```
# the previous code block in a more compact way

r = np.arange(3, dtype=np.int32) * (np.arange(3, dtype=np.int32)+3)[:,None]
print("\nThe result array:\n", r)
```

```
The result array:
 [[ 0  3  6]
 [ 0  4  8]
 [ 0  5 10]]
```

Some important element-by-element (elementwise) operation

```
a = np.arange(6, dtype=np.int32).reshape((2,3))
print("The 2D 'a' array:\n", a)
print("   ... its shape is", a.shape)

vec = np.array([2,4], dtype=np.int32)[:,None]
print("\nThe 'vec' array:\n", vec)
print("   ... its shape is", vec.shape)

#
print("\nAdding:\n", a+vec)
```

```
print("\nSubtracting:\n", a-vec)
print("\nMultiplying:\n", a*vec)
print("\nFloat division:\n", a/vec)   # this would be integer div in Python2 since both array are of int type!!
   !
print("\nInteger division:\n", a//vec)
print("\nPower:\n", a**vec)
print("\nModulo:\n", a%vec)
print("\nElementwise maximum:\n", np.maximum(a, vec))
print("\nEqual:\n", a == vec)
print("\nGreater:\n", a > vec)
```

```
The 2D 'a' array:
 [[0 1 2]
 [3 4 5]]
   ... its shape is (2, 3)

The 'vec' array:
 [[2]
 [4]]
   ... its shape is (2, 1)

Adding:
 [[2 3 4]
 [7 8 9]]

Subtracting:
 [[-2 -1  0]
 [-1  0  1]]

Multiplying:
 [[ 0  2  4]
[... 19 lines omitted ...]
 [[2 2 2]
 [4 4 5]]

Equal:
 [[False False  True]
 [False  True False]]

Greater:
 [[False False False]
 [False False  True]]
```

Elementwise operations on bool type arrays:

```
a = np.arange(10, dtype=np.int32).reshape((2,5))
b = (a % 3 == 0)    # True for elements divisible by 3
print("The 2D 'b' array:\n", b)
print("   ... its shape is", b.shape)
print("   ... its data type is", b.dtype)  # np.bool_

c = (a % 2 != 0)    # True for elements not divisible by 2 (odd numbers)
for i in range(2):
  for j in range(5):
    c[i,j] = a[i,j] % 2 != 0

print("\nThe 2D 'c' array:\n", c)
print("   ... its shape is", c.shape)
print("   ... its data type is", c.dtype)  # np.bool_
```

```
print("\nElementwise logical AND of the two arrays:\n", b & c)  # 'and' (1.2 and 1) => (True and True)
np.logical_and(b, c)
print("\nElementwise logical OR of the two arrays:\n", b | c)
print("\nElementwise logical NOT of array 'b':\n", ~b)
```

```
The 2D 'b' array:
 [[ True False False  True False]
 [False  True False False  True]]
   ... its shape is (2, 5)
   ... its data type is bool

The 2D 'c' array:
 [[False  True False  True False]
 [ True False  True False  True]]
   ... its shape is (2, 5)
   ... its data type is bool

Elementwise logical AND of the two arrays:
 [[False False False  True False]
 [False False False False  True]]

Elementwise logical OR of the two arrays:
 [[ True  True False  True False]
 [ True  True  True False  True]]

Elementwise logical NOT of array 'b':
 [[False  True  True False  True]
 [ True False  True  True False]]
```

3.7. Memory layout, views vs. copies

Slicing and reshaping in NumPy almost never copy, they return a new `ndarray` that *views* the same memory as the original. This is what makes array operations fast, but it is also the single biggest source of surprises for new users: modifying a slice modifies the underlying array too. Understanding when you have a view and when you have a copy is essential before doing anything in place.

Two tools settle the question quickly: `arr.base` tells you whether an array is a view (it is, if `base` is not `None`), and `np.shares_memory(a, b)` tells you whether two arrays are aliased. The full picture also involves *strides*, the per-axis byte offset used to walk the memory, but for now it is enough to know how to check and how to force a copy with `.copy()`.

```
a = np.arange(12, dtype=np.int32).reshape((3, 4))
print('original a:')
print(a)

# A slice is a VIEW of the same memory.
row0 = a[0]
print('\nrow0.base is a:', row0.base is a)        # True, it is a view
print('shares memory?', np.shares_memory(a, row0))  # True

# Writing through the view modifies the original.
row0[0] = 999
print('\na after row0[0] = 999:')
print(a)
```

```python
# Force a copy to break the aliasing.
a = np.arange(6, dtype=np.int32).reshape((2, 3))
b = a[:, :2].copy()            # .copy() guarantees independent memory
print('shares memory?', np.shares_memory(a, b))   # False

b[0, 0] = -1
print('a (unchanged):')
print(a)
print('b (modified):')
print(b)

# A rule of thumb: if you will mutate the derived array and want the
# original left alone, call .copy() explicitly. The cost is a new
# allocation; the benefit is predictability.
```

3.8. Homework (Optional)

1. **View or copy?** Given `a = np.arange(24).reshape(4, 6)`, determine whether each of the following is a view or a copy, and verify with `np.shares_memory`: `a[::2]`, `a[1:3, 1:4]`, `a.ravel()`, `a.flatten()`, `a.T`, `a + 0`.
2. **Image-batch shapes.** Create a random float32 tensor of shape `(8, 3, 32, 32)` representing a mini-batch of RGB images. Reshape it to `(8, 3072)` per-sample feature vectors. Verify the total number of elements is preserved and that the first sample's first-channel first-pixel matches across both views.
3. **Broadcasting prediction.** Predict the output shape of `a * b` for each pair before running numpy: `(3, 1) * (1, 4)`, `(5, 2, 3) * (2, 3)`, `(5,) * (5, 1)`, `(4, 1) * (3,)`. Check your answers empirically.
4. **Strided sliding window.** Using `np.lib.stride_tricks.sliding_window_view`, create a view over `x = np.arange(10)` that yields all length-3 contiguous windows. Print the result. Then replace with `.copy()` to detach and mutate safely.
5. **Type conversion without data loss.** Given `a = np.array([255, 128, 1], dtype=np.uint8)`, convert to `float32`, divide by 255, and verify the result is in [0, 1]. Then attempt to convert back to `uint8`, explain what happens if the float values go outside the uint8 range.

Chapter 4

NumPy Fundamentals

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook: https://colab.research.google.com/drive/1NZuk-VUDWbsWejKSHYVR1VE6k86UIZBv?usp=sharing#scrollTo=QyEuD6e91QM_

4.1. Operations performed along certain axes

These operations create new values from slices taken from one or more axis of an array. They all have an `axis` parameter, with which we can give along what axis the operation should be performed.

For example, calling the `np.sum()` operation on a (2,3) sized array with the `axis=0` parameter will sum the elements along the axis with index #0, while calling it with the `axis=1` parameter will do the same along axis #1. In the former case, the resulting array will become (3,) shaped, while in the latter case it will become (2,) shaped. Multiple axes can be given with the `axis` parameter (by listing them in a tuple), in which case those axes will not appear in the result, as summation happens along all the given axes. If we call the `np.sum()` function on the array without the `axis` parameter, it will sum the entire array, resulting in a scalar (this means the function was called with the default `axis=None` argument)




```
import numpy as np

a = np.arange(6, dtype=np.int32).reshape((2,3))
print("The 2D 'a' array:\n", a)
print("   ... its shape is", a.shape)

print("\nSumming array along axis#0: ", np.sum(a, axis=0))
print("Summing array along axis#1: ", np.sum(a, axis=1))
print("Summing array along axis#0 and axis#1: ", np.sum(a, axis=(0,1)))
print("Summing whole array: ", np.sum(a))
```

```
The 2D 'a' array:
 [[0 1 2]
 [3 4 5]]
   ... its shape is (2, 3)

Summing array along axis#0:  [3 5 7]
Summing array along axis#1:  [ 3 12]
Summing array along axis#0 and axis#1:  15
Summing whole array:  15
```

Other similiar operations: `np.prod()`, `np.mean()`, `np.std()`, `np.amax()`, `np.amin()`

```
# A small feature batch: 4 samples, 3 features each.
X = np.array([[0.5, 0.1, 1.2],
              [0.3, 0.9, 0.4],
              [1.1, 0.2, 0.8],
              [0.0, 0.7, 0.6]], dtype=np.float32)
print('batch shape:', X.shape)   # (n_samples, n_features)
print(X)
```

Logical operations include: `np.all()`, `np.any()`

```
# Column-wise statistics are bread-and-butter preprocessing ops.
print('per-feature mean:', X.mean(axis=0))
print('per-feature std :', X.std(axis=0))
print('per-sample sum  :', X.sum(axis=1))          # one value per row
print('max value overall:', X.max())               # scalar
```

4.2. Matrix and vector operations, linear algebra

Matrix multiplication:

```
m = np.arange(9, dtype=np.float32).reshape((3,3))   # 3 by 3 matrix
print("The 'm' matrix:\n", m)
print("   ... its shape is", m.shape)

mm = np.matmul(m, m)

print("\nThe matrix product is:\n", mm)
print("   ... its shape is", mm.shape)

mm = np.dot(m, m)    # dot product: when applying on 2D matrices, has the same effect as np.matmul()

print("\nThe matrix product is:\n", mm)
print("   ... its shape is", mm.shape)
```

```
The 'm' matrix:
 [[0. 1. 2.]
 [3. 4. 5.]
 [6. 7. 8.]]
   ... its shape is (3, 3)

The matrix product is:
 [[ 15.  18.  21.]
 [ 42.  54.  66.]
 [ 69.  90. 111.]]
   ... its shape is (3, 3)

The matrix product is:
 [[ 15.  18.  21.]
 [ 42.  54.  66.]
 [ 69.  90. 111.]]
   ... its shape is (3, 3)
```

Non-square matrix, $M^T M$

```
m = np.arange(6, dtype=np.float32).reshape((2,3))   # 2 by 3 matrix
print("The 'm' matrix:\n", m)
print("   ... its shape is", m.shape)
```

```
mm = np.matmul(m, m.T)   # transpose: m.T or np.transpose(m)

print("\nThe matrix product is:\n", mm)
print("   ... its shape is", mm.shape)
```

```
The 'm' matrix:
 [[0. 1. 2.]
 [3. 4. 5.]]
   ... its shape is (2, 3)

The matrix product is:
 [[ 5. 14.]
 [14. 50.]]
   ... its shape is (2, 2)
```

Multiplying a vector with a matrix:

```
m = np.arange(6, dtype=np.float32).reshape((2,3))   # 2 by 3 matrix
print("The 'm' matrix:\n", m)
print("   ... its shape is", m.shape)
v1 = np.arange(3, dtype=np.float32)
print("The 'v1' vector:\n", v1)
print("   ... its shape is", v1.shape)

# (2, 3) x (3,) -> (2,)

r1 = np.dot(m, v1)
print("\nThe result vector:\n", r1)
print("   ... its shape is", r1.shape)

# (2,) x (2, 3) -> (3,)

v2 = v1[:2]
print("The 'v2' vector:\n", v2)
print("   ... its shape is", v2.shape)

r2 = np.dot(v2, m)
print("\nThe result vector:\n", r2)
print("   ... its shape is", r2.shape)
```

```
The 'm' matrix:
 [[0. 1. 2.]
 [3. 4. 5.]]
   ... its shape is (2, 3)
The 'v1' vector:
 [0. 1. 2.]
   ... its shape is (3,)

The result vector:
 [ 5. 14.]
   ... its shape is (2,)
The 'v2' vector:
 [0. 1.]
   ... its shape is (2,)

The result vector:
 [3. 4. 5.]
   ... its shape is (3,)
```

Inner product of two vectors:

```
v1 = np.arange(3)
v2 = np.arange(3)
print("\nThe 'v1' vector:", v1)
print("   ... its shape is", v1.shape)
print("The 'v2' vector:", v2)
print("   ... its shape is", v2.shape)

print("\nTheir dot product is: ", np.dot(v1, v2))
```

```
The 'v1' vector: [0 1 2]
   ... its shape is (3,)
The 'v2' vector: [0 1 2]
   ... its shape is (3,)

Their dot product is:  5
```

Other important operations:

```
m = np.array([[2., 1., 3.],[4., 2., 6.],[1., 1., 5.]], dtype=np.float32)
print("The 'm' matrix:\n", m)
print("   ... its shape is", m.shape)

print("\nRank of matrix:", np.linalg.matrix_rank(m)) # not full rank since row#1 == 2*row#0

print("Determinant of matrix:", np.linalg.det(m))  # zero since it is not full rank

v1 = np.arange(3, dtype=np.float32)
print("\nThe 'v1' vector:\n", v1)
print("   ... its shape is", v1.shape)
print("\nLength of vector 'v1' (L2 norm):", np.linalg.norm(v1, ord=2))   # ord=2 is the default (euclidean
   distance)

print("\nLength of row-vectors of 'm' mnatrix:", np.linalg.norm(m, axis=1, ord=2))

# euclidean distance of two points
two_points = np.array([[1.,2.,-2.],[5., 3., 2.]])
print("\nTwo points:\n", two_points)

vec_length = np.linalg.norm(two_points[0,:] - two_points[1,:], ord=2)
print("  ... their distance:", vec_length)
```

```
The 'm' matrix:
 [[2. 1. 3.]
 [4. 2. 6.]
 [1. 1. 5.]]
   ... its shape is (3, 3)

Rank of matrix: 2
Determinant of matrix: 0.0

The 'v1' vector:
 [0. 1. 2.]
   ... its shape is (3,)

Length of vector 'v1' (L2 norm): 2.236068

Length of row-vectors of 'm' mnatrix: [3.7416575 7.483315  5.196152 ]

Two points:
 [[ 1.  2. -2.]
```

```
 [ 5.  3.  2.]]
  ... their distance: 5.744562646538029
```

```
a=np.array([[1,2,3], [2.4, 5, 3]])
a #( two vectors, 3 items long each)
# sum of squares along each rows
np.sqrt(np.sum(a**2,axis=1))
```

```
array([3.74165739, 6.30555311])
```

4.3. Broadcasting in practice

Broadcasting is numpy's rule for operating on arrays of different, but compatible, shapes. The official definition is worth reading once but rarely consulted after. In day-to-day ML code you will use three patterns constantly:

1. *Row broadcasting:* add or multiply a per-feature vector against every row of a matrix. Shapes `(N, D) * (D,) -> (N, D)`.
2. *Column broadcasting:* apply a per-sample scalar to every feature of that sample. Shapes `(N, D) * (N, 1) -> (N, D)` (note the explicit trailing 1).
3. *Outer product:* multiply a column vector by a row vector. Shapes `(N, 1) * (1, D) -> (N, D)`.

The rules: align shapes on the right. Each dimension must either match or equal 1 (a length-1 dimension is stretched virtually). Common pitfall: a `(N,)` vector broadcasts as a row. To broadcast as a column you need `(N, 1)`.

```
# Row broadcasting: subtract the per-feature mean from every row (centering).
X = np.array([[0.5, 0.1, 1.2],
              [0.3, 0.9, 0.4],
              [1.1, 0.2, 0.8]], dtype=np.float32)
mean_per_feature = X.mean(axis=0)        # shape (3,)
X_centered = X - mean_per_feature        # (3,3) - (3,) -> (3,3)
print('column means after centering (should be ~0):')
print(X_centered.mean(axis=0))
```

```
# Column broadcasting: normalise each sample to unit L2 norm.
row_norms = np.linalg.norm(X, axis=1, keepdims=True)   # shape (3, 1), NOT (3,)
X_unit = X / row_norms                                 # (3,3) / (3,1) -> (3,3)
print('per-sample norms after normalisation (should be ~1):')
print(np.linalg.norm(X_unit, axis=1))

# Pitfall: without keepdims=True, row_norms would be shape (3,), which
# broadcasts as ROWS, not columns, and the result would be wrong.
bad = X / np.linalg.norm(X, axis=1)     # implicit (3,3) / (3,)
print('\nwrong result without keepdims:')
print(bad)
```

4.4. Concatenating arrays, inserting, deleting.

It's important to know that if a Numpy array is allocated in memory, its size cannot be changed. If we'd like to append or delete an element/row/column/etc., perhaps concatenate multiple arrays, Numpy must always create a new array, which can be a costly operation.

Concatenating:

`np.concatenate()`: Multiple arrays are concatenated along an already **existing** layer. It's important that all the axes, except the one used for concatenation, must be of equal length.

`np.stack()`: Concatenates multiple arrays along a **new** axis. All arrays must have the same shape.

```
# CONCATENATE

a = np.arange(6, dtype=np.float32).reshape((2,3))
print("The 2D 'a' array:\n", a)
print("   ... its shape is", a.shape)

b = np.zeros((2,4), dtype=np.float32)
print("\nThe 2D 'b' array:\n", b)
print("   ... its shape is", b.shape)

c = np.concatenate([a, b], axis=-1)
print("\nThe two arrays concatenated along their last axis:\n", c)
print("   ... the concatenated array shape is", c.shape)

# STACK: all arrays must have the same shape

d = b[:,:3]
print("\nThe 2D 'd' array:\n", d)
print("   ... its shape is", d.shape)

e = np.stack([a, d], axis=0)
print("\nArrays 'a' and 'd' stacked along a new #0 axis:\n", e)
print("   ... the new shape is", e.shape)

f = np.stack([a, d], axis=-1)
print("\nArrays 'a' and 'd' stacked along a new last (#2) axis:\n", f)
print("   ... the new shape is", f.shape)
```

```
The 2D 'a' array:
 [[0. 1. 2.]
 [3. 4. 5.]]
   ... its shape is (2, 3)

The 2D 'b' array:
 [[0. 0. 0. 0.]
 [0. 0. 0. 0.]]
   ... its shape is (2, 4)

The two arrays concatenated along their last axis:
 [[0. 1. 2. 0. 0. 0. 0.]
 [3. 4. 5. 0. 0. 0. 0.]]
   ... the concatenated array shape is (2, 7)

The 2D 'd' array:
 [[0. 0. 0.]
 [0. 0. 0.]]
   ... its shape is (2, 3)

Arrays 'a' and 'd' stacked along a new #0 axis:
 [[[0. 1. 2.]
  [3. 4. 5.]]

 [[0. 0. 0.]
```

```
  [0. 0. 0.]]]
   ... the new shape is (2, 2, 3)

Arrays 'a' and 'd' stacked along a new last (#2) axis:
 [[[0. 0.]
  [1. 0.]
  [2. 0.]]

 [[3. 0.]
  [4. 0.]
  [5. 0.]]]
   ... the new shape is (2, 3, 2)
```

Inserting and deleting elements:

If we'd like to use a loop to insert multiple elements/slices into an array, then it may be much more efficient if we used a list that contains the elements/slices we want to insert, then finally create an array with the list, using, for example, concatenation.

For inserting a single element/slice we can use the `np.append()` and `np.insert()` operations.

With the `np.pad()` operation, we can pad new elements to the edges of an array, even to multiple axes.

Deleting elements is usually done with correct indexing instead of `np.delete()`.

4.5. Advanced indexing

In **Advanced indexing**, similarly to **basic indexing**, we reference multiple elements/slices of an array. However, while in **basic indexing** we can only reference elements in the index-intervals (range) by using one certain step size, in **advanced indexing** there are two other methods available to us.

On one hand, we can use sequences (lists, iterators, integer arrays) for indexing.

Arrays created using **advanced indexing** will always be **copies** of the original array, not views.

```
# Pick specific samples by integer index, useful for shuffling or mini-batching.
idx = np.array([3, 0, 2])
selected = X[idx]        # shape (3, 3), three samples in the specified order
print('selected rows:')
print(selected)
```

Overwriting elements with advanced indexing:

```
a = np.arange(6, dtype=np.int32)+10
print("The 'a' array:", a)
print("   ... its shape is", a.shape)

a[[1,5,2]] = 99
print("\nThe modified 'a' array:", a)
```

```
The 'a' array: [10 11 12 13 14 15]
   ... its shape is (6,)

The modified 'a' array: [10 99 99 13 14 99]
```

We can use an integer type array for indexing.

```
a = np.arange(6, dtype=np.int32)+10
print("The 'a' array:", a)
print("   ... its shape is", a.shape)

idxs = np.array([0,2,0,-1,-4], dtype=np.int32)
print("\nThe 'a' array indexed with 'idxs' array:", a[idxs])
```

```
The 'a' array: [10 11 12 13 14 15]
   ... its shape is (6,)

The 'a' array indexed with 'idxs' array: [10 12 10 15 12]
```

If we index a 1-dimensional array with a multidimensional index array, the result will also be multidimensional.

```
idxs = np.array([[2,3],[5,2],[1,1]], dtype=np.int32)
print("The 'idxs' array:", idxs)
print("   ... its shape is", idxs.shape)

b = a[idxs]
print("\nThe 'a' array indexed with the 2D 'idxs' array:", b)
print("   ... its shape is", b.shape)
```

```
The 'idxs' array: [[2 3]
 [5 2]
 [1 1]]
   ... its shape is (3, 2)

The 'a' array indexed with the 2D 'idxs' array: [[12 13]
 [15 12]
 [11 11]]
   ... its shape is (3, 2)
```

If we index a multidimensional array, along each axis we can use different indexing techniques. If we create a new array using indexing and at least along one axis we use advanced indexing, then the new array will be a copy of our specific part of the original array, not a view of it.

If we'd like to use a sequence to index along multiple axes, then the length of our two sequences must be equal. (If we use a multidimensional array for indexing along multiple axes, then the two index arrays must be broadcastable to the same shape).

Full documentation at: https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html

If we'd like to select a list of elements from a 2 dimensional array, then along two axes we need one equal length sequence each to separately index the #0 and a #1 axis. For example if we'd like to return the `a[0,1], a[2,2], a[0,5]` elements, then we need to index as follows: `a[[0,2,0], [1,2,5]]`.

```
a = np.arange(6, dtype=np.int32).reshape((2,3))+10
print("The 'a' array:\n", a)
print("   ... its shape is", a.shape)

idxs0 = [0,1,0]
idxs1 = [1,2,2]

b = a[idxs0, idxs1]  # extracting a[0,1], a[1,2], a[0,2] into the 'b' array

print("\nThe 'b' array:", b)
print("   ... its shape is", b.shape)
```

```
# we can also use a tuple of lists/arrays/... to do multi-dimensional indexing
# in fact, this is the same thing as above, due to automatic tuple-packing
idxs = (idxs0, idxs1)
b = a[idxs]
print("\nThe 'b' array:", b)
print("   ... its shape is", b.shape)

# advanced indexing only along a single axis: extracting slices from 'a'
c = a[[0,1,0],:]
print("\nThe 'c' array:\n", c)
print("   ... its shape is", c.shape)

# advanced indexing along one axis, basic indexing along the other
d = a[[0,1,0],::2]
print("\nThe 'd' array:\n", d)
print("   ... its shape is", d.shape)

# e = a[[0,1,0],[0,2]]  # but we can't do this:
#   IndexError: shape mismatch: indexing arrays could not be broadcast together with shapes (3,) (2,)

# but we can do this: reshape #1 axis index array to (2,1) to be broadcastable to shape (2,3)
#     and match #0 axis index array shape (3,)
# indexed elements this way: [[a[0,0], a[1,0], a[0,0]], [a[0,2], a[1,2], a[0,2]]]
f = a[np.array([0,1,0]), np.array([0,2])[:,None]]
print("\nThe 'f' array:\n", f)
print("   ... its shape is", f.shape)

# NOTE
# There is an inconsistency how two basic indoces work vs how two array indexing works.
# If you have two slice indexing, the direct product of their indices are selected
# In contrast, two arrays form a list of pairs, describing the list of integer indices
a = np.arange(9, dtype=np.int32).reshape((3,3))
print("\nThe 'a' array:\n", a)

d1 = a[0::2, 0::2]
print("The 'd1' array:\n", d1)
d2 = a[[0,2], [0,2]]
print("The 'd1' array:\n", d2)
```

```
The 'a' array:
 [[10 11 12]
 [13 14 15]]
   ... its shape is (2, 3)

The 'b' array: [11 15 12]
   ... its shape is (3,)

The 'b' array: [11 15 12]
   ... its shape is (3,)

The 'c' array:
 [[10 11 12]
 [13 14 15]
 [10 11 12]]
   ... its shape is (3, 3)

The 'd' array:
 [[10 12]
 [13 15]
```

```
 [10 12]]
   ... its shape is (3, 2)

The 'f' array:
 [[10 13 10]
 [12 15 12]]
   ... its shape is (2, 3)

The 'a' array:
 [[0 1 2]
 [3 4 5]
 [6 7 8]]
The 'd1' array:
 [[0 2]
 [6 8]]
The 'd1' array:
 [0 8]
```

Instead of using multidimensional indexing on a multidimensional array, the array we want to index can be flattened to a single dimension (`ndarray.reshape(-1)`), then the multidimensional indices can be converted into 1-dimensional indices (`np.ravel_multi_index()`), this way the indexing can now be done on the flattened array. This technique can be useful if we want to perform an operation on our multidimensional array that is implemented only for one-dimensional arrays. The array's original shape can be regained after this with `ndarray.reshape()`. The indexes referencing elements of the flattened array can be converted back to multidimensional indexes (`np.unravel_index()`).

Another method of using **advanced indexing** is **indexing with a boolean mask**. Maskin can also happen along one or more axes. The point is that it must be possible to broadcast the mask's size to become compatible with the array's shape along the desired axes.

```
a = np.arange(6, dtype=np.int32).reshape((2,3))
print("The 'a' array:\n", a)
print("   ... its shape is", a.shape)

mask = a < 4
print("\nThe 'mask' array:\n", mask)
print("   ... its shape is", mask.shape)
print("   ... its data type is", mask.dtype)

b = a[mask]  # this is a copy, not a view!
print("\nThe 'b' array:\n", b)
print("   ... its shape is", b.shape)
```

```
The 'a' array:
 [[0 1 2]
 [3 4 5]]
   ... its shape is (2, 3)

The 'mask' array:
 [[ True  True  True]
 [ True False False]]
   ... its shape is (2, 3)
   ... its data type is bool

The 'b' array:
 [0 1 2 3]
   ... its shape is (4,)
```

Printing elements selected using a mask:

```
a = np.arange(6, dtype=np.int32).reshape((2,3))
print("The 'a' array:\n", a)
print("   ... its shape is", a.shape)

# Select all even elements larger than 3
a[(a % 2 == 0) | (a > 3)] = 99

print("\nThe modified 'a' array:\n", a)
```

```
The 'a' array:
 [[0 1 2]
 [3 4 5]]
   ... its shape is (2, 3)

The modified 'a' array:
 [[99  1 99]
 [ 3 99 99]]
```

It's possible to use a mask only along the certain axes that we want to.

Let's zero out all the rows of array a that have at least one negative number!

```
a = np.array([[1.2,2.5],[1.,-.6],[2.8,1.7],[-1.5,.7]], dtype=np.float32)
print("The 'a' array:\n", a)
print("   ... its shape is", a.shape)

a[np.any(a < 0., axis=1),1] = 0

print("\nThe modified 'a' array:\n", a)
print("   ... its shape is", a.shape)
```

```
The 'a' array:
 [[ 1.2  2.5]
 [ 1.  -0.6]
 [ 2.8  1.7]
 [-1.5  0.7]]
   ... its shape is (4, 2)

The modified 'a' array:
 [[ 1.2  2.5]
 [ 1.   0. ]
 [ 2.8  1.7]
 [-1.5  0. ]]
   ... its shape is (4, 2)
```

```
a   # display value
```

```
array([[ 1.2,  2.5],
       [ 1. ,  0. ],
       [ 2.8,  1.7],
       [-1.5,  0. ]], dtype=float32)
```

```
a<0
```

```
array([[False, False],
       [False, False],
       [False, False],
       [ True, False]])
```

```
np.any(a<0, axis=1)
```

```
array([False, False, False,  True])
```

It's possible to use sequences and masks for indexing at the same time on different axes of an array.

A sequence can be created from a mask using the `np.where()` function.

4.6. Other useful Numpy operations

Sorting, counting, searching:

https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.sort.html

Set operations:

https://docs.scipy.org/doc/numpy-1.13.0/reference/routines.set.html

`np.apply_along_axis()`: This operation performs a function that is defined for 1-dimensional arrays, on a multidimensional array's 1-dimensional slices, then concatenates the results. Although this function may be useful at times, we must know that this technique is about as useful as writing a simple Python loop to keep calling the function on the sliced array, then concatenating the results. So **if we want to perform the inner operation on many small slices, this method is not efficient.**

4.7. Homework (Optional)

1. **Broadcasting shape puzzles.** For each expression, predict the output shape, then verify: `a.shape = (5, 3); b.shape = (3,); a + b`; `a.shape = (5, 3); b.shape = (5, 1); a * b`; `a.shape = (5,); b.shape = (1, 5); a + b`; `a.shape = (2, 1, 3); b.shape = (4, 3); a * b`.
2. **Standardise a feature matrix.** Given `X` of shape `(N, D)`, compute `X_std = (X - mean) / std` where `mean` and `std` are per-feature statistics. Make the code work for any `N` and `D`. Verify that the standardised matrix has per-column mean ≈ 0 and std ≈ 1.
3. **Pairwise distance matrix.** Given `P` of shape `(M, 2)` representing 2-D points, compute the `(M, M)` matrix of pairwise Euclidean distances using broadcasting, without any explicit `for` loop.
4. **Boolean masking.** Given `losses = np.array([0.5, 0.1, 0.8, 0.2, 0.9, 0.3])` and `threshold = 0.4`, use boolean indexing to keep only the losses below the threshold and to replace the remaining values with `threshold`.
5. **`np.einsum` trial.** Rewrite `(X @ W).sum(axis=1)` using `np.einsum`. Confirm the numerical result matches. Einsum syntax is essential reading for anyone building custom attention layers later in the course.

Chapter 5

PyTorch GPU Integration

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/12VTIeP1MCFyaLHWo1zSWxKcCQ76fi7hu?usp=sharing

PyTorch provides seamless GPU integration through its native CUDA interface, allowing developers to effortlessly transfer tensors and models to GPU devices for optimized parallel computation. This integration accelerates the training and inference of complex neural networks.

5.1. Necessary Imports

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import time
from IPython.display import Image as IMG
```



5.2. GPU

Basic GPU monitoring informations that can be useful. `!nvidia-smi` (NVIDIA Management Library) can be used to get a summary table. Unfortunately, it cannot be called directly, so other libraries must be used to monitor the gpu in the program. Currently, it only works stably on Nvidia GPUs. To take advantage of the GPU, Cuda Toolkit "https://developer.nvidia.com/cuda-downloads%22 needs to be installed first.

```
!nvidia-smi
```

```
Tue Aug 13 04:02:36 2024
+---------------------------------------------------------------------------------
--------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version:
12.2     |
|-----------------------------------------+----------------------+----------------
--------+
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile
Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util
Compute M. |
|                                         |                      |
```

```
MIG M. |
|=========================================+======================+==============
========|
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |
0 |
| N/A   43C    P8               9W /  70W |      0MiB / 15360MiB |      0%
Default |
|                                         |                      |
N/A |
+-----------------------------------------+----------------------+--------------
--------+

+---------------------------------------------------------------------------------
--------+
| Processes:
|
|  GPU   GI   CI        PID   Type   Process name                            GPU
Memory |
|        ID   ID
Usage      |
|=================================================================================
========|
|  No running processes found
|
+---------------------------------------------------------------------------------
--------+
```

5.3. PyTorch

At its core PyTorch provides two features: 1. An n-dimensional Tensor, similar to NumPy but switches easily between CPU and GPUs. 2. Automatic differentiation for building and training (neural) networks.

Let's take a look on some common features: \ - `numpy.ndarray` vs `torch.tensor` \ - `numpy.ones` vs `torch.ones` \ - `numpy.zeros` vs `torch.zeros` \ - `numpy.copy()` vs `torch.clone()` \ - `.reshape` vs `.view` (numpy vs torch) \ - `numpy.linalg.inv()` vs `torch.inverse()` \ - `torch.from_numpy()` \ - `torch.where()` \

5.3.1. Numpy vs Torch

Two additional parameters can be specified for the Torch tensor. The device can be 'cpu' or cuda, where the number can be selected. The need for a gradient increases its space requirements but is necessary for training.

```
#install cudatoolkit
my_tensor = torch.tensor(data=[2.5], dtype=torch.float64, device='cuda:0', requires_grad=True)
```

```
features_np = np.random.rand(16, 3).astype(np.float32)   # a small feature matrix
print(f'features_np type: {type(features_np)}')
features_t = torch.from_numpy(features_np)
print(f'features_t type: {type(features_t)}')
```

```
print(features_t.shape)
print(features_t.size(0))   # equivalent to .shape[0]
print(features_t.dim())
```

Difference in execution time - numpy vs pytorch

The comparison should consider the conversion time and the time of data movement to/from the GPU memory. Because of the copy times, when possible it is worth initializing variables directly on the GPU.

- `%%time`-total execution time of the cell,
- `%time` -next line execution time, only for Win64 (Win32 chase=0)
- Compiler time: shows up if>0.1 s
- Wall time: time diff.(kernel start and end)
- user: code execution time (maybe more than wall time in paralell case)
- sys: pre and after processes (memory allocation. . .)
- CPU time is user time + sys time

```
# A workload sized like a dense layer on a moderate batch:
#   batch @ W  where  batch is (N, D_in)  and  W is (D_in, D_out).
N, D_in, D_out = 8192, 4096, 2048
batch_np = np.random.randn(N, D_in).astype(np.float32)
weight_np = np.random.randn(D_in, D_out).astype(np.float32)
```

```
%%time
out_np = batch_np @ weight_np
print(out_np[0, :5])
```

```
# Move once, reuse many times, transfer cost is paid only at first use.
batch_t = torch.from_numpy(batch_np).cuda()
weight_t = torch.from_numpy(weight_np).cuda()
```

```
%%time
out_t = batch_t @ weight_t
torch.cuda.synchronize()           # wait for GPU kernels to finish before timing
print(out_t[0, :5])
```

Reshape

```
# Image-batch tensor: (B, C, H, W) = (2, 3, 4, 4)
img_np = np.arange(2*3*4*4, dtype=np.float32).reshape(2, 3, 4, 4)
img_t = torch.tensor(img_np, device='cuda:0')

print('per-image flatten (B, C*H*W):', img_t.reshape(2, -1).shape)
print('flatten spatial dims only (B, C, H*W):', img_t.view(2, 3, 16).shape)
print('swap to channels-last (B, H, W, C):', img_t.permute(0, 2, 3, 1).shape)
```

Matrix inverse, determinant and transpose

Try different types and devices and check accuracy and speed.

```
n = np.array([[2.,3.],[5.,5.]], dtype=np.float32)
p = torch.tensor([[2.,3.],[5.,5.]], dtype=torch.float32, device='cuda:0')

print(f'numpy inverse:\n {np.linalg.inv(n)}')
print(f'torch inverse:\n {p.inverse()}')
```

```
print()

print(f'numpy determinant:\n {np.linalg.det(n)}')
print(f'torch determinant:\n {torch.det(p)}')
print()

print(f'numpy transpose:\n {n.T}')
print(f'torch transpose:\n {p.t()}')
print()
```

```
numpy inverse:
 [[-1.   0.6]
 [ 1.  -0.4]]
torch inverse:
 tensor([[-1.0000,  0.6000],
        [ 1.0000, -0.4000]], device='cuda:0')

numpy determinant:
 -5.0
torch determinant:
 -4.999999523162842

numpy transpose:
 [[2. 5.]
 [3. 5.]]
torch transpose:
 tensor([[2., 5.],
        [3., 5.]], device='cuda:0')
```

Parallel calculations

GPU calculations do not take CPU time.

A simple parallel task where values smaller than 0 are replaced by pi

```
x_torch = torch.randn(10000, 10000, dtype=torch.double, device='cuda:0')
x_numpy = np.random.randn(10000,10000)
# directly on GPU
S2 = torch.Size((100, 100))
x = torch.cuda.HalfTensor(S2)
torch.randn(S2, out=x)
```

```
<ipython-input-12-3db1e3d59094>:5: UserWarning: The torch.cuda.*DtypeTensor
constructors are no longer recommended. It's best to use methods such as
torch.tensor(data, dtype=*, dev ... [99 chars truncated]
  x = torch.cuda.HalfTensor(S2)
```

```
tensor([[ 1.1543, -0.0870, -0.6357,  ...,  0.7729, -1.8184,  1.0146],
        [-0.0368,  1.7432, -1.7773,  ..., -0.7139, -0.6562,  0.3630],
        [ 0.0455, -0.8540,  0.1711,  ..., -0.2925, -1.8477,  0.0518],
        ...,
        [ 1.6680,  0.2681, -0.1237,  ..., -1.0850, -1.3447,  0.4875],
        [ 1.4766, -0.9688, -0.0275,  ...,  0.3203,  0.1554, -0.1052],
        [-0.3711, -1.0449, -0.2446,  ...,  0.4783, -0.4148,  0.4751]],
       device='cuda:0', dtype=torch.float16)
```

```
%%time
tstart = time.time()
x_torch=torch.where(x_torch > 0, x_torch, torch.pi)
thalf = time.time()
x_numpy=np.where(x_numpy > 0, x_numpy, np.pi)
tend = time.time()
print('GPU calculation time pytorch: ',thalf-tstart, ' s')
print('CPU calculation time numpy: ',tend-thalf, ' s')
```

```
GPU calculation time pytorch:  0.050916194915771484  s
CPU calculation time numpy:  0.7665483951568604  s
CPU times: user 619 ms, sys: 163 ms, total: 783 ms
Wall time: 820 ms
```

Memory management on GPU

GPU memory is a scarce resource: a single RTX 3090 has 24 GB, a T4 in Colab has 16 GB, and typical deep models eat tens of gigabytes. Unlike CPU RAM, running out of GPU memory stops the program with a `RuntimeError: CUDA out of memory`. Understanding where memory goes is therefore essential.

PyTorch caches freed GPU memory internally so that repeated allocations are fast. This cache shows up in `nvidia-smi` as "used" memory even after Python objects are deleted, do not panic when that happens. `torch.cuda.empty_cache()` releases the cache back to the driver.

```
# Moving tensors between devices
cpu_tensor = torch.randn(1024, 1024)
gpu_tensor = cpu_tensor.to('cuda:0')       # preferred, device-agnostic
gpu_tensor_alt = cpu_tensor.cuda()         # shorthand, CUDA-only
back_on_cpu = gpu_tensor.cpu()

print('cpu_tensor.device:', cpu_tensor.device)
print('gpu_tensor.device:', gpu_tensor.device)
print('back_on_cpu.device:', back_on_cpu.device)
```

```
def mib(n_bytes):
    return n_bytes / 1024**2

print(f'before allocation:  {mib(torch.cuda.memory_allocated()):.1f} MiB allocated, '
      f'{mib(torch.cuda.memory_reserved()):.1f} MiB reserved')

big = torch.randn(4096, 4096, device='cuda:0')   # 64 MiB at float32
print(f'after allocation:   {mib(torch.cuda.memory_allocated()):.1f} MiB allocated, '
      f'{mib(torch.cuda.memory_reserved()):.1f} MiB reserved')

del big
torch.cuda.empty_cache()
print(f'after empty_cache:  {mib(torch.cuda.memory_allocated()):.1f} MiB allocated, '
      f'{mib(torch.cuda.memory_reserved()):.1f} MiB reserved')
```

Two rules of thumb that save hours of debugging:

1. `torch.cuda.memory_allocated()` reflects what PyTorch currently holds; `memory_reserved()` includes the cached pool. The gap between the two is normal.
2. When an OOM hits, the useful diagnostic is `torch.cuda.max_memory_allocated()`, the peak since the last `reset_peak_memory_stats()`.

5.4. Homework (Optional)

1. **Warm-up vs. steady-state timing.** Transfer a pair of large tensors to GPU and run a matrix multiplication *three times in a row*, timing each run. Explain the gap between the first and the subsequent runs. Use `torch.cuda.synchronize()` before recording elapsed time and justify why it is necessary.
2. **Peak memory of a forward pass.** Create an `(N, 3, 224, 224)` image batch on GPU for `N = 1, 4, 16, 64`. For each `N`, pass it through `torch.nn.Conv2d(3, 64, 3, padding=1)` and record `torch.cuda.max_memory_allocated()`. Plot peak memory vs. batch size and estimate the largest `N` that fits in 8 GiB.
3. **Avoiding redundant transfers.** Write two versions of the same pipeline: one that moves a tensor from CPU to GPU inside a loop of 100 iterations, one that moves it once before the loop. Compare total time and discuss why the second version is always preferable when possible.
4. **`to(device)` as a generic pattern.** Refactor the benchmark code in this notebook so that a single variable `device` (equal to `'cuda:0'` if available, else `'cpu'`) controls placement. Verify that the code runs unchanged on a CPU-only machine. This pattern is essential for reproducible examples in the rest of the course.
5. **Pinned memory for faster transfer.** Read the PyTorch documentation for `torch.Tensor.pin_memory()` and `DataLoader(..., pin_memory=True)`. Measure the transfer time of a 1 GiB CPU tensor to GPU with and without pinned memory. Report the speed-up and explain what pinned memory is doing behind the scenes.

Chapter 6

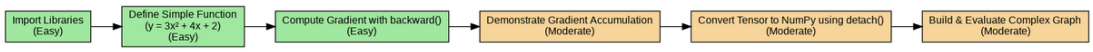
PyTorch Autograd

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/12eGIhIfzor-5FgfLUdwTQdMwbmh3PLXR?usp=sharing

PyTorch Autograd is a powerful automatic differentiation engine that dynamically builds a computation graph as operations are executed, allowing it to automatically compute gradients for tensor operations during backpropagation. This feature simplifies model training and debugging by enabling seamless gradient calculation for complex neural network architectures without the need for manual derivative computations.

6.1. Necessary Imports

```
import numpy as np
import torch
import matplotlib.pyplot as plt
from IPython.display import Image as IMG
```



6.2. Computation graphs and Autograd

$ y = 3x^2 + 4x + 2$

x = 5

$ y = 35^2 + 45 + 2 \quad y = 3*25 + 20 + 2 \quad y = 75 + 20 + 2 \quad y = 97 $

```
#set requires_grad=true if you need the grad later
x = torch.tensor(5., requires_grad=True)
```

```
y = 3*x**2 + 4*x + 2
%whos
```

```
Variable                   Type          Data/Info
-------------------------------------------------
Digraph                    type          <class 'graphviz.graphs.Digraph'>
IMG                        type          <class 'IPython.core.display.Image'>
create_autograd_flowchart  function      <function create_autograd<...>wchart
at 0x78273374a160>
flowchart_output           CapturedIO
np                         module        <module 'numpy' from
'/us<...>kages/numpy/__init__.py'>
```

```
plt                         module        <module
'matplotlib.pyplo<...>es/matplotlib/pyplot.py'>
torch                       module        <module 'torch' from
'/us<...>kages/torch/__init__.py'>
x                           Tensor        tensor(5., requires_grad=True)
y                           Tensor        tensor(97., grad_fn=<AddBackward0>)
```

The `backward()` method is used to compute the gradient during the backward pass in a neural network.

The derivative of `y` with respect to the variable `x`.
$\frac{dy}{dx} = 2 * 3x + 4 = 6x + 4$
`x = 5`

$6x + 4 = 6 * 5 + 4 = 30 + 4 = 34$

```
#y.backward()
x.grad   # display value
```

PyTorch accumulates the derivatives (gradients) on each subsequent backward passes. This is convinient during training **R**ecurrent **N**eural **N**etworks \ To set it zero in each iteration you can use `grad.zero_` function on a `Tensor`.

```
x = torch.tensor(5., requires_grad=True)

for i in range(3):
    y = 3*x**2 + 4*x + 2
    y.backward()
    print(x.grad)
    #x.grad.zero_() ## pytorch accumulates the derivatives by default
```

```
tensor(34.)
tensor(68.)
tensor(102.)
```

to convert a `torch.tensor` to `numpy.ndarray` you have to explicitly remove the computational graph from the tensor using `detach()` command

```
x = torch.tensor(5., requires_grad=False) # torch.tensor can not converted to numpy if grad is True
np_x = x.numpy()
np_x = x.detach().numpy()
print(f'np_x type: {type(np_x)}')
print(f'np_x: {np_x}')
```

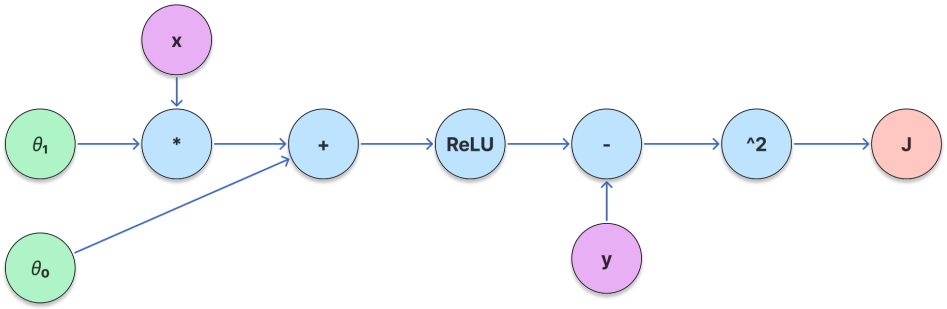
```
np_x type: <class 'numpy.ndarray'>
np_x: 5.0
```

A bit more complex calculation of gradients

6.3. Lecture example

$$J(\theta) = (\hat{y} - y)^2, \quad \hat{y} = ReLU(\theta_0 + x\theta_1), \quad ReLU(x) = \begin{cases} 0, x \leq 0 \\ x, x > 0 \end{cases}$$

6.3.1. 1. Create the directed acyclic graph of the operations



6.3.2. 2. Substitution

$$x = 2,\ y = 1, \qquad \theta_0 = 5,\ \theta_1 = -3, \qquad \alpha = 0.1$$

```
import torch.nn.functional as F
```

```
x = torch.tensor([2.], requires_grad=False)
y = torch.tensor([1.], requires_grad=False)

theta_0 = torch.tensor([5.], requires_grad=True)
theta_1 = torch.tensor([-3.], requires_grad=True)

alpha = 0.1
```

$\ J(\theta) = (ReLU(\theta_0 + \theta_1 x) - y)^2$

```
model = F.relu(theta_0 + theta_1 * x)
J = (model - y)**2
```

```
print(f'model prediction: {model.detach().numpy()}, actual: {y.numpy()}')
print(f'loss: {J.detach().numpy()}')
```

```
model prediction: [0.], actual: [1.]
loss: [1.]
```

```
print(f'theta_0.grad = {theta_0.grad}')
print(f'theta_1.grad = {theta_1.grad}')
```

```
theta_0.grad = None
theta_1.grad = None
```

```
J.backward()
```

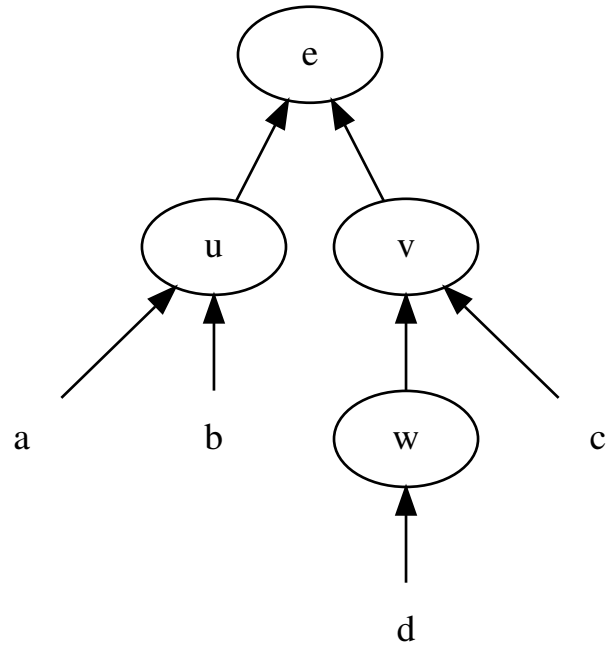
```
print(f'theta_0.grad = {theta_0.grad}')
print(f'theta_1.grad = {theta_1.grad}')
```

```
theta_0.grad = tensor([0.])
theta_1.grad = tensor([0.])
```

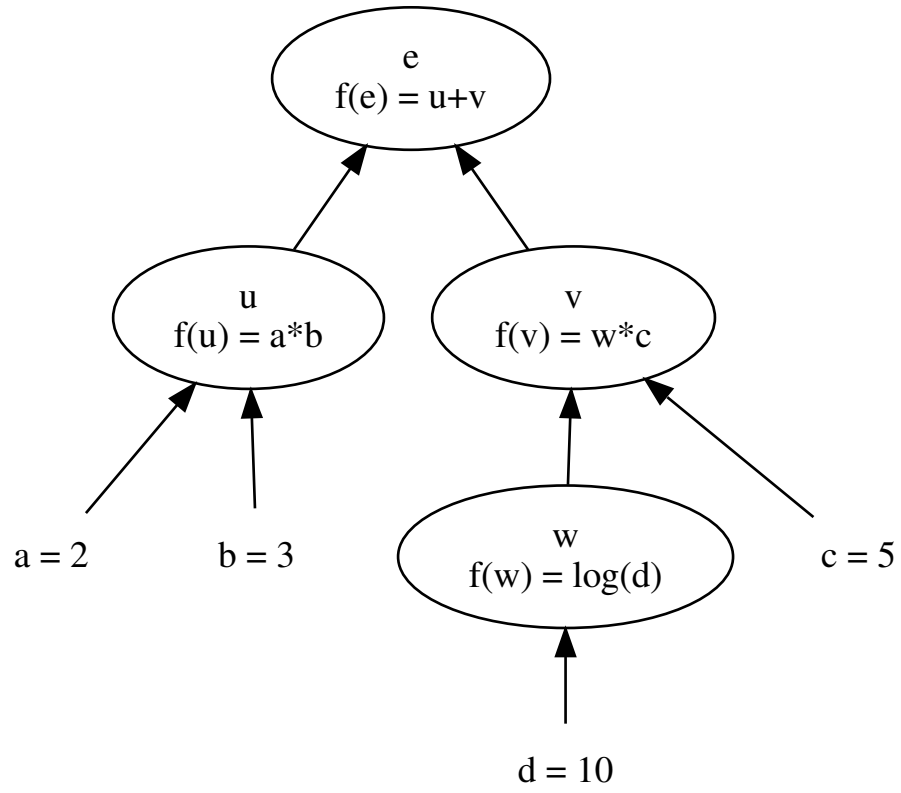
$$e = \underbrace{(a * b)}_{\text{u}} + \underbrace{c * \underbrace{log(d)}_{\text{w}}}_{\text{v}}$$

```
# conda install python-graphviz
import graphviz #graphviz, python-graphviz, (pydot)
g = graphviz.Digraph('G', filename='hello.gv')
g.node('e')
g.node('u')
g.node('v')
g.node('w')
g.node('a', color='transparent')
g.node('b', color='transparent')
g.node('c', color='transparent')
g.node('d', color='transparent')
g.edge('e', 'u', dir='back')
g.edge('e', 'v', dir='back')
g.edge('v', 'w', dir='back')
g.edge('v', 'c', dir='back')
g.edge('u', 'a', dir='back')
g.edge('u', 'b', dir='back')
g.edge('w', 'd', dir='back')
print(g.source)
g
```

```
digraph G {
        e
        u
        v
        w
        a [color=transparent]
        b [color=transparent]
        c [color=transparent]
        d [color=transparent]
        e -> u [dir=back]
        e -> v [dir=back]
        v -> w [dir=back]
        v -> c [dir=back]
        u -> a [dir=back]
        u -> b [dir=back]
        w -> d [dir=back]
}
```




```
g = graphviz.Digraph('G', filename='hello.gv')
g.node('e\nf(e) = u+v')
g.node('u\nf(u) = a*b')
g.node('v\nf(v) = w*c')
g.node('w\nf(w) = log(d)')
g.node('a = 2', color='transparent')
g.node('b = 3', color='transparent')
g.node('c = 5', color='transparent')
g.node('d = 10', color='transparent')
g.edge('e\nf(e) = u+v', 'u\nf(u) = a*b', dir='back')
g.edge('e\nf(e) = u+v', 'v\nf(v) = w*c', dir='back')
g.edge('v\nf(v) = w*c', 'w\nf(w) = log(d)', dir='back')
g.edge('v\nf(v) = w*c', 'c = 5', dir='back')
g.edge('u\nf(u) = a*b', 'a = 2', dir='back')
g.edge('u\nf(u) = a*b', 'b = 3', dir='back')
g.edge('w\nf(w) = log(d)', 'd = 10', dir='back')
g
```



In case of forward-pass (of backpropagation) we have to replace the variables in the equation:

$$e = \underbrace{(a * b)}_{\text{u}} + \underbrace{c * \underbrace{log(d)}_{\text{w}}}_{\text{v}} = (2 * 3) + (5 * \underbrace{log(10)}_{1}) = 11$$

To calculate the backward-pass simply let `torch` do the math and calculate the derivative of **e** with respect to the inputs leading to **e**

$\frac{\partial e}{\partial a} = ? \setminus \frac{\partial e}{\partial b} = ? \setminus \frac{\partial e}{\partial c} = ? \setminus \frac{\partial e}{\partial d} = ?$

```
a = torch.tensor(2., requires_grad=True)
b = torch.tensor(3., requires_grad=True)
c = torch.tensor(5., requires_grad=True)
d = torch.tensor(10., requires_grad=True)

# e = (a * b) + c * torch.log10(d)
u = a*b
w = torch.log10(d)
v = c * w
e = u + v
e

u.retain_grad()
w.retain_grad()
v.retain_grad()

e.retain_grad()
```

```
e.backward()
print(f"a:\n\t is_leaf: {a.is_leaf} \n\t grad_fn: {a.grad_fn} \n\t grad: {a.grad}")
print(f"b:\n\t is_leaf: {b.is_leaf} \n\t grad_fn: {b.grad_fn} \n\t grad: {b.grad}")
print(f"c:\n\t is_leaf: {c.is_leaf} \n\t grad_fn: {c.grad_fn} \n\t grad: {c.grad}")
print(f"d:\n\t is_leaf: {d.is_leaf} \n\t grad_fn: {d.grad_fn} \n\t grad: {d.grad}")
print()
print(f"u:\n\t is_leaf: {u.is_leaf} \n\t grad_fn: {u.grad_fn} \n\t grad: {u.grad}")
print(f"w:\n\t is_leaf: {w.is_leaf} \n\t grad_fn: {w.grad_fn} \n\t grad: {w.grad}")
print(f"v:\n\t is_leaf: {v.is_leaf} \n\t grad_fn: {v.grad_fn} \n\t grad: {v.grad}")
print()
print(f"e:\n\t is_leaf: {e.is_leaf} \n\t grad_fn: {e.grad_fn} \n\t grad: {e.grad}")
```

```
a:
         is_leaf: True
         grad_fn: None
         grad: 3.0
b:
         is_leaf: True
         grad_fn: None
         grad: 2.0
c:
         is_leaf: True
         grad_fn: None
         grad: 1.0
d:
         is_leaf: True
         grad_fn: None
         grad: 0.21714723110198975

u:
         is_leaf: False
         grad_fn: <MulBackward0 object at 0x7826546384c0>
         grad: 1.0
w:
         is_leaf: False
         grad_fn: <Log10Backward0 object at 0x78265463b610>
         grad: 5.0
v:
         is_leaf: False
         grad_fn: <MulBackward0 object at 0x7826546384c0>
         grad: 1.0

e:
         is_leaf: False
         grad_fn: <AddBackward0 object at 0x7826546384c0>
         grad: 1.0
```

The partial derivatives of `u`, `w` and `v` variables:

```
from IPython.display import display, Math
display(Math(fr'\frac{{\partial e}}{{\partial a}} = {a.grad.item()}'))
print()
display(Math(fr'\frac{{\partial e}}{{\partial b}} = {b.grad.item()}'))
print()
display(Math(fr'\frac{{\partial e}}{{\partial c}} = {c.grad.item()}'))
print()
display(Math(fr'\frac{{\partial e}}{{\partial d}} = {d.grad.item()}'))
print()
print()
display(Math(fr'\frac{{\partial e}}{{\partial u}} = {u.grad.item()}'))
```

```
print()
display(Math(fr'\frac{{\partial e}}{{\partial w}} = {w.grad.item()}'))
print()
display(Math(fr'\frac{{\partial e}}{{\partial v}} = {v.grad.item()}'))
```

$$\frac{\partial e}{\partial a} = 3.0$$

$$\frac{\partial e}{\partial b} = 2.0$$

$$\frac{\partial e}{\partial c} = 1.0$$

$$\frac{\partial e}{\partial d} = 0.21714723110198975$$

$$\frac{\partial e}{\partial u} = 1.0$$

$$\frac{\partial e}{\partial w} = 5.0$$

$$\frac{\partial e}{\partial v} = 1.0$$

6.4. Substitution with other values

$$x = 2,\ y = 1, \qquad \theta_0 = 5,\ \theta_1 = -1, \qquad \alpha = 0.1$$

```
x = torch.tensor([2.], requires_grad=False)
y = torch.tensor([1.], requires_grad=False)

theta_0 = torch.tensor([5.], requires_grad=True)
theta_1 = torch.tensor([-1.], requires_grad=True)

alpha = 0.1
```

```
model = F.relu(theta_0 + theta_1 * x)
J = (model - y)**2
```

```
print(f'model prediction: {model.detach().numpy()}, actual: {y.numpy()}')
print(f'loss: {J.detach().numpy()}')
```

```
model prediction: [3.], actual: [1.]
loss: [4.]
```

```
J.backward()
```

```
print(f'theta_0.grad = {theta_0.grad}')
print(f'theta_1.grad = {theta_1.grad}')
```

```
theta_0.grad = tensor([4.])
theta_1.grad = tensor([8.])
```

6.5. Update parameters

$$\theta_0 = \theta_0 - \alpha * \frac{\partial J}{\partial \theta_0}$$

$$\theta_1 = \theta_1 - \alpha * \frac{\partial J}{\partial \theta_1}$$

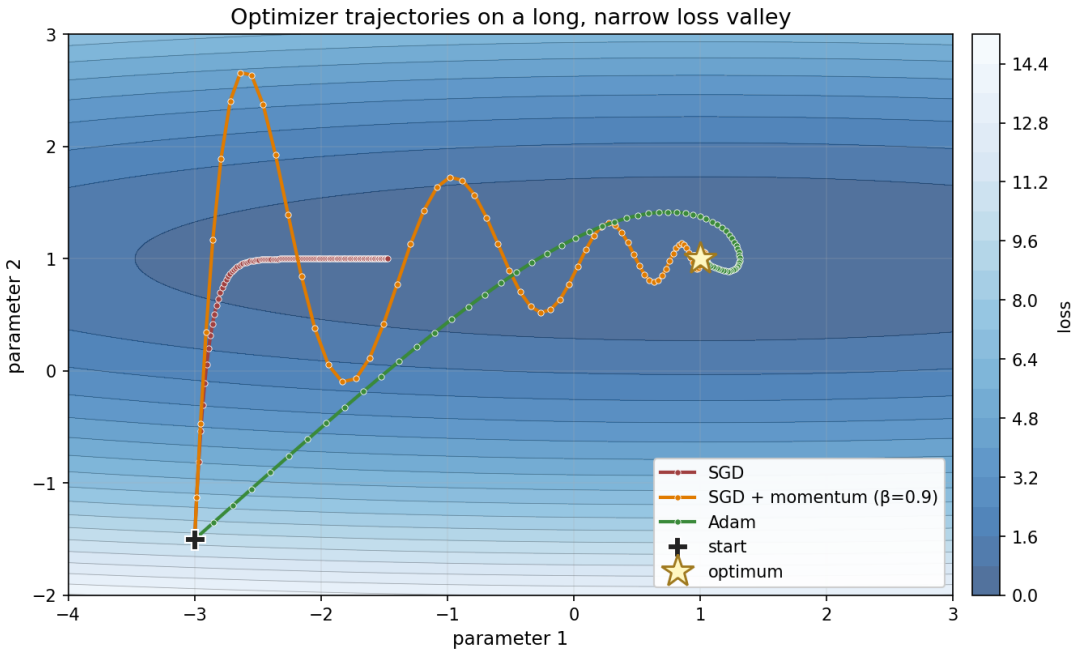
```
theta_0 = theta_0 - alpha * theta_0.grad
theta_1 = theta_1 - alpha * theta_1.grad
```

```
print(f'theta_0: {theta_0.detach().numpy()}')
print(f'theta_1: {theta_1.detach().numpy()}')
```

```
theta_0: [4.6]
theta_1: [-1.8]
```

6.6. How different optimisers traverse the loss landscape

Plain gradient descent (SGD) zig-zags badly when the loss surface has a long, narrow valley. **Momentum** dampens the perpendicular oscillation by averaging recent gradients. **Adam** additionally rescales each parameter by an estimate of its gradient variance, giving a much smoother trajectory.



6.7. Detaching, `no_grad`, and gradient accumulation

Three autograd idioms show up in every training loop. Missing them is the source of subtle bugs where losses explode, evaluation metrics leak into the computation graph, or memory usage creeps up epoch by epoch.

`torch.no_grad()` disables autograd tracking inside a block. Use it for evaluation and inference, the forward pass no longer builds a graph, so memory is much lower and no `.backward()` is possible.

`.detach()` returns a tensor sharing the same storage but removed from the graph. Handy when you need a numeric value for logging (e.g. `loss.detach().item()`) or when you want to stop gradients through a specific tensor.

`grad.zero_()` (or `optimizer.zero_grad()`) resets accumulated gradients at the start of each step. PyTorch accumulates gradients on repeated `.backward()` calls by default, which is convenient for RNNs and gradient-accumulation training, but a silent bug in a normal training loop.

```
# torch.no_grad() for evaluation: no graph, no backward, much less memory.
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)

with torch.no_grad():
    y = (x ** 2).sum()
    print('y inside no_grad:', y)
    print('requires_grad on y?', y.requires_grad)

# Outside the block tracking is on again:
z = (x ** 2).sum()
print('z outside no_grad:', z)
print('requires_grad on z?', z.requires_grad)
```

```
# .detach() to log a value without polluting the graph.
w = torch.tensor(3.0, requires_grad=True)
loss = (w - 1.0) ** 2
print('loss as-is       :', loss)
print('loss.detach()    :', loss.detach())
print('loss.item()      :', loss.item())        # scalar Python float, no graph

# .detach() is also used to freeze a tensor's contribution to the graph.
# For example, if you want to use a running mean as a constant target:
target = w.detach()     # same value as w, but no grad flows into it
```

```
# Gradient accumulation pitfall. Without zero_() the grads add up.
x = torch.tensor(5., requires_grad=True)

print('Three backward passes WITHOUT zeroing, gradients accumulate:')
for step in range(3):
    y = 3 * x**2 + 4 * x + 2
    y.backward()
    print(f'  step {step}: x.grad = {x.grad.item()}')

print('\nAnd now WITH zero_() between passes:')
x = torch.tensor(5., requires_grad=True)
for step in range(3):
    y = 3 * x**2 + 4 * x + 2
    y.backward()
    print(f'  step {step}: x.grad = {x.grad.item()}')
    x.grad.zero_()
```

6.8. Homework (Optional)

1. **Analytical vs. autograd gradient.** Implement $f(x) = 3x^2 + 4x + 2$ with `x = torch.tensor(5., requires_grad=True)`. Use autograd to get $f'(5)$, then compare with the analytical answer $f'(x) = 6x + 4 = 34$.
2. **Two-weight linear model.** Define a mini linear model $\hat{y} = \theta_0 + \theta_1 x$ with θ_0 and θ_1 as learnable tensors. For three `(x, y)` training pairs and a mean-squared-error loss, perform a single backward pass and report `theta_0.grad` and `theta_1.grad`. Verify by hand that the signs make sense.
3. **Accumulation vs. zero_().** Run a 5-step loop computing the same loss each time. First run without `grad.zero_()` and print `x.grad` at each step. Then repeat with zeroing. Explain in one sentence why both behaviours exist.
4. **`no_grad` for evaluation.** Wrap an evaluation pass in `torch.no_grad()` and confirm via `requires_grad` that the output tensor has no graph attached. Measure memory with `torch.cuda.memory_allocated()` (if you have a GPU) inside and outside the block to see the difference.
5. **Leaf vs. non-leaf tensors.** Build a tiny graph: `a = tensor(2., requires_grad=True); b = a * 3; c = b + 1`. Inspect `.is_leaf` on each tensor. Call `b.retain_grad()` and then `c.backward()`, what is `b.grad`? Why is `retain_grad()` necessary for non-leaf tensors?

Chapter 7

Linear Regression

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1ZPeKUjwnp1UeWFo62ZpcTaeI4JrJllKM?usp=sharing

Linear regression is a method for `modeling the relationship between a dependent variable` and one or more independent variables by fitting a straight line to the observed data. It is commonly used to predict outcomes and understand how changes in one variable affect another, making it a simple yet powerful tool for data analysis.

In this example, we continue following Tim and Tom's journey as we estimate their life expectancy [10] using two distinct approaches. The first method employs a closed-form solution based on the Moore-Penrose pseudoinverse, a technique rooted in linear algebra, while the second ustilizes a neural network to capture more complex, non-linear relationships in the data.

7.1. Necessary Imports

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn import metrics
from IPython.display import Image as IMG
```

7.2. Reading the Data

```
!mkdir -p ../data
!wget -q --show-progress -O ../data/LifeExpectancyData.csv "https://www.tamastheactual.com/assets/teaching/
  dnd24251/practice/LifeExpectancyData.csv"

life_exp = pd.read_csv('../data/LifeExpectancyData.csv')
```

```
../data/L    0%[                    ]       0  --.-KB/s
../data/LifeExpecta 100%[===================>] 325.63K  --.-KB/s    in 0.01s
```

7.3. Exploring the Data

```
life_exp.shape   # display value
```

```
(2938, 22)
```

```
life_exp.head() #shows first 5 elements
```

```
Country  Year      Status  Life expectancy   Adult Mortality  \
0  Afghanistan  2015  Developing              65.0            263.0
1  Afghanistan  2014  Developing              59.9            271.0
2  Afghanistan  2013  Developing              59.9            268.0
3  Afghanistan  2012  Developing              59.5            272.0
4  Afghanistan  2011  Developing              59.2            275.0

   infant deaths  Alcohol  percentage expenditure  Hepatitis B  Measles   ...  \
0             62     0.01               71.279624         65.0      1154  ...
1             64     0.01               73.523582         62.0       492  ...
2             66     0.01               73.219243         64.0       430  ...
3             69     0.01               78.184215         67.0      2787  ...
4             71     0.01                7.097109         68.0      3013  ...

   Polio  Total expenditure  Diphtheria    HIV/AIDS         GDP  Population  \
0    6.0               8.16         65.0        0.1  584.259210  33736494.0
1   58.0               8.18         62.0        0.1  612.696514    327582.0
2   62.0               8.13         64.0        0.1  631.744976  31731688.0
3   67.0               8.52         67.0        0.1  669.959000   3696958.0
4   68.0               7.87         68.0        0.1   63.537231   2978599.0

    thinness  1-19 years   thinness 5-9 years  \
0                   17.2                 17.3
1                   17.5                 17.5
2                   17.7                 17.7
3                   17.9                 18.0
4                   18.2                 18.2

   Income composition of resources  Schooling
0                            0.479       10.1
1                            0.476       10.0
2                            0.470        9.9
3                            0.463        9.8
4                            0.454        9.5

[5 rows x 22 columns]
```

```
life_exp.info() #gives information about the fields
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2938 entries, 0 to 2937
Data columns (total 22 columns):
 #   Column                           Non-Null Count  Dtype
---  ------                           --------------  -----
 0   Country                          2938 non-null   object
 1   Year                             2938 non-null   int64
 2   Status                           2938 non-null   object
 3   Life expectancy                  2928 non-null   float64
 4   Adult Mortality                  2928 non-null   float64
 5   infant deaths                    2938 non-null   int64
 6   Alcohol                          2744 non-null   float64
 7   percentage expenditure           2938 non-null   float64
 8   Hepatitis B                      2385 non-null   float64
 9   Measles                          2938 non-null   int64
 10   BMI                             2904 non-null   float64
 11  under-five deaths                2938 non-null   int64
 12  Polio                            2919 non-null   float64
 13  Total expenditure                2712 non-null   float64
```

```
 14  Diphtheria                       2919 non-null   float64
 15   HIV/AIDS                        2938 non-null   float64
 16  GDP                              2490 non-null   float64
 17  Population                       2286 non-null   float64
 18   thinness  1-19 years            2904 non-null   float64
 19   thinness 5-9 years              2904 non-null   float64
 20  Income composition of resources  2771 non-null   float64
 21  Schooling                        2775 non-null   float64
dtypes: float64(16), int64(4), object(2)
memory usage: 505.1+ KB
```

```
life_exp.describe()
```

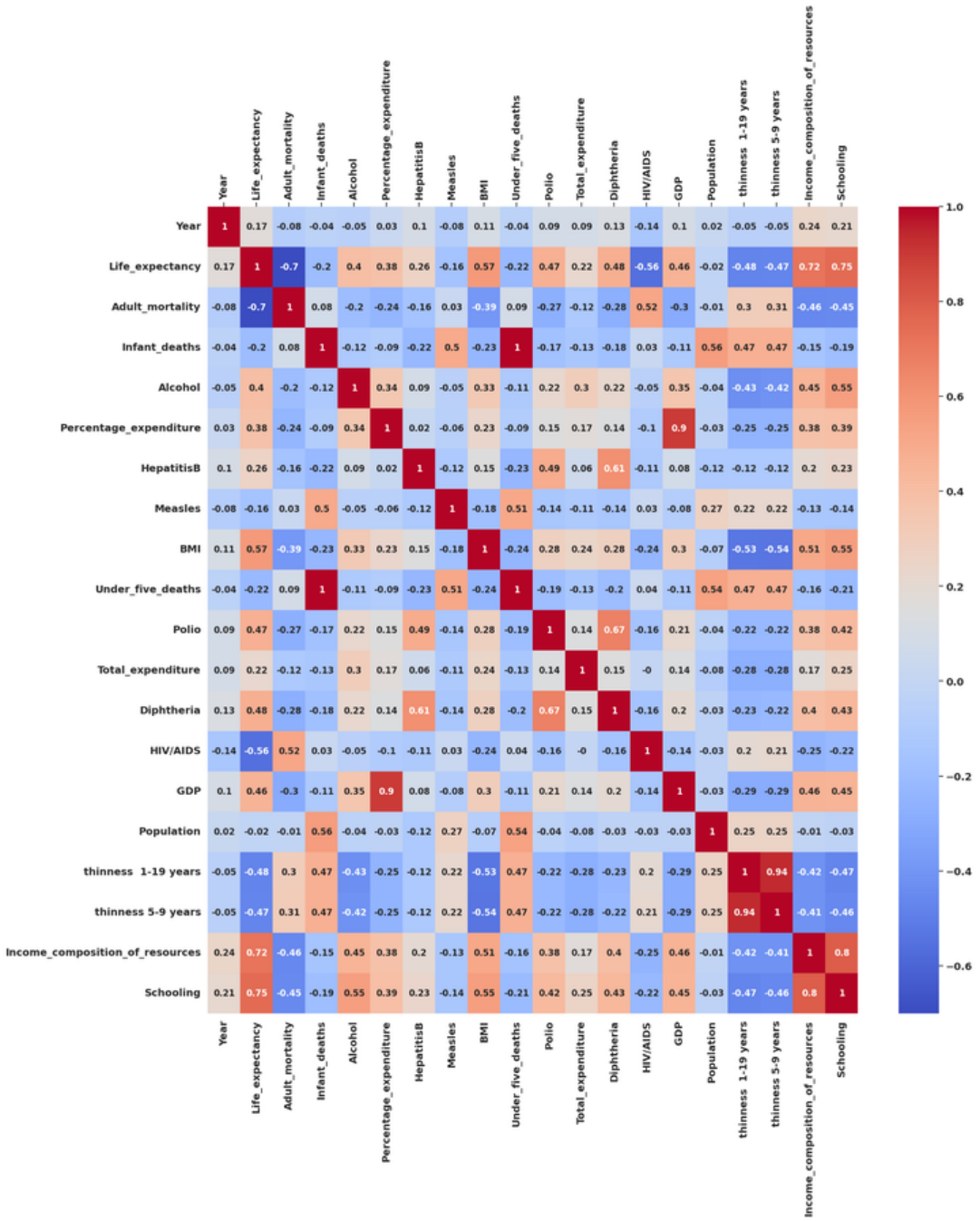
7.4. Data Preprocessing

```
life_exp.drop(columns=['Country', 'Status'], inplace=True)
```

```
#Change column names for conveniency
life_exp.rename(columns = {" BMI " :"BMI",
  "Life expectancy ": "Life_expectancy",
  "Adult Mortality":"Adult_mortality",
  "infant deaths":"Infant_deaths",
  "percentage expenditure":"Percentage_expenditure",
  "Hepatitis B":"HepatitisB",
  "Measles ":"Measles",
  "under-five deaths ": "Under_five_deaths",
  "Total expenditure":"Total_expenditure",
  "Diphtheria ": "Diphtheria",
  " thinness 1-19 years":"Thinness_1-19_years",
  " thinness 5-9 years":"Thinness_5-9_years",
  " HIV/AIDS":"HIV/AIDS",
  "Income composition of resources":
  "Income_composition_of_resources"}, inplace = True)
```

7.5. Correlation Heatmap

```
sns.set(rc = {'figure.figsize':(20,20), 'font.weight': 'bold', 'font.size': 12, 'xtick.labelsize': 14, 'ytick.
   labelsize': 14, 'xtick.top': True, 'xtick.labeltop': True})
ax = sns.heatmap(life_exp.corr().round(2), annot=True, cmap="coolwarm") #visualizing the correlation between
   the variables (how do they influence each other)
plt.savefig('corr.png', bbox_inches='tight')
```



7.6. Handling data

```
X = life_exp
Y = life_exp["Life_expectancy"]
```

```
X.head()
```

```
Y.head()
```

```
0    65.0
1    59.9
2    59.9
3    59.5
4    59.2
Name: Life_expectancy, dtype: float64
```

7.7. Handling missing values

```
X.isnull().sum() #alias for isna -- checks the number of empty, nan, nat, ... elements
```

```
Year                              0
Life_expectancy                  10
Adult_mortality                  10
Infant_deaths                     0
Alcohol                         194
Percentage_expenditure            0
HepatitisB                      553
Measles                           0
BMI                              34
Under_five_deaths                 0
Polio                            19
Total_expenditure               226
Diphtheria                       19
HIV/AIDS                          0
GDP                             448
Population                      652
 thinness  1-19 years            34
 thinness 5-9 years              34
Income_composition_of_resources  167
Schooling                        163
dtype: int64
```

```
#There are many ways to fill missing values: https://www.geeksforgeeks.org/working-with-missing-data-in-pandas/
def fillmissing(df, feature, method):
  if method == "mode":
    df[feature] = df[feature].fillna(df[feature].mode()[0])
  elif method == "median":
    df[feature] = df[feature].fillna(df[feature].median())
  else:
    df[feature] = df[feature].fillna(df[feature].mean())

#We'll use the mean value
features_missing = X.columns[X.isna().any()]
for feature in features_missing:
  fillmissing(X, feature= feature, method= "mean")

#for Y we'll use the median value
Y.fillna(Y.median(), inplace=True)
```

```
Y = np.array(Y)
```

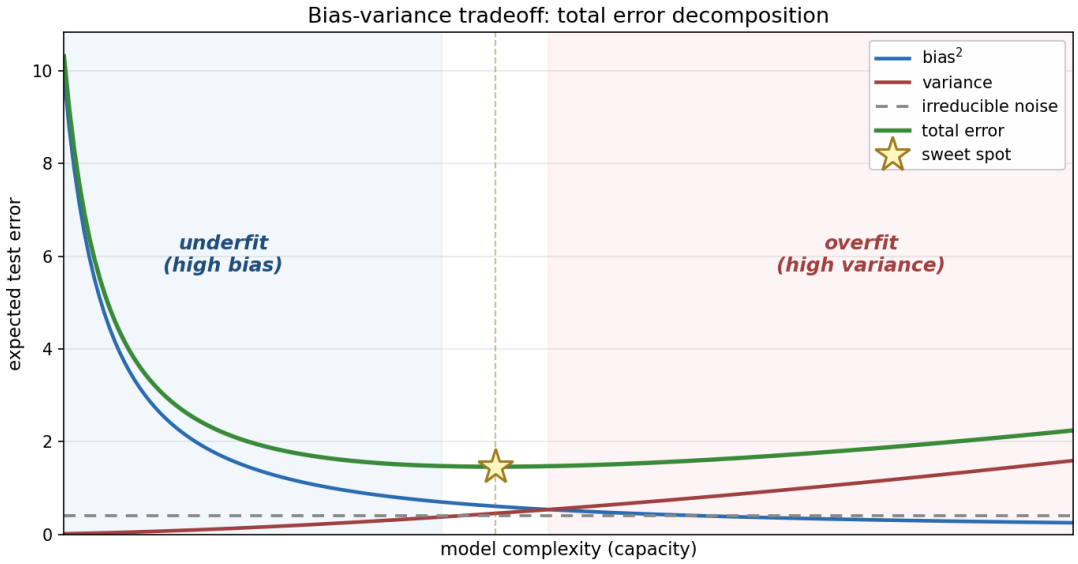
```
X.isnull().sum() #no more missing values
```

```
Year                             0
Life_expectancy                  0
Adult_mortality                  0
Infant_deaths                    0
```

```
Alcohol                          0
Percentage_expenditure           0
HepatitisB                       0
Measles                          0
BMI                              0
Under_five_deaths                0
Polio                            0
Total_expenditure                0
Diphtheria                       0
HIV/AIDS                         0
GDP                              0
Population                       0
 thinness  1-19 years            0
 thinness 5-9 years              0
Income_composition_of_resources  0
Schooling                        0
dtype: int64
```

7.8. The bias-variance tradeoff

Before fitting any model we should remember that test error decomposes into three parts: bias^2, **variance**, and an irreducible **noise** term that no model can remove. Underfit models have high bias (too rigid). Overfit models have high variance (memorise noise). The sweet spot minimises the sum.



7.9. Linear Regression

x - data

y - label

If we have **n** datapoints, then our data and label matrix will look like this.

$$\underset{(n,2)}{x} = \begin{bmatrix} 1 & x^{(1)} \\ 1 & x^{(2)} \\ \dots & \dots \\ 1 & x^{(n)} \end{bmatrix}, \quad \underset{(n,1)}{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \dots \\ y^{(n)} \end{bmatrix}$$

We want to find $\underset{(2,1)}{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$ for $h(x) = \hat{y} = \theta x$

If we substitute our $\boldsymbol{x}$ and θ in our hypothesis function we will get:

$$h(x) = \begin{bmatrix} 1 \cdot \theta_0 + x^{(1)}\theta_1 \\ 1 \cdot \theta_0 + x^{(2)}\theta_1 \\ \dots \\ 1 \cdot \theta_0 + x^{(200)}\theta_1 \end{bmatrix}$$

In case of linear regression we can use the ***Normal Equation*** to calculate the optimal theta

$\theta^* = (X^T X)^{-1}(X^T Y)$

7.10. Linear Regression with Tim

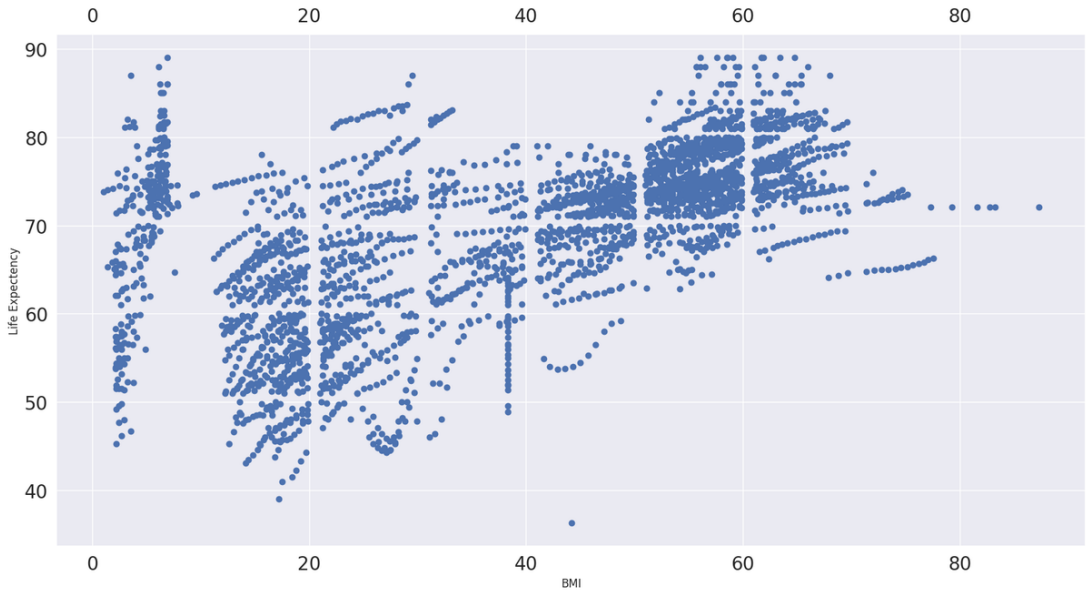
```
tim_data = X['BMI']
tim_data.head()
```

```
0    19.1
1    18.6
2    18.1
3    17.6
4    17.2
Name: BMI, dtype: float64
```

7.10.1. Visualizing Life expectancy vs BMI plot

```
sns.set(rc = {'figure.figsize':(20,10), 'font.weight': 'regular', 'font.size': 12, 'xtick.labelsize': 20,
   'ytick.labelsize': 20})
plt.scatter(X['BMI'], X['Life_expectancy'])
plt.xlabel('BMI')
plt.ylabel('Life Expectency')
```

```
Text(0, 0.5, 'Life Expectency')
```



7.11. Preprocessing data

```
tim_data   # display value
```

```
0       19.1
1       18.6
2       18.1
3       17.6
4       17.2
        ...
2933    27.1
2934    26.7
2935    26.3
2936    25.9
2937    25.5
Name: BMI, Length: 2938, dtype: float64
```

```
def preprocess(X):
    return np.stack((np.ones(X.shape), X)).T

tim_data_ready = preprocess(tim_data.to_numpy())
tim_data_ready.shape
```

```
(2938, 2)
```

```
tim_data_ready   # display value
```

```
array([[ 1. , 19.1],
       [ 1. , 18.6],
       [ 1. , 18.1],
       ...,
       [ 1. , 26.3],
       [ 1. , 25.9],
       [ 1. , 25.5]])
```

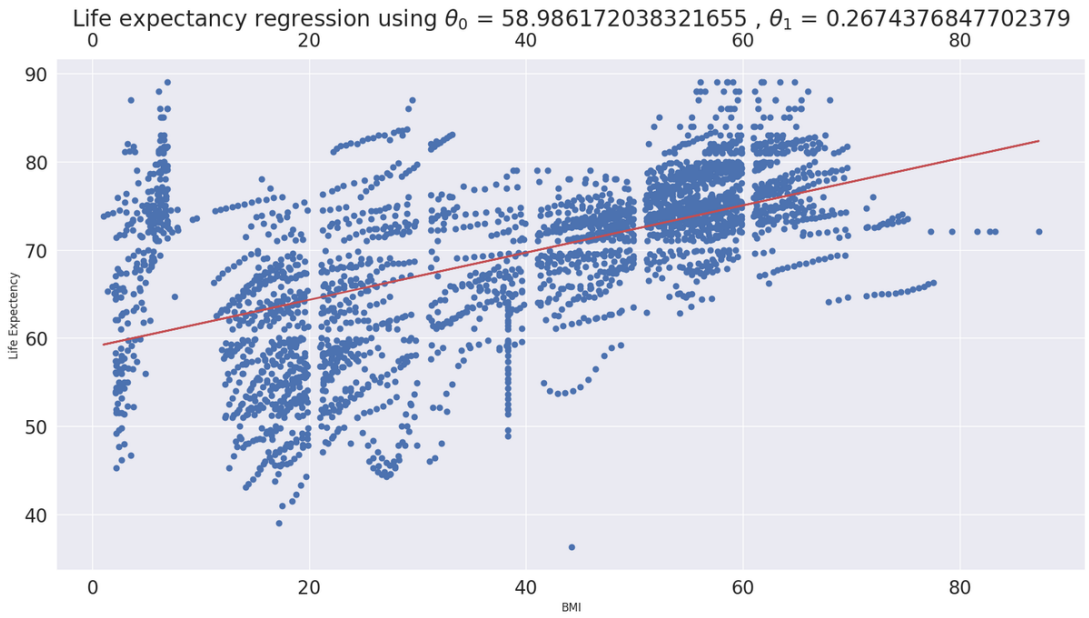
```
def normal_equation(X,Y):
    return np.linalg.inv(X.T @ X) @ (X.T @ Y)

tim_theta = normal_equation(tim_data_ready, Y)
print(f'y = mx + b')
print(f'y = {tim_theta[1]}x + {tim_theta[0]}')
```

```
y = mx + b
y = 0.2674376847702379x + 58.986172038321655
```

```
plt.title('Life expectancy regression using $\\theta_0$ = {} , $\\theta_1$ = {}'.format(tim_theta[0],
   tim_theta[1]), fontsize=24)
plt.scatter(tim_data,Y, marker='o')
plt.plot(tim_data,np.dot(tim_data_ready, tim_theta), color='r', linewidth=2)
plt.xlabel('BMI')
plt.ylabel('Life Expectency')
```

```
Text(0, 0.5, 'Life Expectency')
```




```
def life_expectancy(X, theta):
    return np.concatenate([[1], X]) @ theta

height = 1.82 #float(input("Please input your height (in meters): ")) #1.82
weight = 80 #float(input("Please input your weight (in kilograms): ")) #80
bmi = weight / height**2
print("BMI:", bmi)
life_exp_tim = life_expectancy([bmi], tim_theta)
print("Tim's life expectancy is", life_exp_tim.round(1) , "years.")
```

```
BMI: 24.151672503320853
Tim's life expectancy is 65.4 years.
```

```
height = 1.80 #float(input("Please input your height (in meters): ")) #1.82
weight = 73 #float(input("Please input your weight (in kilograms): ")) #80
bmi = weight / height**2
print("BMI:",bmi)
life_exp_tim_tom = life_expectancy([bmi], tim_theta)
print("Tom's life expectancy predicted by Tim's model is", life_exp_tim_tom.round(1) , "years.")
```

```
BMI: 22.530864197530864
Tom's life expectancy predicted by Tim's model is 65.0 years.
```

7.12. Linear Regression with Tom

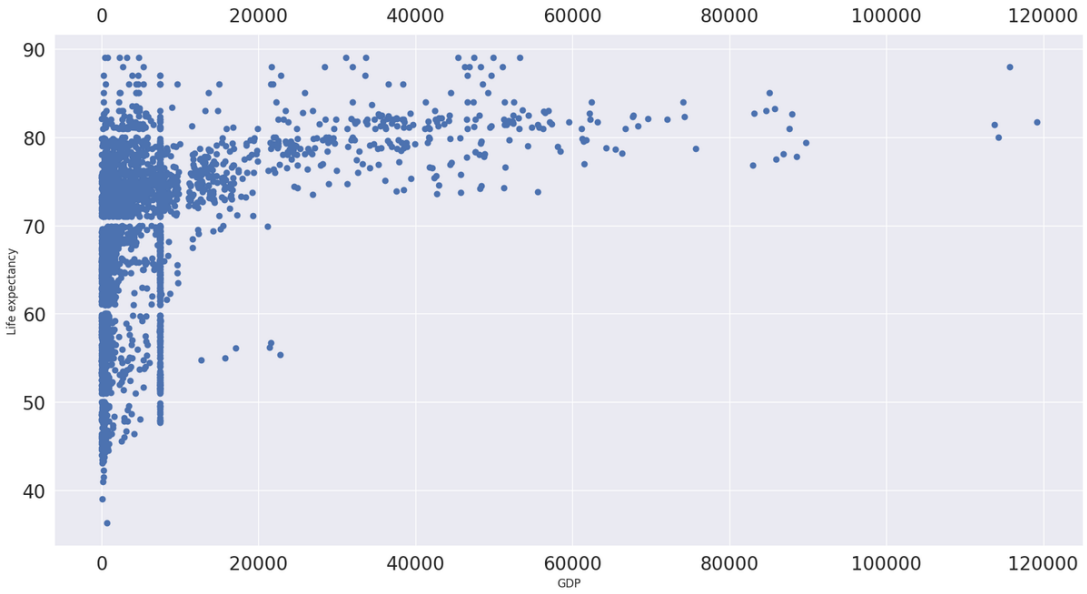
```
tom_data = X['GDP'].to_numpy()
tom_data
```

```
array([584.25921  , 612.696514 , 631.744976 , ...,  57.34834  ,
       548.587312 , 547.3588785])
```

7.13. Visualzing Life expectancy vs GDP plot

```
plt.scatter(tom_data, Y)
plt.xlabel('GDP')
plt.ylabel('Life expectancy')
```

```
Text(0, 0.5, 'Life expectancy')
```



7.14. Preprocessing data

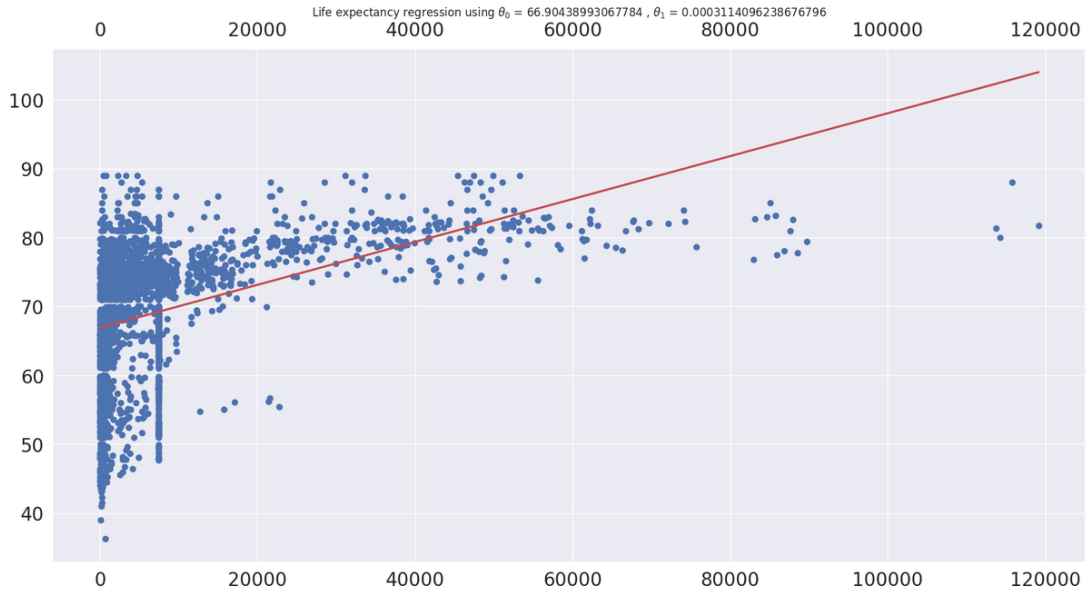
```
tom_data_ready = preprocess(tom_data)
tom_data_ready.shape
```

```
(2938, 2)
```

```
tom_theta = normal_equation(tom_data_ready, Y)
print(f'y = mx + b')
print(f'y = {tom_theta[1]}x + {tom_theta[0]}')
```

```
y = mx + b
y = 0.0003114096238676796x + 66.90438993067784
```

```
plt.title('Life expectancy regression using $\\theta_0$ = {} , $\\theta_1$ = {}'.format(tom_theta[0],
   tom_theta[1]))
plt.scatter(tom_data,Y, marker='o')
plt.plot(tom_data,np.dot(tom_data_ready, tom_theta), color='r', linewidth=2)
plt.show()
```




```
gdp = 8500 #float(input("Please input the GDP of your country: ")) #8500
life_exp_tom = life_expectancy([gdp], tom_theta)
print("Tom's life expectancy is", life_exp_tom , "years.")
```

```
Tom's life expectancy is 69.55137173355311 years.
```

```
gdp = 5000 #float(input("Please input the GDP of your country: ")) #5000
life_exp_tom_tim = life_expectancy([gdp], tom_theta)
print("Tim's life expectancy predicted by Tom is", life_exp_tom_tim , "years.")
```

```
Tim's life expectancy predicted by Tom is 68.46143805001624 years.
```

7.15. Multiple Regression

x - data

y - label

If we have **n** datapoints, then our data and label matrix will look like this.

$$\underset{(m,n)}{X} = \begin{bmatrix} 1 & x_1^{(1)} & \dots & x_n^{(1)} \\ 1 & x_1^{(2)} & \dots & x_n^{(2)} \\ \dots & \dots & \dots & \dots \\ 1 & x_1^{(m)} & \dots & x_n^{(m)} \end{bmatrix}, \quad \underset{(n,1)}{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \dots \\ y^{(n)} \end{bmatrix}$$

We want to find $\underset{(n,1)}{\theta} = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \dots \\ \theta_n \end{bmatrix}$ for $h(x) = \hat{y} = \theta x$

If we substitute our $\boldsymbol{x}$ and θ in our hypothesis function we will get:

$h(x) = \theta x$

In case of linear regression we can use the ***Normal Equation*** to calculate the optimal theta

$\theta^* = (X^T X)^{-1}(X^T Y)$

7.16. Multiple Regression with Tim

```
tim_data = X[['BMI', 'Adult_mortality', 'HIV/AIDS']]
tim_data.head()
```

```
BMI  Adult_mortality  HIV/AIDS
0  19.1            263.0       0.1
1  18.6            271.0       0.1
2  18.1            268.0       0.1
3  17.6            272.0       0.1
4  17.2            275.0       0.1
```

Adding a column of ones to the feature matrix lets the bias term be learned together with the weights via a single matrix operation. If `X` has shape `(n, d)`, the augmented matrix `[1, X]` has shape `(n, d + 1)`. The first entry of θ then becomes the intercept and the rest the per-feature slopes.

```
def add_ones(X):
    return np.concatenate([np.ones((X.shape[0], 1)),  X], axis=1)

tim_data_ready = add_ones(tim_data.to_numpy())
tim_data_ready.shape
```

```
(2938, 4)
```

```
tim_theta = normal_equation(tim_data_ready, Y)
print(f'theta = {tim_theta}')
```

```
theta = [ 6.95250308e+01  1.58686254e-01 -3.38199840e-02 -4.58041192e-01]
```

```
def RMSE(y_pred, y):
    return np.sqrt(metrics.mean_squared_error(y, y_pred))

tim_y_pred = np.dot(tim_data_ready, tim_theta)
tim_rmse = RMSE(tim_y_pred, Y)

print("Tim's RMSE: ", tim_rmse)
```

```
Tim's RMSE:  5.784750918898876
```

```
height = 1.82 #float(input("Please input your height (in meters): "))
weight = 80 #float(input("Please input your weight (in kilograms): "))
bmi = weight / height**2
adult_mortality = 53
hiv = 0.1
life_exp_tim = life_expectancy([bmi,adult_mortality,hiv], tim_theta)
print("Tim's life expectancy is", life_exp_tim .round(1), "years.")
```

```
Tim's life expectancy is 71.5 years.
```

```
height = 1.80 #float(input("Please input your height (in meters): "))
weight = 73 #float(input("Please input your weight (in kilograms): "))
bmi = weight / height**2
adult_mortality = 70
```

```
hiv = 0.1
life_exp_tim_tom = life_expectancy([bmi,adult_mortality,hiv], tim_theta)
print("Tom's life expectancy predicted by Tim is", life_exp_tim_tom.round(1) , "years.")
```

```
Tom's life expectancy predicted by Tim is 70.7 years.
```

7.17. Multiple Regression with Tom

```
tom_data = X[['GDP', 'Income_composition_of_resources', 'Schooling']]
tom_data.head()
```

```
GDP  Income_composition_of_resources  Schooling
0  584.259210                            0.479       10.1
1  612.696514                            0.476       10.0
2  631.744976                            0.470        9.9
3  669.959000                            0.463        9.8
4   63.537231                            0.454        9.5
```

```
tom_data_ready = add_ones(tom_data)
tom_data_ready.shape
```

```
(2938, 4)
```

```
tom_theta = normal_equation(tom_data_ready, Y)
print(f'theta = {tom_theta}')
```

```
theta = [4.49681347e+01 8.26882411e-05 1.42744076e+01 1.22489350e+00]
```

```
def RMSE(y_pred, y):
    return np.sqrt(metrics.mean_squared_error(y, y_pred))

tom_y_pred = np.dot(tom_data_ready, tom_theta)
tom_rmse = RMSE(tom_y_pred, Y)

print("Tom's RMSE: ", tom_rmse)
```

```
Tom's RMSE:  6.288890108092894
```

```
gdp = 8500
income = 0.8
school = 16.5
life_exp_tom = life_expectancy([gdp,income,school], tom_theta)
print("Tom's life expectancy is", life_exp_tom.round(1) , "years.")
```

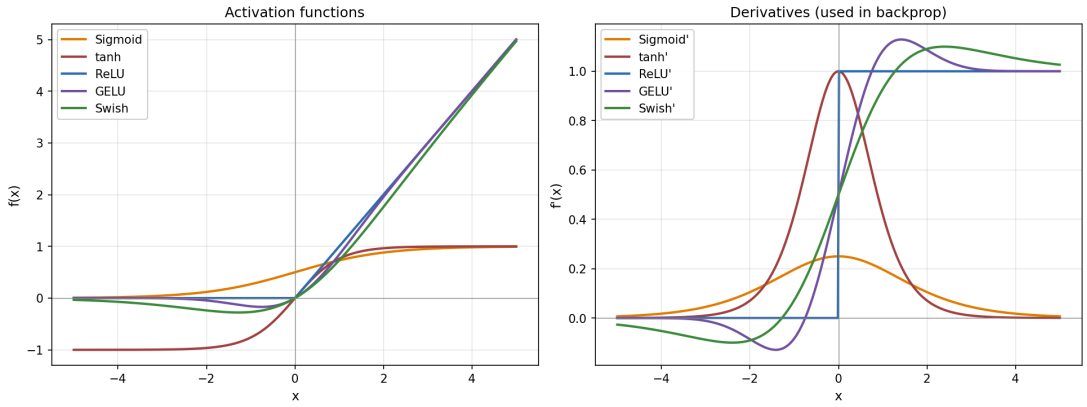
```
Tom's life expectancy is 77.3 years.
```

```
gdp = 5000
income = 0.77
school = 15.5
life_exp_tom_tim = life_expectancy([gdp,income,school], tom_theta)
print("Tim's life expectancy predicted by Tom is", life_exp_tom_tim.round(1) , "years.")
```

```
Tim's life expectancy predicted by Tom is 75.4 years.
```

7.18. A note on activation functions

Linear regression is, by definition, *linear*. To go beyond linear decision boundaries we stack linear layers and put a **non-linear activation function** between them. The choice of activation matters for both expressiveness and gradient flow.



ReLU and its smooth siblings (GELU, Swish) have become the default for deep networks because their derivatives do not saturate for positive inputs. Sigmoid and tanh saturate at the tails and contribute to the **vanishing gradient** problem in deep stacks.

7.19. Solving the problem using a Deep Neural Network

```
import torch
import torch.nn as nn
import torch.optim as optim
```

7.20. Preprocessing

7.20.1. Selecting features

Let's select those features whose positive correlation is >0.4 and negative correlation < -0.5.

```
correlation = life_exp.corr()
correlation
```

```
filtered = correlation['Life_expectancy']
filtered
```

```
Year                              0.170819
Life_expectancy                   1.000000
Adult_mortality                  -0.696251
Infant_deaths                    -0.196769
Alcohol                           0.390675
Percentage_expenditure            0.381418
HepatitisB                        0.203519
Measles                          -0.157767
BMI                               0.560453
Under_five_deaths                -0.222738
Polio                             0.461526
```

```
Total_expenditure                 0.208879
Diphtheria                        0.475378
HIV/AIDS                         -0.556703
GDP                               0.430211
Population                       -0.019683
 thinness  1-19 years            -0.472829
 thinness 5-9 years              -0.467295
Income_composition_of_resources   0.692560
Schooling                         0.714358
Name: Life_expectancy, dtype: float64
```

```
columns = filtered.index.values
columns
```

```
array(['Year', 'Life_expectancy', 'Adult_mortality', 'Infant_deaths',
       'Alcohol', 'Percentage_expenditure', 'HepatitisB', 'Measles',
       'BMI', 'Under_five_deaths', 'Polio', 'Total_expenditure',
       'Diphtheria', 'HIV/AIDS', 'GDP', 'Population',
       ' thinness  1-19 years', ' thinness 5-9 years',
       'Income_composition_of_resources', 'Schooling'], dtype=object)
```

```
mask = (filtered < -0.5) | (filtered > 0.4)
mask
```

```
Year                             False
Life_expectancy                   True
Adult_mortality                   True
Infant_deaths                    False
Alcohol                          False
Percentage_expenditure           False
HepatitisB                       False
Measles                          False
BMI                               True
Under_five_deaths                False
Polio                             True
Total_expenditure                False
Diphtheria                        True
HIV/AIDS                          True
GDP                               True
Population                       False
 thinness  1-19 years            False
 thinness 5-9 years              False
Income_composition_of_resources   True
Schooling                         True
Name: Life_expectancy, dtype: bool
```

```
filtered_columns = columns[mask][1:]
filtered_columns
```

```
array(['Adult_mortality', 'BMI', 'Polio', 'Diphtheria', 'HIV/AIDS', 'GDP',
       'Income_composition_of_resources', 'Schooling'], dtype=object)
```

```
X = life_exp[filtered_columns]
X.head()
```

```
Adult_mortality   BMI  Polio  Diphtheria  HIV/AIDS         GDP  \
0            263.0  19.1    6.0        65.0       0.1  584.259210
1            271.0  18.6   58.0        62.0       0.1  612.696514
2            268.0  18.1   62.0        64.0       0.1  631.744976
```

```
3            272.0  17.6   67.0        67.0       0.1  669.959000
4            275.0  17.2   68.0        68.0       0.1   63.537231

   Income_composition_of_resources  Schooling
0                            0.479       10.1
1                            0.476       10.0
2                            0.470        9.9
3                            0.463        9.8
4                            0.454        9.5
```

```
X.describe()
```

```
Adult_mortality          BMI        Polio   Diphtheria     HIV/AIDS  \
count      2938.000000  2938.000000  2938.000000  2938.000000  2938.000000
mean        164.796448    38.321247    82.550188    82.324084     1.742103
std         124.080302    19.927677    23.352143    23.640073     5.077785
min           1.000000     1.000000     3.000000     2.000000     0.100000
25%          74.000000    19.400000    78.000000    78.000000     0.100000
50%         144.000000    43.000000    93.000000    93.000000     0.100000
75%         227.000000    56.100000    97.000000    97.000000     0.800000
max         723.000000    87.300000    99.000000    99.000000    50.600000

                GDP  Income_composition_of_resources    Schooling
count    2938.000000                      2938.000000  2938.000000
mean     7483.158469                         0.627551    11.992793
std     13136.800417                         0.204820     3.264381
min         1.681350                         0.000000     0.000000
25%       580.486996                         0.504250    10.300000
50%      3116.561755                         0.662000    12.100000
75%      7483.158469                         0.772000    14.100000
max    119172.741800                         0.948000    20.700000
```

```
Y = life_exp['Life_expectancy']
Y.head()
```

```
0    65.0
1    59.9
2    59.9
3    59.5
4    59.2
Name: Life_expectancy, dtype: float64
```

```
X.shape, Y.shape
```

```
((2938, 8), (2938,))
```

```
X = X.to_numpy()
Y = Y.to_numpy()
```

7.20.2. Normalizing values

```
def standardize(X):
    mu = X.mean()
    sigma = X.std()
```

```
    normalized = ( X - mu ) / sigma
    return normalized, mu, sigma
```

```
x_norm, mu, sigma = standardize(X)
```

7.20.3. Shuffling and splitting the dataset

Use a train:validation:test split ration of 6:2:2

```
dataset = np.concatenate((x_norm, Y[:,None]), axis=1)
dataset.shape
```

```
(2938, 9)
```

```
from sklearn.model_selection import train_test_split

# implement your solution BELOW
data_train, rest = train_test_split(dataset, train_size=0.6, shuffle=True)
data_val, data_test = train_test_split(rest, train_size=0.5)

print((data_train.shape, data_val.shape, data_test.shape))
```

```
((1762, 9), (588, 9), (588, 9))
```

```
from torch.utils.data import Dataset, DataLoader

class CustomDataset(Dataset):

    def __init__(self, dataset):
        self.features = torch.from_numpy(dataset[:,:-1]).float()
        self.labels = torch.from_numpy(dataset[:,-1]).float().unsqueeze(1)

    def __len__(self):
        return self.features.shape[0]

    def __getitem__(self, idx):
        return self.features[idx, :], self.labels[idx]

train_dataset = CustomDataset(data_train)
val_dataset = CustomDataset(data_val)
test_dataset = CustomDataset(data_test)

dataloader_train = DataLoader(train_dataset, batch_size=32, shuffle=True)
dataloader_val = DataLoader(val_dataset, batch_size=32, shuffle=True)
dataloader_test = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

7.21. Defining DataLoader and Model

```
# implement your solution BELOW
class CustomModel(nn.Module):
    def __init__(self, in_features):
        super(CustomModel, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(in_features, 20),
            nn.ReLU(),
            nn.Linear(20, 1)
        )
```

```
    def forward(self, X):
        return self.model(X)
```

```
model = CustomModel(in_features=8)
lossfn = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop with early stopping
num_epochs = 200
patience = 20
best_val_loss = float('inf')
patience_counter = 0
best_model_wts = None

train_losses = []
val_losses = []

for epoch in range(num_epochs):
    for x, y in dataloader_train:
    # Training phase
        model.train()
        optimizer.zero_grad()
        outputs = model(x)
        loss = lossfn(outputs, y)
        loss.backward()
        optimizer.step()

    train_losses.append(loss.mean().item())

    # Validation phase
    model.eval()
    with torch.no_grad():
        for x, y in dataloader_val:
            val_outputs = model(x)
            val_loss = lossfn(val_outputs, y)

    val_losses.append(val_loss.mean().item())

    if epoch % 10 == 0:
        print(f"Epoch {epoch}/{num_epochs}, Train Loss: {loss.item()}, Val Loss: {val_loss.item()}")
    best_model_wts = model.state_dict()

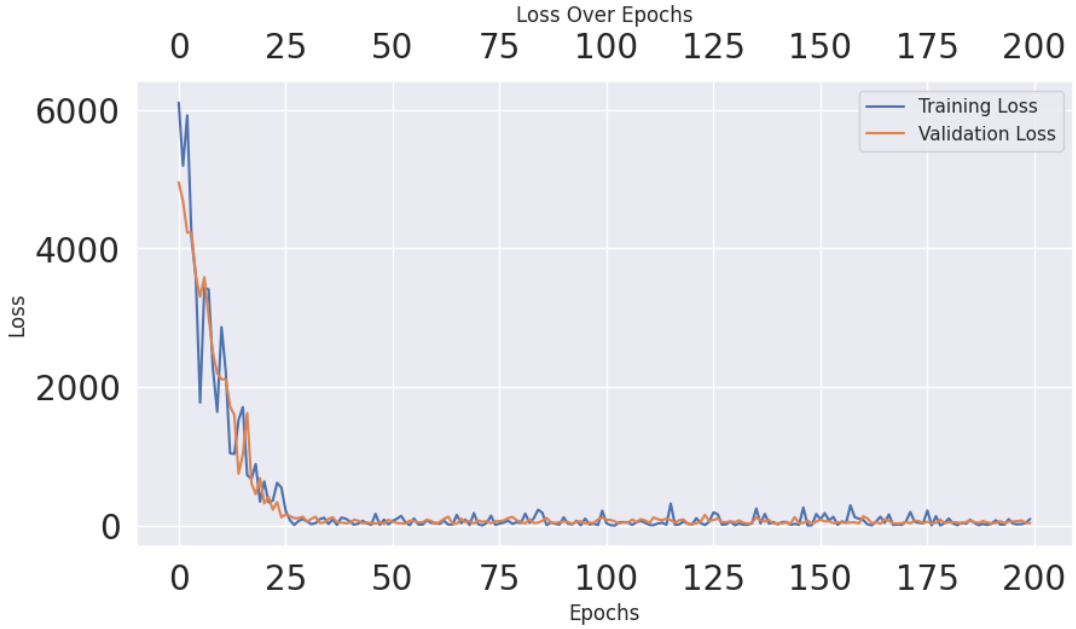
# Load the best model weights (if saved)

# Plotting the training and validation losses
plt.figure(figsize=(10, 5))
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss Over Epochs')
plt.show()

best_model = model.load_state_dict(best_model_wts)
```

```
Epoch 0/200, Train Loss: 6102.595703125, Val Loss: 4953.0283203125
Epoch 10/200, Train Loss: 2862.83837890625, Val Loss: 2111.593505859375
Epoch 20/200, Train Loss: 636.7042236328125, Val Loss: 320.8240051269531
```

```
Epoch 30/200, Train Loss: 59.794063568115234, Val Loss: 58.6087760925293
Epoch 40/200, Train Loss: 58.898460388183594, Val Loss: 40.4240608215332
Epoch 50/200, Train Loss: 69.2608413696289, Val Loss: 56.9272575378418
Epoch 60/200, Train Loss: 38.48622131347656, Val Loss: 22.441946029663086
Epoch 70/200, Train Loss: 18.650909423828125, Val Loss: 72.29264831542969
Epoch 80/200, Train Loss: 37.32881546020508, Val Loss: 53.0180549621582
Epoch 90/200, Train Loss: 118.40205383300781, Val Loss: 57.76023864746094
Epoch 100/200, Train Loss: 30.945518493652344, Val Loss: 86.01058197021484
Epoch 110/200, Train Loss: 9.037251472473145, Val Loss: 36.997135162353516
Epoch 120/200, Train Loss: 15.778244972229004, Val Loss: 23.378000259399414
Epoch 130/200, Train Loss: 3.8428943157196045, Val Loss: 48.43668746948242
Epoch 140/200, Train Loss: 17.050846099853516, Val Loss: 33.16398620605469
Epoch 150/200, Train Loss: 89.37934112548828, Val Loss: 83.75541687011719
Epoch 160/200, Train Loss: 79.60900115966797, Val Loss: 133.830322265625
Epoch 170/200, Train Loss: 60.766502380371094, Val Loss: 52.51370620727539
Epoch 180/200, Train Loss: 102.09611511230469, Val Loss: 48.58198928833008
Epoch 190/200, Train Loss: 26.9462947845459, Val Loss: 31.899065017700195
```



7.21.1. Make a prediction for Tim and Tom:

7.22. Tim:

Adult Mortality: 53
BMI = 24.2
Polio: 98
Difteria: 98
HIV: 0.1
GDP = 5000

Income = 0.77
School = 15.5

7.22.1. Tom:

Adult Mortality: 70
BMI = 22.5
Polio: 97
Difteria: 97
HIV: 0.1
GDP = 8500
Income = 0.8
School = 16.5

```
tim = [53, 24.2, 98, 98, 0.1, 5000, 0.77, 15.5]
tom = [70, 22.5, 97, 97, 0.1, 8500, 0.8, 16.5]
```

7.23. Standardizing the inputs and running a prediction

```
tim_std = (tim - mu) / sigma
tom_std = (tom - mu) / sigma
(tim_std, tom_std)
```

```
(array([-0.17704225, -0.18252374, -0.16847743, -0.16847743, -0.18711067,
         0.76451685, -0.18698315, -0.1841796 ]),
 array([-0.17380666, -0.1828473 , -0.16866776, -0.16866776, -0.18711067,
         1.43066945, -0.18697744, -0.18398927]))
```

```
with torch.no_grad():
    life_exp_tim = model(torch.Tensor(tim_std))

print(f'Tim\'s life expectancy is {life_exp_tim.numpy().round(2)} years')
```

```
Tim's life expectancy is [72.92] years
```

```
with torch.no_grad():
    life_exp_tim = model(torch.Tensor(tom_std))

print(f'Tom\'s life expectancy is {life_exp_tim.numpy().round(2)} years')
```

```
Tom's life expectancy is [73.29] years
```

7.24. Interpreting the fitted coefficients

Once the normal equation has given us θ, each entry has a concrete meaning, and it is unitful. For single-variable regression, θ`[0]` is the intercept (the predicted y when $x = 0$), and θ`[1]` is the slope (how much y changes per one-unit change in x).

For the multi-variable case trained on standardised features, the intuition shifts: each θ`[i]` is the predicted change in y per **one-standard-deviation** change in the corresponding feature, holding all other features fixed. This is often easier to compare across features than the raw-unit coefficients.

Two rules of thumb for sanity-checking a fit:

1. **Signs should match intuition.** If a feature correlates positively with `y` in the EDA plots, its fitted coefficient should be positive too. A flipped sign usually indicates multicollinearity or a data-leakage feature.
2. **Magnitudes should be stable across runs.** Small changes to the preprocessing (e.g. dropping a few rows with missing values) should not produce wildly different coefficients. Unstable fits mean the problem is poorly conditioned.

```
# Pretty-print the fitted theta with feature names so the reader can
# match each coefficient to its meaning.
def report_coefficients(theta, feature_names):
    print(f'{"intercept":30s}  {theta[0]:+8.3f}')
    for name, coef in zip(feature_names, theta[1:]):
        print(f'{name:30s}  {coef:+8.3f}')

# Example: report the Tim multi-regression fit from earlier in the notebook
try:
    report_coefficients(tim_theta, ['BMI', 'Adult_mortality', 'HIV/AIDS'])
except NameError:
    print('Run the multi-regression cells above first to populate tim_theta.')
```

7.25. Implement Early Stopping

```
model = CustomModel(in_features=8)
lossfn = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop with early stopping
num_epochs = 200
patience = 20
best_val_loss = float('inf')
patience_counter = 0
best_model_wts = None

train_losses = []
val_losses = []

for epoch in range(num_epochs):
    for x, y in dataloader_train:
    # Training phase
        model.train()
        optimizer.zero_grad()
        outputs = model(x)
        loss = lossfn(outputs, y)
        loss.backward()
        optimizer.step()

    train_losses.append(loss.mean().item())

    # Validation phase
    model.eval()
    with torch.no_grad():
        for x, y in dataloader_val:
            val_outputs = model(x)
            val_loss = lossfn(val_outputs, y)
```

```
    val_losses.append(val_loss.mean().item())

    # Check early stopping
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        patience_counter = 0
        # Save the best model (optional)
        best_model_wts = model.state_dict()
    else:
        patience_counter += 1

    if patience_counter >= patience:
        print(f"Early stopping at epoch {epoch}")
        break

    if epoch % 10 == 0:
        print(f"Epoch {epoch}/{num_epochs}, Train Loss: {loss.item()}, Val Loss: {val_loss.item()}")

# Load the best model weights (if saved)

# Plotting the training and validation losses
plt.figure(figsize=(10, 5))
plt.plot(train_losses, label='Training Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss Over Epochs')
plt.show()

best_model = model.load_state_dict(best_model_wts)
```

7.26. Homework (Optional)

1. **Feature selection by correlation.** Using the life-expectancy dataset, compute the correlation of every feature with the target. Fit a regression using only features whose `|correlation|` exceeds 0.4 and compare its RMSE on a held-out test split against a regression using all features.
2. **Standardise vs. raw features.** Train one regression on raw features and a second on standardised features (`(x - μ) / σ` per feature). Report the fitted coefficients in both cases and explain why their magnitudes differ even though the underlying model is the same.
3. **Residual plot.** After fitting any regression, plot residuals (`y_pred - y_true`) against predicted values. A well-fitted model's residuals should scatter randomly around zero. Systematic patterns suggest missing features or a non-linear relationship.
4. **Manual normal equation.** Without using `np.linalg.solve` or `inv`, implement the closed-form $\theta = (X^T X)^{-1} X^T y$ using `np.linalg.pinv`. Compare with a small `sklearn.linear_model.LinearRegression` fit and verify the coefficients match to 4 decimal places.
5. **DNN vs. linear baseline.** For the `CustomModel` from the DNN section, report training and validation RMSE after 50 epochs. Compare against the multi-variable linear regression RMSE on the same features. Argue in 2-3 sentences whether the added model capacity is justified.

6. **Early stopping.** Modify the training loop so it stops early when validation RMSE has not improved for 5 consecutive epochs. Report the epoch at which training stopped and the final validation RMSE.

Chapter 8

Image Classification with CNNs

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1sS6vjgYlf6F2YhrC51SMHPwJba20DxxJ?usp=sharing

Image classification using convolutional neural networks (CNNs) employs a multi-layered architecture that automatically extracts hierarchical features from raw pixel data, enabling the model to discern complex patterns within images.

In this practice, we will use the CIFAR-10 dataset, a benchmark collection of 60,000 32x32 color images spanning 10 classes (with 50,000 training and 10,000 test image), which can be explored further at the official CIFAR website. We will develop a custom, lightweight convolutional neural network (CNN) to accurately classify these images, providing hands-on experience with core deep learning and computer vision techniques.

8.1. Necessary Imports

```
!pip install torchsummary --quiet
```

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
import copy
import random
from torchsummary import summary
from IPython.display import Image as IMG
from torch.utils.data import Subset

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```



8.2. Data - CIFAR10

Let's use a simple built-in dataset, that is available and easy to download. The CIFAR-10 dataset (Canadian Institute for Advanced Research, 10 classes) is a subset of the Tiny Images dataset and consists of 60000 32x32 color images. Good collection of datasets and methods by Meta.

PyTorch's `torchvision.transforms` module provides a comprehensive set of transformations that are commonly used for data augmentation and preprocessing in computer vision tasks. These transforms can be applied to images to enhance model training by artificially expanding the training dataset or by preprocessing the data to better suit the learning algorithms.

Here's a list of different transformation methods provided by PyTorch's `torchvision.transforms`:

8.2.1. Geometric Transformations

- **Resize**: Resize the input image to a specified size.
- **CenterCrop**: Crop the image at the center.
- **RandomCrop**: Crop the image at a random location.
- **RandomResizedCrop**: Crop a random portion of the image and resize it.
- **RandomHorizontalFlip**: Horizontally flip the image with a given probability.
- **RandomVerticalFlip**: Vertically flip the image with a given probability.
- **RandomRotation**: Rotate the image by an angle.
- **RandomAffine**: Apply random affine transformations such as rotation, translation, scaling, and shearing.
- **RandomPerspective**: Apply a random perspective transformation.
- **FiveCrop**: Crop the image into four corners and the central crop.
- **TenCrop**: Similar to `FiveCrop`, but also includes flipped versions of the crops.
- **Pad**: Pad the image on all sides with a specified value.

8.2.2. Color Transformations

- **Grayscale**: Convert the image to grayscale.
- **RandomGrayscale**: Convert the image to grayscale with a given probability.
- **ColorJitter**: Randomly change the brightness, contrast, saturation, and hue of the image.
- **RandomInvert**: Randomly invert the colors of the image.
- **RandomPosterize**: Reduce the number of bits for each color channel.
- **RandomSolarize**: Invert all pixel values above a certain threshold.
- **RandomAdjustSharpness**: Adjust the sharpness of the image.
- **RandomAutocontrast**: Apply an autocontrast operation on the image.
- **RandomEqualize**: Equalize the histogram of the image.

8.2.3. Miscellaneous Transformations

- **GaussianBlur**: Apply Gaussian blur to the image.
- **RandomErasing**: Randomly erase a rectangle region in the image.
- **Normalize**: Normalize a tensor image with mean and standard deviation.
- **ToTensor**: Convert a `PIL.Image` or `numpy.ndarray` to a tensor.
- **Lambda**: Apply a user-defined lambda function as a transformation.

These transforms can be combined using `transforms.Compose` to create complex data augmentation pipelines that are crucial for training robust machine learning models.

For more detailed information, you can explore the official PyTorch documentation.

The `DataLoader` in PyTorch is a powerful tool designed to handle the complexities of loading and managing data during the training process. It is particularly useful when working with large datasets, as it helps to streamline the data loading process by handling batching, shuffling, and parallel data loading.

8.2.4. Key Features of `DataLoader`

1. **Batching**: The `DataLoader` allows you to split your dataset into smaller batches. This is crucial for training models on large datasets, as it enables the processing of smaller subsets of data at a time, reducing memory usage and speeding up training.
2. **Shuffling**: To prevent the model from learning the order of the data, which could lead to overfitting, the `DataLoader` can shuffle the data at every epoch. This ensures that the model generalizes better to unseen data.
3. **Parallel Loading with Multiple Workers**: The `DataLoader` can utilize multiple subprocesses to load data in parallel using the `num_workers` parameter. This is particularly beneficial when dealing with large datasets or complex preprocessing, as it reduces the data loading time by parallelizing the process.
4. **Custom Samplers**: The `DataLoader` allows the use of custom sampling strategies by providing different `Sampler` classes. This is useful when you need to apply specific sampling methods, such as weighted sampling, which can help in dealing with imbalanced datasets.
5. **Support for Iterable Datasets**: Besides regular datasets, the `DataLoader` can work with iterable datasets, which are particularly useful when the data is generated on-the-fly or when working with streams of data.
6. **Collate Function**: The `DataLoader` provides an option to define a custom `collate_fn`, which allows you to specify how the individual data samples should be merged into a single batch. This is useful for handling variable-length sequences or more complex data structures.

```
transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomVerticalFlip(),
    transforms.RandomAffine(degrees=(5, 50), translate=(0.1, 0.3), scale=(0.9, 1.1)),
    transforms.RandomPerspective(distortion_scale=0.2),
    transforms.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
```

```
full_trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
./data/cifar-10-python.tar.gz
```

```
100%|##########| 170M/170M [00:04<00:00, 42.6MB/s]
```

```
Extracting ./data/cifar-10-python.tar.gz to ./data
```

```
# Define how many samples per batch to load
batch_size = 16

# Calculate the number of samples for the training set based on the percentage
train_percentage = 90
train_size = len(full_trainset)
train_size = int(train_size * (train_percentage / 100))

# Generate a list of indices from 0 to len(full_trainset) - 1
indices = list(range(0, len(full_trainset)))
```

```
print(indices)
random.seed(42)
random.shuffle(indices)
print(indices)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41,
42, 43, 44, 45, 46, 4 ... [338710 chars truncated]
[9926, 35283, 27382, 36541, 23204, 30508, 24079, 48121, 43668, 43464, 14772,
4994, 6959, 23321, 29370, 19160, 32006, 29728, 39381, 21886, 1380, 5942, 27236,
47325, 7675, 21429, 790 ... [338710 chars truncated]
```

```
# Select the first `train_size` indices for the training set
train_indices = indices[:train_size]

# Create a subset of the training dataset using the calculated indices
small_trainset = Subset(full_trainset, train_indices)

# Create a DataLoader for the subset training dataset
trainloader = torch.utils.data.DataLoader(small_trainset, batch_size=batch_size, shuffle=True, num_workers=0)
```

```
# Calculate the number of samples for the validation set based on the percentage
validation_percentage = 10  # Percentage of the dataset to be used for validation
valid_size = len(full_trainset)  # Total size of the dataset (50k)
validation_size = int(valid_size * (validation_percentage / 100))  # Number of validation samples
```

```
# Generate the indices for the validation set
valid_indices = indices[train_size:train_size+valid_size]

# Create a subset of the dataset for validation using the calculated indices
small_validationset = Subset(full_trainset, valid_indices)

# Create a DataLoader for the validation dataset
validloader = torch.utils.data.DataLoader(small_validationset, batch_size=batch_size, shuffle=False,
   num_workers=0)
```

```
test_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])

# Load the full CIFAR-10 test dataset without applying transformations
# Note: Test sets typically do not apply the same data augmentation transformations as training sets
testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=test_transform)

# Create a DataLoader for the test dataset
testloader = torch.utils.data.DataLoader(testset, batch_size=batch_size, shuffle=False, num_workers=0)
```

```
(test_size, _, _, _) = testset.data.shape  # Get the size of the test dataset

# CIFAR-10 dataset has 10 classes, listed here for reference
classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
Files already downloaded and verified
```

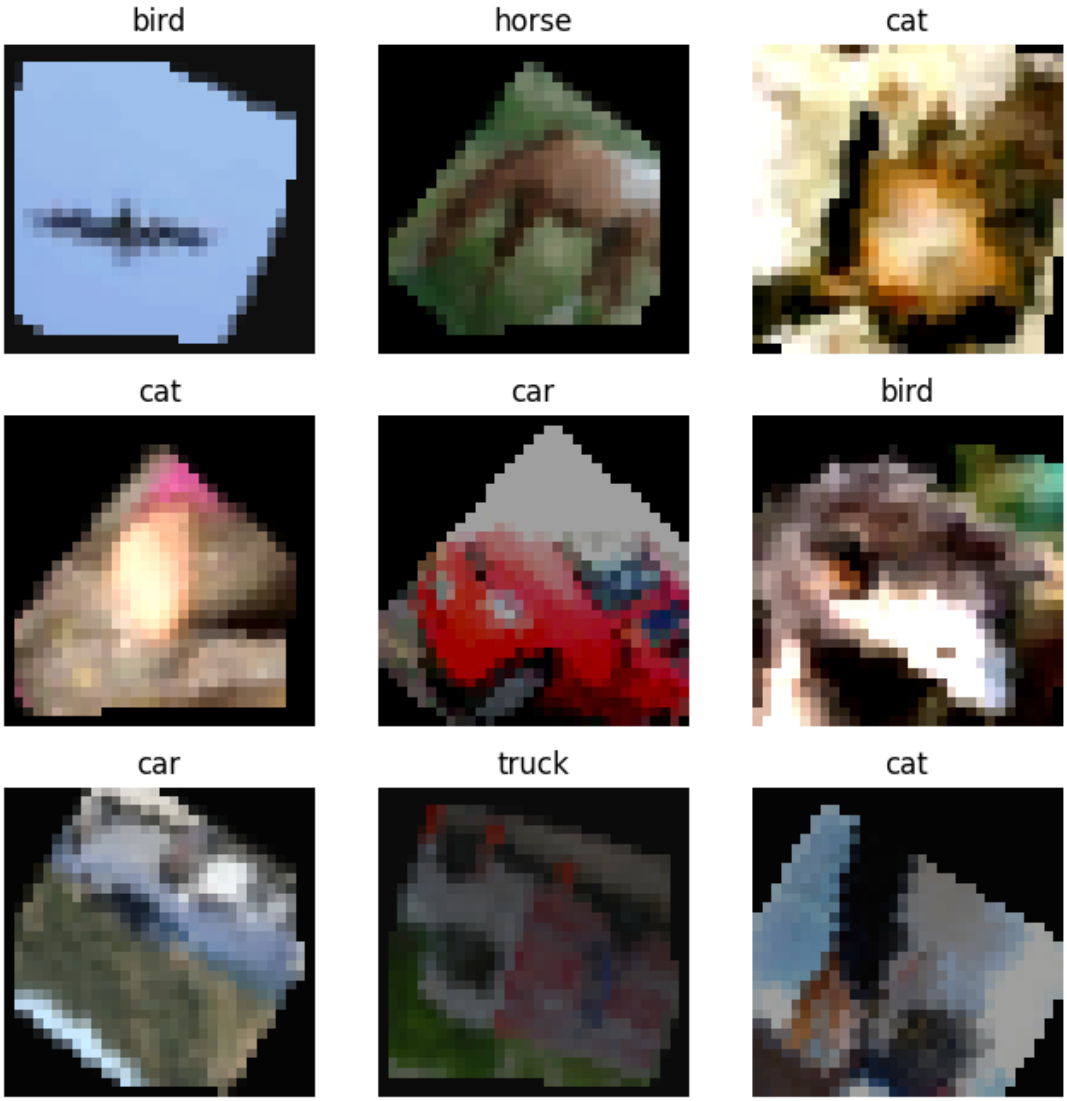
```
print("Number of Images in the Training set:\n",len(small_trainset))
print("Number of Images in the Validation set:\n",len(small_validationset))
print("Test set shape (Number of Images, Height, Width, Number of channels):\n", testset.data.shape)
print("Available classes: ", classes)
```

```
Number of Images in the Training set:
 45000
Number of Images in the Validation set:
 5000
Test set shape (Number of Images, Height, Width, Number of channels):
 (10000, 32, 32, 3)
Available classes:  ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog',
'horse', 'ship', 'truck')
```

Plot images

```
def plot_images(dataset):
    figure = plt.figure(figsize=(8, 8))
    cols, rows = 3, 3
    for i in range(1, cols * rows + 1):
        sample_idx = torch.randint(len(dataset), size=(1,)).item()
        img, label = dataset[sample_idx]
        img = img / 2 + 0.5 # unnormalize since it was normalized during loading
        npimg = img.numpy()
        img = np.transpose(npimg, (1, 2, 0))
        figure.add_subplot(rows, cols, i)
        plt.title(classes[label])
        plt.axis("off")
        plt.imshow(img)
    plt.show()
```

```
plot_images(full_trainset)
```




```
plot_images(testset)
```



8.3. Create and Train a Multi-Layered Perceptron (MLP)

In this example, we intentionally treat every pixel of an image as an independent feature, a practice that is not recommended for image classification. This is because such an approach ignores the spatial relationships and local patterns inherent in images, which are critical for effective feature extraction. Our feedforward network consists of three linear layers: the first layer accepts an input of size `3×32×32` (3072 pixels per image in the **CIFAR-10 dataset**), and the final layer outputs logits corresponding to the `10` classes in the dataset.

```
class FeedForwardNet(nn.Module):
    def __init__(self, num_classes=10):
        super(FeedForwardNet, self).__init__()
        self.flatten = nn.Flatten()
        self.fc1 = nn.Linear(3 * 32 * 32, 512)
        self.fc2 = nn.Linear(512, 256)
        self.fc3 = nn.Linear(256, num_classes)
```

```
    def forward(self, x):
        x = self.flatten(x)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

8.3.1. Let's create the neural network, set up the device and get a simple vizualisation

```
# Lightweight feed-forward net used as a baseline before the CNN.
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Using device:', device)

ff_net = FeedForwardNet(num_classes=10).to(device)
summary(ff_net, (3, 32, 32))     # the feed-forward net flattens the input
```

8.4. Loss function

`CrossEntropyLoss` is a commonly used loss function in PyTorch for classification tasks. It combines `nn.LogSoftmax` and `nn.NLLLoss` in one single class. This loss function is particularly useful when training models for multi-class classification, where the target is expected to be in the form of class indices (not one-hot encoded).

Key Points: - **Input**: The input to `nn.CrossEntropyLoss` should be raw, unnormalized scores (also known as logits) of shape `(N, C)` where `N` is the batch size and `C` is the number of classes. - **Target**: The target should contain the class indices, with shape `(N)` for each sample in the batch. - **Output**: The output is a scalar that represents the average negative log-likelihood across the batch.

Example input and target: input = torch.randn(3, 5, requires_grad=True) # 3 samples, 5 classes target = torch.tensor([1, 0, 4]) # Class indices for each sample

8.4.1. Optimizers and their options

There are some pre-built Optimizers in PyTorch, they are sufficient in most cases, especially if their parameters are well set. The two most well-known are ADAM and SGD, both of which originate from Gradient Descent, which we implemented earlier.

- **S**tochastic **G**radient **D**escent (SGD)
- **ADA**ptive **M**oment optimizer (ADAM)
- A good general overview

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(ff_net.parameters(), lr=0.01, momentum=0.9)
n_epochs = 10
```

8.5. Train

- load the batch - inputs, labels = data[0].to(device), data[1].to(device)
- zero the parameter gradients - optimizer.zero_grad()

- forward function - outputs = convnet(inputs)
- calculate the loss - loss = criterion(outputs, labels)
- calculate the gradient - loss.backward()
- update the parameters - optimizer.step() - update the parameters

validation: - “with torch.no_grad” is recommended for comparison it keeps the “training layers” - “model.eval() remove every”training layers" - calculate the loss - loss = criterion(outputs, labels)

```
hist_train = []
hist_valid = []
best_loss = float('inf')
best_model_wts = copy.deepcopy(ff_net.state_dict())
early_stop_tolerant = 10
early_stop_count = 0
```

```
for epoch in range(n_epochs):
    ff_net.train()
    train_loss = 0.0
    for inputs, labels in trainloader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = ff_net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    train_loss /= len(trainloader)
    hist_train.append(train_loss)

    ff_net.eval()
    valid_loss = 0.0
    with torch.no_grad():
        for inputs, labels in validloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = ff_net(inputs)
            loss = criterion(outputs, labels)
            valid_loss += loss.item()

    valid_loss /= len(validloader)
    hist_valid.append(valid_loss)

    # Early stopping check (simple version)
    if valid_loss < best_loss:
        best_loss = valid_loss
        best_model_wts = copy.deepcopy(ff_net.state_dict())
        early_stop_count = 0
    else:
        early_stop_count += 1
        if early_stop_count >= early_stop_tolerant:
            print("Early stopping triggered")
            break

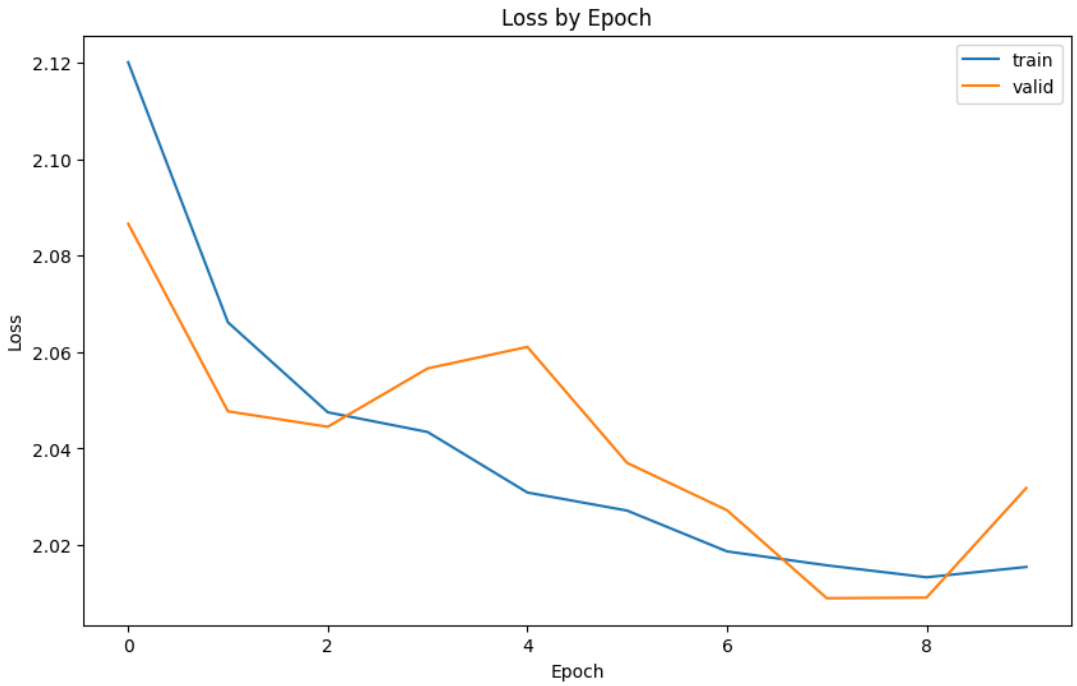
    print(f"Epoch {epoch+1}/{n_epochs}: Train Loss = {train_loss:.4f}, Valid Loss = {valid_loss:.4f}")

print("Finished training the feedforward network.")
```

```
Epoch 1/10: Train Loss = 2.1201, Valid Loss = 2.0866
Epoch 2/10: Train Loss = 2.0662, Valid Loss = 2.0477
Epoch 3/10: Train Loss = 2.0475, Valid Loss = 2.0445
Epoch 4/10: Train Loss = 2.0434, Valid Loss = 2.0566
Epoch 5/10: Train Loss = 2.0309, Valid Loss = 2.0611
Epoch 6/10: Train Loss = 2.0271, Valid Loss = 2.0370
Epoch 7/10: Train Loss = 2.0187, Valid Loss = 2.0272
Epoch 8/10: Train Loss = 2.0158, Valid Loss = 2.0090
Epoch 9/10: Train Loss = 2.0133, Valid Loss = 2.0091
Epoch 10/10: Train Loss = 2.0154, Valid Loss = 2.0318
Finished training the feedforward network.
```

```
plt.figure(figsize=(10,6))
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss by Epoch')

plt.plot(hist_train)
plt.plot(hist_valid)
#plt.ylim((1,3))
plt.legend(['train', 'valid'])
```



Save trained model

```
PATH = './fftnet.pth'
torch.save(ff_net.state_dict(), PATH)
```

Load saved model and predict some images

```
def imshow(img, mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5)):
    img = img.cpu().numpy()
    img = img.transpose((1, 2, 0))  # Convert from (C, H, W) to (H, W, C) for plotting
```

```
    img = img * std + mean
    img = np.clip(img, 0, 1)
    return img
```

```
dataiter = iter(validloader)
data = next(dataiter)
images = data[0].to(device)
labels = data[1].to(device)

ff_net = FeedForwardNet(num_classes=10).to(device)
ff_net.load_state_dict(torch.load(PATH, weights_only=True))

outputs = ff_net(images)

_, predicted = torch.max(outputs, 1)

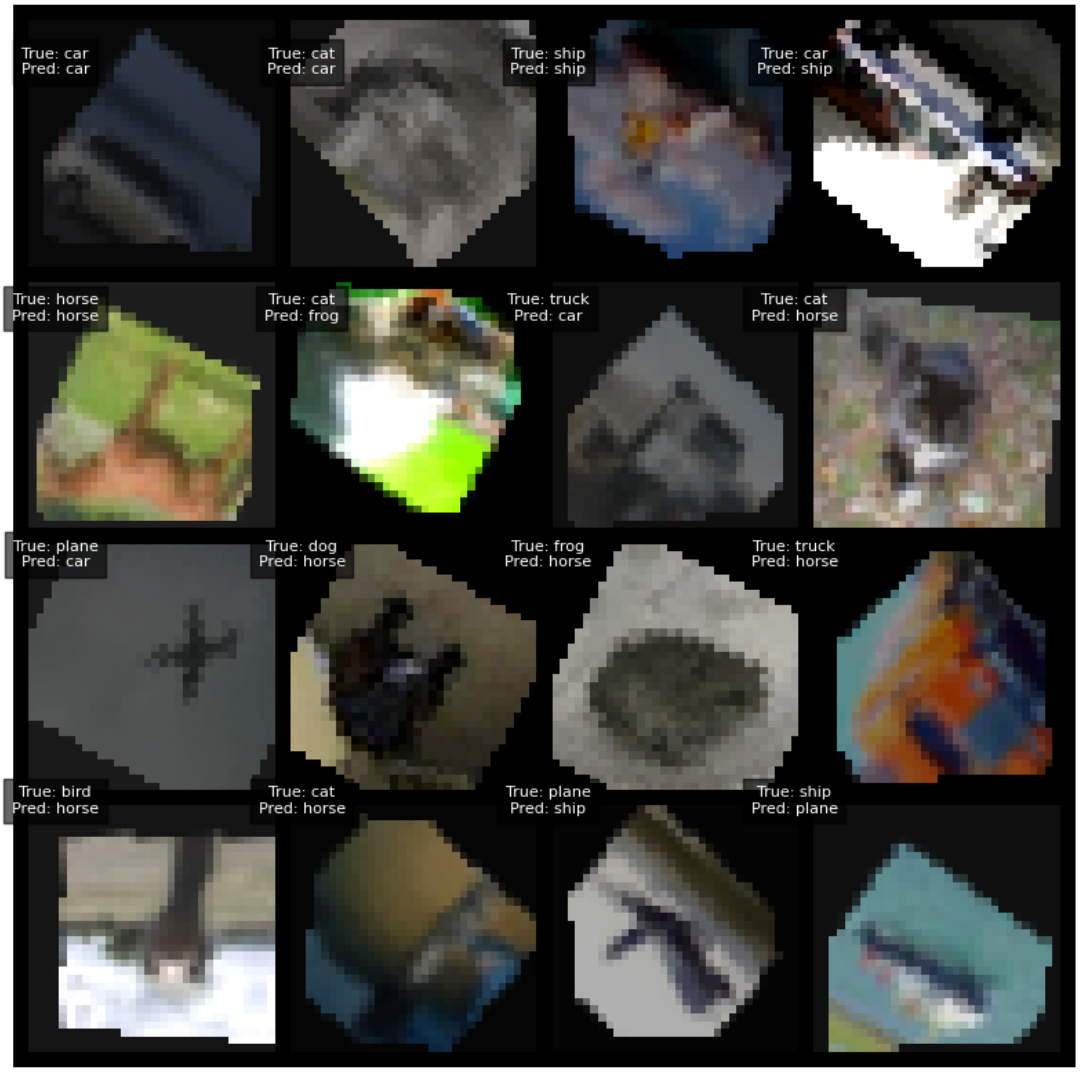
grid_img = torchvision.utils.make_grid(images.cpu(), nrow=4, padding=2, normalize=True)

plt.figure(figsize=(10, 10))

plt.imshow(np.transpose(grid_img.numpy(), (1, 2, 0)))

for i in range(len(images)):
    plt.text(i % 4 * 32 + 5, (i // 4) * 32 + 5, f'True: {classes[labels[i]]}\nPred: {classes[predicted[i]]}',
             color="white", fontsize=8, ha="center", va="top", bbox=dict(facecolor='black', alpha=0.6))

plt.axis('off')
plt.show()
```



Accuracy on Test set

```
ff_net.eval()
test_loss = 0.0
correct = 0
total = 0
with torch.no_grad():
    for inputs, labels in testloader:
        inputs, labels = inputs.to(device), labels.to(device)
        outputs = ff_net(inputs)
        loss = criterion(outputs, labels)
        test_loss += loss.item()
        _, preds = torch.max(outputs, 1)
        correct += (preds == labels).sum().item()
        total += labels.size(0)

test_loss /= len(testloader)
accuracy = correct / total
print(f"Feedforward NN Test Loss: {test_loss:.4f}, Test Accuracy: {accuracy:.4f}")
```

```
Feedforward NN Test Loss: 2.1008, Test Accuracy: 0.2153
```

8.5.1. Data augmentation for classification

On a dataset the size of CIFAR-10 (50 000 training images over 10 classes) a convolutional network will quickly overfit without extra regularisation. Data augmentation is the cheapest regulariser available: it expands the effective training set by applying label-preserving transformations each time a sample is drawn. For natural images the transformations are simple: horizontal flip, small random crop with padding, small colour jitter. They do not change what the image is, but they force the network to learn representations that are invariant to these incidental variations.

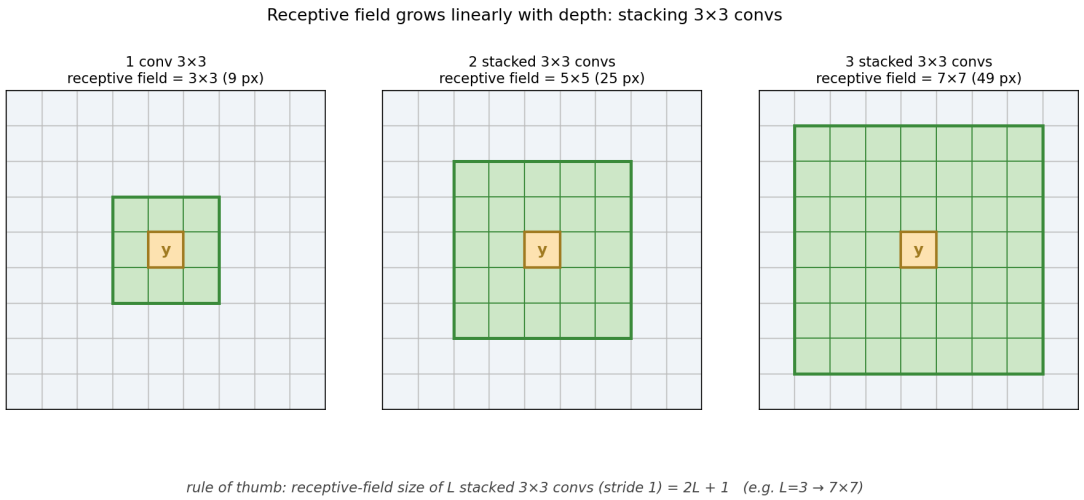
A second, less obvious benefit: augmentation smooths the loss landscape. Networks trained with augmentation generalise better even when the augmented training accuracy is lower, because the network can no longer memorise exact pixel patterns.

```python
# Augmentation pipeline used at training time only, validation and test
# data are fed through the deterministic transform from earlier in the notebook.
train_transform = transforms.Compose([
    transforms.RandomCrop(32, padding=4),        # shift by up to 4 pixels, pad with zeros
    transforms.RandomHorizontalFlip(),           # left-right flip with probability 0.5
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5, 0.5, 0.5), std=(0.5, 0.5, 0.5)),
])

# Preview a few augmented versions of the same underlying image so the
# transformation is visible.
import matplotlib.pyplot as plt
fig, axes = plt.subplots(1, 5, figsize=(12, 3))
raw = torchvision.datasets.CIFAR10(root='./data', train=True, download=False)
img, label = raw[0]
for ax in axes:
    ax.imshow(transforms.ToPILImage()(train_transform(img) * 0.5 + 0.5))
    ax.set_title(raw.classes[label])
    ax.axis('off')
plt.tight_layout()
plt.show()
```

8.5.2. Why CNN depth gives larger receptive fields

Each 3×3 convolution looks at a 3×3 window of its input. Stacking convolutions causes the **effective receptive field** in the original image to grow linearly with depth: a stack of L such layers (stride 1) sees a $(2L + 1) \times (2L + 1)$ region.



This is why deeper networks can model larger spatial structure even when each layer only uses small kernels.

8.6. Create and Train a Convolutional Neural Network (CNN)

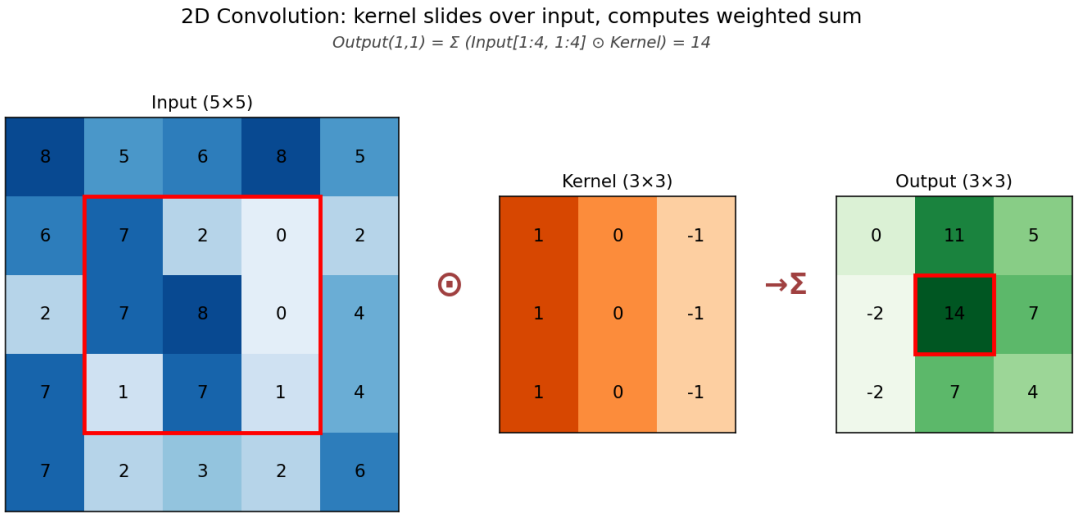
Convolution and pooling layers are fundamental operations for building CNN models. There are a number of parameters and if their definitions are not clear, it could lead to great confusion. - Parameters for convolution (pooling) layers - **stride:** how many "steps" that the filter makes for each advance - **kernel size**: how large is the kernel (filter) is - **number of filters (channels):** designates the "depth" of the data. Most image inputs have three filters (RGB) - **padding:** how to pad the input sample with zero in the border - How to calculate output size of convolution/pooling operation

(W - F + 2P)/S + 1 - *W*: input size - *F*: kernel size - *P*: padding - *S*: stride

Convolution 2D

The 2D convolution use 'out_channels' number kernel. This kernel tensor "slides" (naive implementation) over the 2D input data, performing an elementwise multiplication with the overlaped input and then summing up the results into a single output.

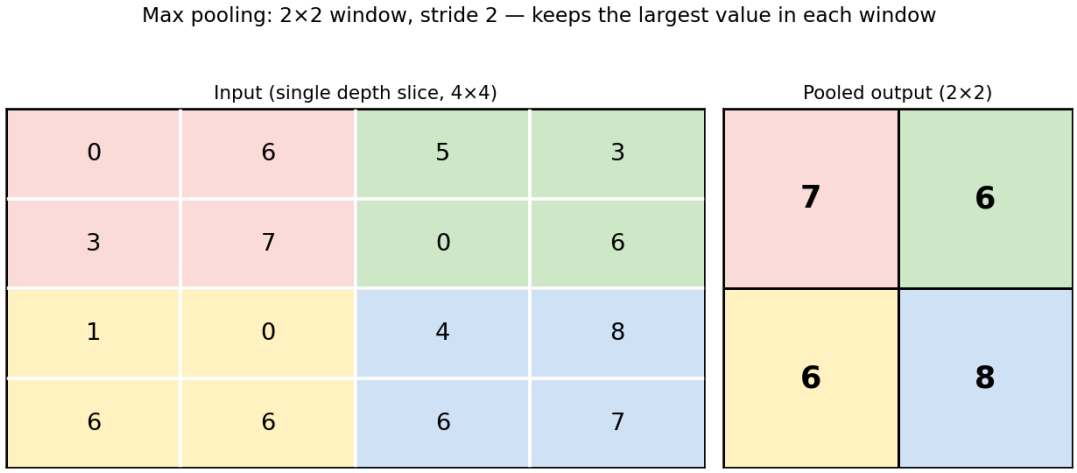
nn.Conv2d() - in_channels (int) - Number of channels in the input image - out_channels (int) - Number of channels produced by the convolution - kernel_size (int or tuple) - Size of the convolving kernel - stride (int or tuple, optional) - Stride of the convolution. Default: 1 - padding (int, tuple or str, optional) - Padding added to all four sides of the input. Special keywords 'same', or 'valid'. Default: 0 - padding_mode (str, optional) - 'zeros', 'reflect', 'replicate', 'circular'. Default: 'zeros' - dilation (int or tuple, optional) - Spacing between kernel elements. Default: 1 - groups (int, optional) - Number of blocked connections from input channels to output channels. Default: 1 - bias (bool, optional) - If True, adds a learnable bias to the output. Default: True



Max Pooling 2D

Convolution is not responsible for data compression, pooling layers are used to reduce the size of neural networks and speed up the process, by highlighting the most important parts.

nn.MaxPool2d() - kernel_size (int or tuple) - Size of the convolving kernel - stride (int or tuple, optional) - Stride of the convolution. Default: 1 - padding (int, tuple or str, optional) - Padding added to all four sides of the input. Default: 0



Let's test the pooling and convolutional operations on some examples (check input type requirements too)

```
input = torch.tensor([[[0,0,1],
                       [1,2,3],
                       [7,5,3],
                       [5,3,6]],

                      [[9,11,10],
                       [10,11,12],
                       [31,55,32],
                       [17,29,32]]], dtype=torch.float32, requires_grad=True)

print(input.shape)
print(input)
```

```
# layer = nn.MaxPool2d(kernel_size=2, stride=2, padding=1)
# layer = nn.AvgPool2d(kernel_size=2, stride=1, padding=1)
# layer = nn.Conv2d(in_channels=2, out_channels=4, kernel_size=5, stride=1, padding='same')
layer = nn.MaxPool2d(kernel_size = 3, stride = 3)
print()
print(layer(input).shape)
print(layer(input))
```

```
torch.Size([2, 4, 3])
tensor([[[ 0.,  0.,  1.],
         [ 1.,  2.,  3.],
         [ 7.,  5.,  3.],
         [ 5.,  3.,  6.]],

        [[ 9., 11., 10.],
         [10., 11., 12.],
         [31., 55., 32.],
         [17., 29., 32.]]], requires_grad=True)

torch.Size([2, 1, 1])
tensor([[[ 7.]],

        [[55.]]], grad_fn=<MaxPool2DWithIndicesBackward0>)
```

8.7. Model

Convolutional Neural Network (two convolutional blocks + fully connected block). The fully connected layers are optional and nowadays rarely used, but help with the flexible output size. Try different models and check how they influence accuracy and execution times.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class ConvNet(nn.Module):
    def __init__(self, num_classes=10):
        super(ConvNet, self).__init__()

        # First convolutional layer with BatchNorm and Dropout
        self.layer1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=5, stride=1, padding='same'),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout(p=0.25)
        )

        # Second convolutional layer with BatchNorm and Dropout
        self.layer2 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5, stride=1, padding='same'),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout(p=0.25)
        )

        # Third convolutional layer with BatchNorm and Dropout
```

```
        self.layer3 = nn.Sequential(
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, stride=1, padding='same'),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout(p=0.25)
        )

        # Fully connected layers with Dropout
        self.fc1 = nn.Sequential(
            nn.Linear(128 * 4 * 4, 256),
            nn.ReLU(),
            nn.Dropout(p=0.5)
        )

        self.fc2 = nn.Sequential(
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(p=0.25)
        )
        self.fc3 = nn.Sequential(
            nn.Linear(128, num_classes),
            nn.ReLU()
        )

    def forward(self, x):
        out = self.layer1(x)
        out = self.layer2(out)
        out = self.layer3(out)
        out = torch.flatten(out, 1)  # flatten all dimensions except batch
        out = self.fc1(out)
        out = self.fc2(out)
        out = self.fc3(out)
        return out
```

How can **we decide the number of layers, kernel size, etc.**?

If we understand how the network works and have a clear understanding of the task and data, we can initialize a suitable network. If the problem is common, we can use similar pre-trained models as a starting point. Otherwise, it is recommended to conduct tests, analyze the results, and make modifications based on expert knowledge or perform a **hyperparameter search**.

8.7.1. Let's create the neural network, set up the device and get a simple vizualisation

```
convnet = ConvNet(10).to(device)
convnet
```

```
ConvNet(
  (layer1): Sequential(
    (0): Conv2d(3, 32, kernel_size=(5, 5), stride=(1, 1), padding=same)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (4): Dropout(p=0.25, inplace=False)
  )
```

```
  (layer2): Sequential(
    (0): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1), padding=same)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (4): Dropout(p=0.25, inplace=False)
  )
  (layer3): Sequential(
    (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=same)
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (4): Dropout(p=0.25, inplace=False)
  )
  (fc1): Sequential(
    (0): Linear(in_features=2048, out_features=256, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
  )
  (fc2): Sequential(
    (0): Linear(in_features=256, out_features=128, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.25, inplace=False)
  )
  (fc3): Sequential(
    (0): Linear(in_features=128, out_features=10, bias=True)
    (1): ReLU()
  )
)
```

```
summary(convnet, (3, 32, 32))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 32, 32, 32]           2,432
       BatchNorm2d-2           [-1, 32, 32, 32]              64
              ReLU-3           [-1, 32, 32, 32]               0
         MaxPool2d-4           [-1, 32, 16, 16]               0
           Dropout-5           [-1, 32, 16, 16]               0
            Conv2d-6           [-1, 64, 16, 16]          51,264
       BatchNorm2d-7           [-1, 64, 16, 16]             128
              ReLU-8           [-1, 64, 16, 16]               0
         MaxPool2d-9             [-1, 64, 8, 8]               0
          Dropout-10             [-1, 64, 8, 8]               0
           Conv2d-11            [-1, 128, 8, 8]          73,856
      BatchNorm2d-12            [-1, 128, 8, 8]             256
             ReLU-13            [-1, 128, 8, 8]               0
        MaxPool2d-14            [-1, 128, 4, 4]               0
          Dropout-15            [-1, 128, 4, 4]               0
           Linear-16                  [-1, 256]         524,544
```

```
              ReLU-17                  [-1, 256]               0
           Dropout-18                  [-1, 256]               0
            Linear-19                  [-1, 128]          32,896
              ReLU-20                  [-1, 128]               0
           Dropout-21                  [-1, 128]               0
            Linear-22                   [-1, 10]           1,290
              ReLU-23                   [-1, 10]               0
================================================================
Total params: 686,730
Trainable params: 686,730
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.01
Forward/backward pass size (MB): 1.54
Params size (MB): 2.62
Estimated Total Size (MB): 4.17
----------------------------------------------------------------
```

8.8. Visualisation with Tensorboard

Tensorboard has nice visualization features

- SummaryWriter('runs/CIFAR10') - specifying the log directory, default is "runs"
- read one or more batches with iter() and next()
- visualize the batch in a grid view
- writer.add_image('CIFAR10_batch', img_grid) - show img_grid in Tensorboard as 'CIFAR10_batch'

```
from torch.utils.tensorboard import SummaryWriter

# default `log_dir` is "runs" - we'll be more specific here
writer = SummaryWriter('runs/CIFAR10')

# get some random training images
dataiter = iter(trainloader)
images, labels = next(dataiter)
images = images / 2 + 0.5

# create grid of images
img_grid = torchvision.utils.make_grid(images)

# write to tensorboard
writer.add_image('CIFAR10_batch', img_grid)
```

```
%load_ext tensorboard
```

```
# Set up a logs directory, so Tensorboard knows where to look for files.
%tensorboard --logdir="runs" --host localhost  # visit http://localhost:6006 for a larger view (not for colab)
```

```
writer.add_graph(convnet, images.to(device)) # refresh the tensorboard to see the changes
writer.close()
```

8.9. Loss function

```
criteria = nn.CrossEntropyLoss()
```

8.9.1. Optimizers and their options

```
optimizer = optim.SGD(convnet.parameters(), lr=0.001, momentum=0.9)
#optimizer = optim.Adam(convnet.parameters(), lr=0.01, betas=[0.9, 0.99])
```

8.10. Train

```
n_epochs = 10
hist_train=[]
hist_valid=[]
best_loss=float('inf')
best_model_wts = copy.deepcopy(convnet.state_dict())
early_stop_tolerant_count=0
early_stop_tolerant=10
```

```
for epoch in range(n_epochs):  # loop over the dataset multiple times
    train_loss = 0.0
    convnet.train()
    for inputs, labels in trainloader:
        inputs, labels = inputs.to(device), labels.to(device)

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + loss + backward + optimize
        outputs = convnet(inputs)
        loss = criteria(outputs, labels)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()

    train_loss /= len(trainloader)  # Divide by number of batches
    hist_train.append(train_loss)

    # Validation
    convnet.eval()
    valid_loss = 0.0
    with torch.no_grad():
        for inputs, labels in validloader:
            inputs, labels = inputs.to(device), labels.to(device)
            outputs = convnet(inputs)
            loss = criteria(outputs, labels)
            valid_loss += loss.item()

    valid_loss /= len(validloader)  # Divide by number of batches
    hist_valid.append(valid_loss)

    early_stop_tolerant_count += 1
    if valid_loss < best_loss:
        early_stop_tolerant_count = 0
        best_loss = valid_loss
        best_model_wts = copy.deepcopy(convnet.state_dict())

    if early_stop_tolerant_count >= early_stop_tolerant:
        break
```

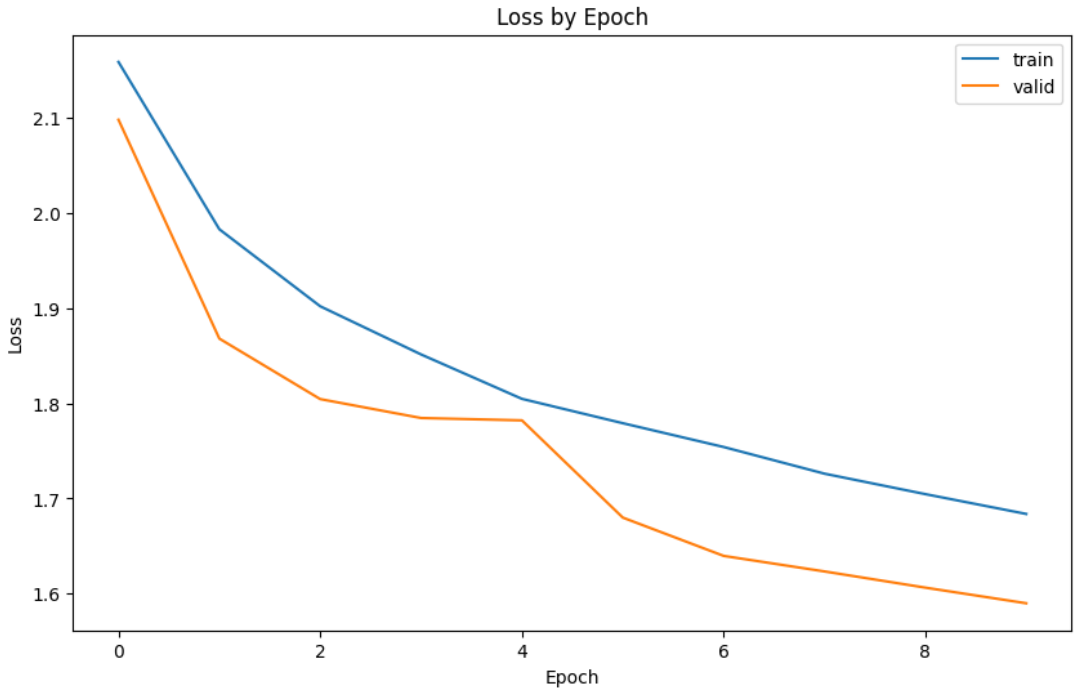
```
    print(f"Epoch {epoch:04d}: train loss {train_loss:.4f}, valid loss {valid_loss:.4f}")

print('Finished Training')
```

```
Epoch 0000: train loss 2.1591, valid loss 2.0984
Epoch 0001: train loss 1.9832, valid loss 1.8682
Epoch 0002: train loss 1.9022, valid loss 1.8045
Epoch 0003: train loss 1.8515, valid loss 1.7846
Epoch 0004: train loss 1.8048, valid loss 1.7822
Epoch 0005: train loss 1.7792, valid loss 1.6799
Epoch 0006: train loss 1.7541, valid loss 1.6396
Epoch 0007: train loss 1.7261, valid loss 1.6233
Epoch 0008: train loss 1.7046, valid loss 1.6062
Epoch 0009: train loss 1.6837, valid loss 1.5898
Finished Training
```

```
plt.figure(figsize=(10,6))
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Loss by Epoch')

plt.plot(hist_train)
plt.plot(hist_valid)
#plt.ylim((1,3))
plt.legend(['train', 'valid'])
```



Save trained model

```
PATH = './convnet.pth'
torch.save(convnet.state_dict(), PATH)
```

Load saved model and predict some images

```
dataiter = iter(validloader)
data = next(dataiter) #dataiter.next()
images = data[0].to(device)
labels = data[1].to(device)

# Load the model and perform prediction
convnet = ConvNet().to(device)
convnet.load_state_dict(torch.load(PATH, weights_only=True))

# Forward pass through the model
outputs = convnet(images)

# Get predicted class indices
_, predicted = torch.max(outputs, 1)

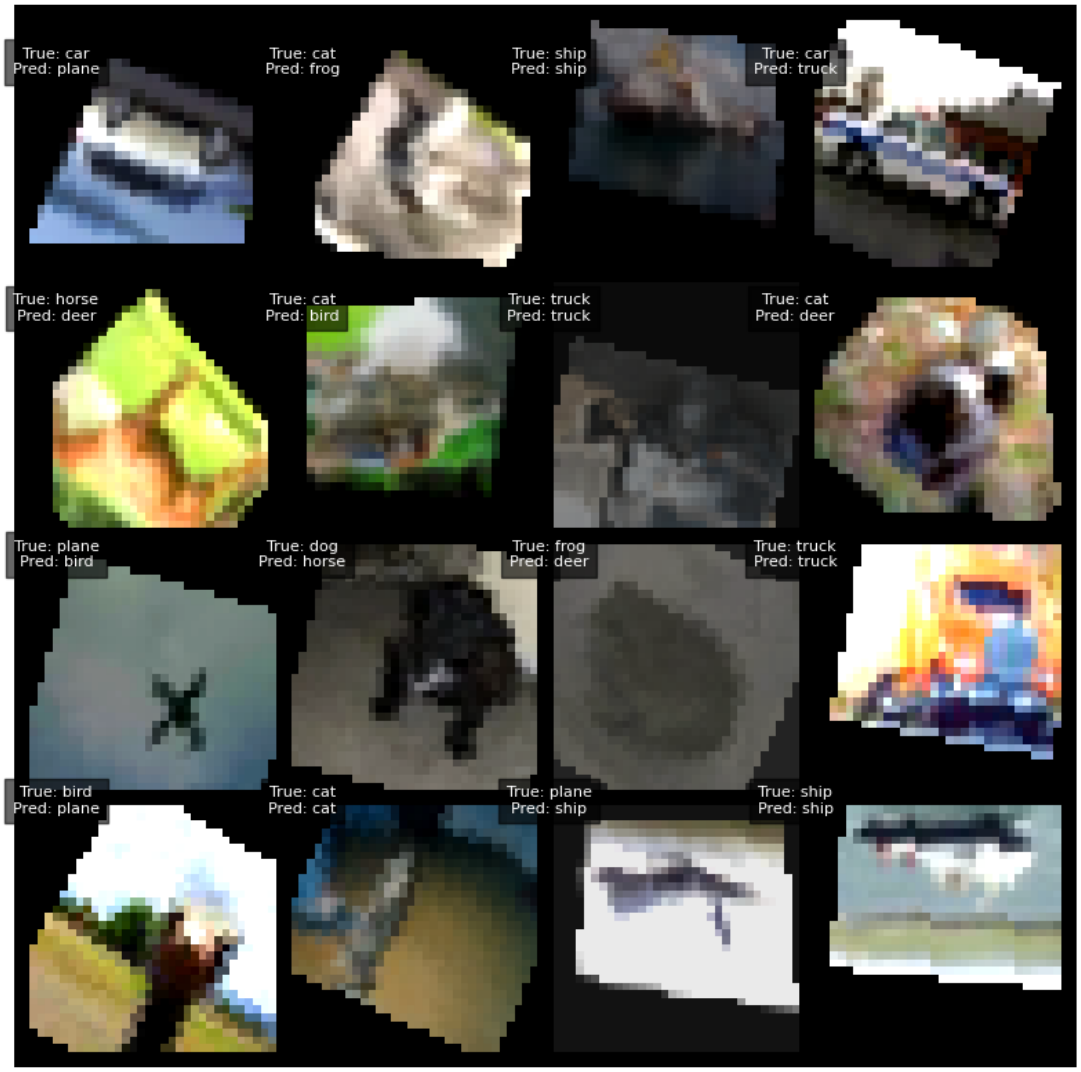
# Make a grid of images
grid_img = torchvision.utils.make_grid(images.cpu(), nrow=4, padding=2, normalize=True)

# Set up the plot
plt.figure(figsize=(10, 10))

# Show the image grid
plt.imshow(np.transpose(grid_img.numpy(), (1, 2, 0)))

# Add the ground truth and predicted labels to the images
for i in range(len(images)):
    plt.text(i % 4 * 32 + 5, (i // 4) * 32 + 5, f'True: {classes[labels[i]]}\nPred: {classes[predicted[i]]}',
             color="white", fontsize=8, ha="center", va="top", bbox=dict(facecolor='black', alpha=0.6))

plt.axis('off')
plt.show()
```



Accuracy on Test set

```
correct = 0
total = 0
labels_total=[]
prediction_total=[]
# no need of gradients
with torch.no_grad():
    for inputs, labels in testloader:
        #images, labels = data
        inputs, labels = inputs.to(device), labels.to(device) #set for GPU if available
        # calculate outputs by running images through the network
        outputs = convnet(inputs).to(device)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        labels_total=np.append(labels_total,labels.cpu().numpy(), axis=0)
        prediction_total=np.append(prediction_total,predicted.cpu().numpy(), axis=0)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
```

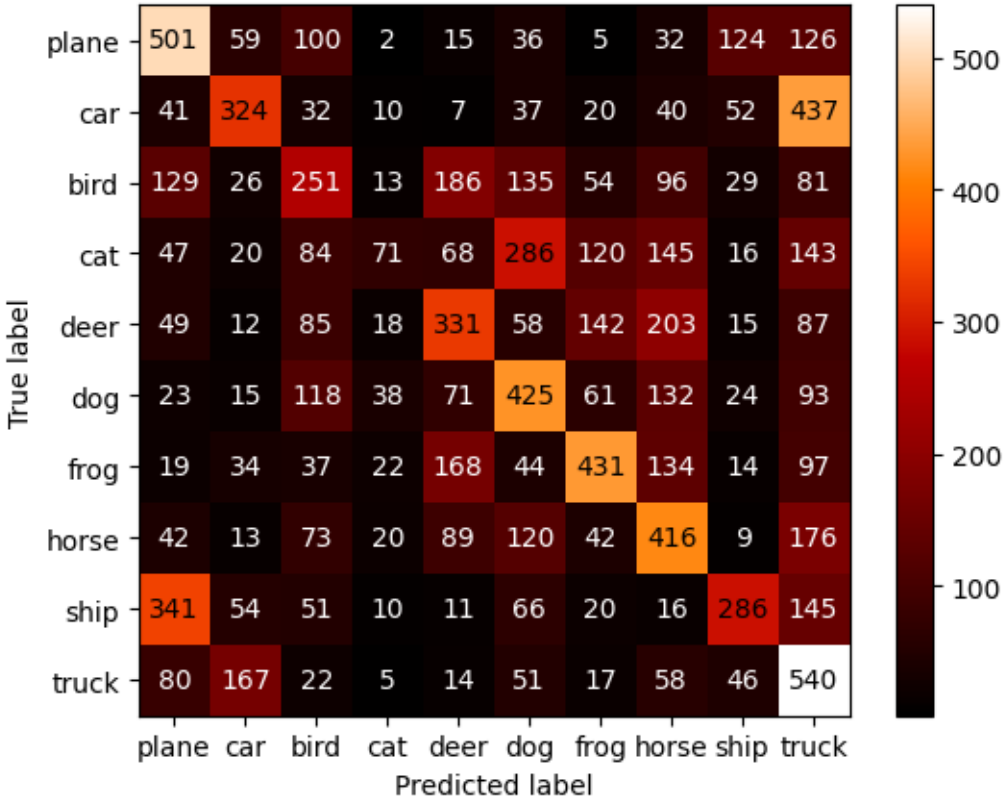
```
print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))
```

```
Accuracy of the network on the 10000 test images: 35 %
```

8.11. Confusion Matrix

color maps

```
#!pip install scikit-learn
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
cm=confusion_matrix(labels_total, prediction_total)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=classes)
disp = disp.plot(cmap="gist_heat")
#disp.plot()
```



Accuracy on test set per class

```
# prepare to count predictions for each class
correct_pred = {classname: 0 for classname in classes}
total_pred = {classname: 0 for classname in classes}
# again no gradients needed
with torch.no_grad():
    for data in validloader:
        images, labels = data
        #inputs, labels = data[0].to(device), data[1].to(device) #set for GPU if available
        outputs = convnet(images.to(device))
```

```
        _, predictions = torch.max(outputs, 1)
        # collect the correct predictions for each class
        for label, prediction in zip(labels, predictions):
            if label == prediction:
                correct_pred[classes[label]] += 1
            total_pred[classes[label]] += 1

# print accuracy for each class
for classname, correct_count in correct_pred.items():
    accuracy = 100 * float(correct_count) / total_pred[classname]
    print("Accuracy for class {:5s} is: {:.1f} %".format(classname,
                                                        accuracy))
```

```
Accuracy for class plane is: 42.9 %
Accuracy for class car   is: 40.1 %
Accuracy for class bird  is: 22.1 %
Accuracy for class cat   is: 8.0 %
Accuracy for class deer  is: 42.7 %
Accuracy for class dog   is: 42.5 %
Accuracy for class frog  is: 52.0 %
Accuracy for class horse is: 37.0 %
Accuracy for class ship  is: 32.0 %
Accuracy for class truck is: 48.7 %
```

8.12. Homework (Optional)

1. **Replacement architectures.** Replace the feed-forward net with a deeper variant (e.g. three hidden layers of 512-256-128) and the CNN with one extra convolutional block. Report the validation accuracy of each over 10 epochs and discuss which change paid off more per parameter added.
2. **Augmentation ablation.** Train the same CNN three times: without augmentation, with crop+flip only, and with crop+flip+colour jitter. Plot validation accuracy vs. epoch for all three runs on the same figure. Explain the gap in 1-2 sentences.
3. **Learning-rate schedule.** Replace the plain SGD with `torch.optim.SGD` combined with `torch.optim.lr_scheduler.CosineAnnealingLR`. Train for 30 epochs and compare the final accuracy with the constant-LR baseline.
4. **Confusion-matrix insight.** After training a model, identify the two most-confused class pairs from the confusion matrix. For each pair, show five sample images that were misclassified and speculate on the visual reason.
5. **Deploy a single prediction.** Save your trained CNN with `torch.save(model.state_dict(), ...)`, reload it in a fresh cell, and run it on one new image (either a CIFAR-10 test image or an image you upload). Display the top-3 predicted classes with their softmax probabilities.
6. **Model size vs. accuracy.** Measure the parameter count of your best model with `sum(p.numel() for p in model.parameters())`. Plot accuracy vs. parameter count for three checkpoints (early, mid, late training) and relate this to the bias-variance trade-off.

Chapter 9

Spectrogram Classification with CNNs

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1DAx-TA67ndHJihb8OQUGpsd31MTlbNUN?usp=sharing

Sound can be represented digitally in various ways, most commonly by encoding the temporal variations of the sound wave. When analyzing audio, such as music, a **visual representation** is often more useful and accessible. One effective way to achieve this is by using **spectrograms**, which visually depict sound. Examples of spectrograms can be seen here.

A **spectrogram** is generated from the sound wave using **Fourier transformation**, more details about it can be found here.

The choice of representation significantly impacts tasks such as **music classification, speech recognition, and sound analysis**. Below, we outline five key representations and their applications.

9.0.1. 1. Raw Waveform

The most direct representation, storing the **amplitude of the sound wave over time**.

- **X-axis:** Time
- **Y-axis:** Amplitude

Applications:

- Used in **deep learning-based** approaches (e.g., WaveNet, SampleRNN).
- Retains all information but is high-dimensional and challenging for direct analysis.

9.0.2. 2. Mel-Frequency Cepstral Coefficients (MFCC)

MFCCs transform the frequency domain into **features that mimic human auditory perception**.

- Extracted through **Fourier Transform**, **Mel-scale filter banks**, and **Discrete Cosine Transform (DCT)**.
- Reduces data complexity while preserving key perceptual features.

Applications:

- Widely used in **speech and speaker recognition**.
- Compact representation but sensitive to noise.

9.0.3. 3. Mel Spectrogram

A time-frequency representation where **frequencies are mapped to the Mel scale**.

- **X-axis:** Time
- **Y-axis:** Frequency (Mel scale)
- **Color intensity:** Amplitude (power)

Applications:

- Effective for **music genre classification** and **sound event detection**.
- Captures both temporal and frequency characteristics but is computationally expensive.

9.0.4. 4. Tempogram

Encodes **tempo and rhythmic patterns** by analyzing changes in beat structure.

- **X-axis:** Time
- **Y-axis:** Tempo (beats per minute - BPM)

Applications:

- Used in **beat tracking** and **genre classification**.
- Effective for distinguishing rhythmic structures but lacks spectral information.

9.0.5. 5. Chromogram

Represents the **harmonic content** of a signal by mapping energy to **pitch classes**.

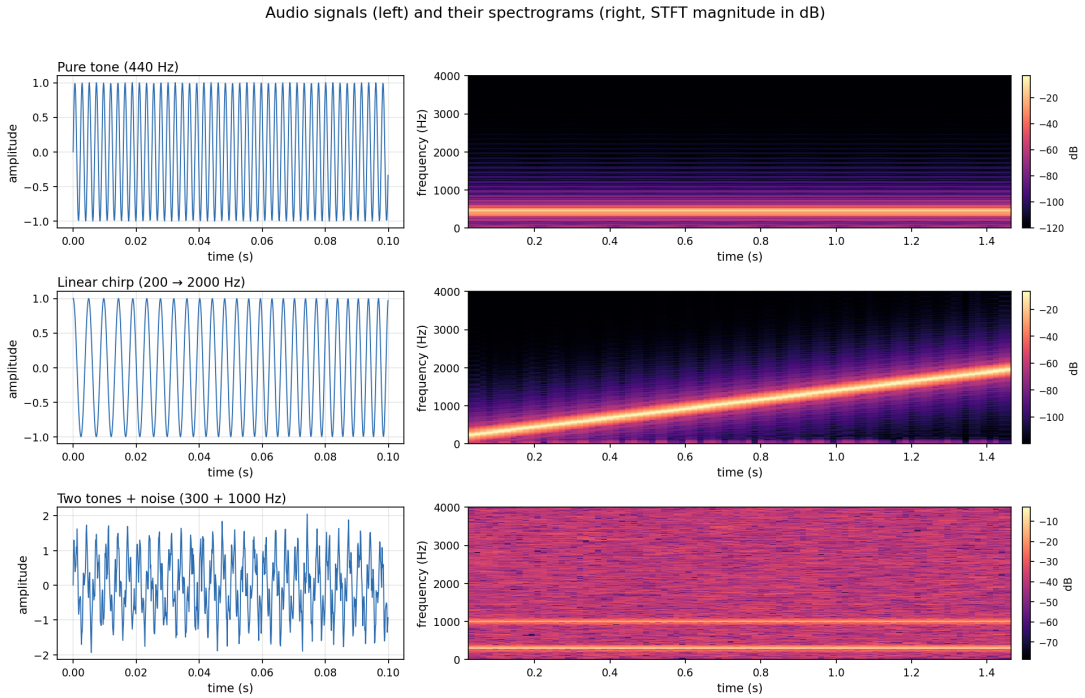
- **X-axis:** Time
- **Y-axis:** Pitch classes (C, C#, D, etc.)

Applications:

- Useful for **chord recognition**, **melody extraction**, and **key detection**.
- Focuses on harmonic structure but does not capture rhythm.

9.0.6. Comparison of Representations

Representation	Captures	Best for	Example Use Case
Raw Waveform	Full signal	General audio modeling	Speech synthesis, music generation
MFCC	Perceived frequency	Speech recognition, speaker ID	Voice assistants, speech-to-text
Mel Spectrogram	Frequency over time	Music classification, event detection	Genre classification
Tempogram	Rhythmic structure	Beat tracking, tempo analysis	Music information retrieval
Chromogram	Harmonic structure	Chord recognition, key detection	Music transcription



For this task, we will use the **GTZAN [4] dataset**, which contains **music files**, their **corresponding spectrograms**, and **pre-extracted numerical features**. The compressed GTZAN dataset can be downloaded here.

This dataset consists of **10 different music genres**, each with audio samples: - `blues` - `classical` - `country` - `disco` - `hiphop` - `jazz` - `metal` - `pop` - `reggae` - `rock`

For each genre, the dataset provides: - **Spectrogram representations** - **Precomputed audio feature metrics** relevant to music analysis

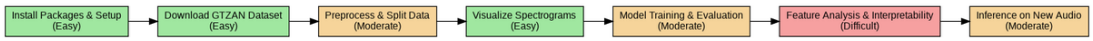
In this project, we will focus **exclusively on image-based data**, the **spectrograms**, to classify music genres.
We will train **classification models** using these spectrogram images.

9.1. Necessary Imports

```
!sudo apt-get install -y ffmpeg --quiet
!pip install librosa --quiet
!pip install imageio --quiet
!pip install imageio-ffmpeg --quiet
!pip install captum --quiet
```

```
import os
import glob
import imageio
import random, shutil
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.datasets as datasets
import torchvision.transforms as transforms
import numpy as np
import matplotlib.pyplot as plt
import IPython.display as display
import librosa
import librosa.display
import matplotlib
import seaborn as sns
from IPython.display import Image as IMG
from PIL import Image
from tqdm.notebook import tqdm
from torch.utils.data import DataLoader, Subset
from sklearn.model_selection import train_test_split
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.manifold import TSNE
from captum.attr import IntegratedGradients

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```



9.1.1. Downloading the GTZAN Dataset

The **GTZAN dataset**, which we will use for **music genre classification**, can be downloaded in compressed form from the following link: Dataset Link.

By running the code below, the dataset will be **automatically downloaded and extracted** into a folder named **"data"**. Inside this folder, the **"images_original"** directory contains the spectrogram images, organized into separate subfolders by genre.

```
%%bash
if [ -d "data" ]; then
    echo "Data folder already exists. Skipping download and extraction."
else
    echo "Data folder does not exist. Downloading and extracting..."
```

```
    wget https://osf.io/drjhb/download/ -O music.zip --no-check-certificate --quiet
    unzip -q music.zip -d data
    mv data/Data/* data/
    rm -rf data/Data
    rm music.zip
fi
```

```
Data folder does not exist. Downloading and extracting...
```

9.1.2. Loading the Data

As a first step, we will load and visualize an audio file. From this, we will generate its corresponding **Mel-scaled spectrogram**, a process described in more detail here, and compare it with the precomputed spectrogram provided in the dataset.

```
sample_path_jazz = 'data/genres_original/jazz/jazz.00011.wav'
display.Audio(sample_path_jazz)
```

```
sample_path_metal = 'data/genres_original/metal/metal.00008.wav'
display.Audio(sample_path_metal)
```

```
sample_path_reggae = 'data/genres_original/reggae/reggae.00007.wav'
display.Audio(sample_path_reggae)
```

9.2. Looking at Different Representations

```
# 'y' is a NumPy array containing the audio samples (amplitude values over time).
# 'sample_rate' is an integer representing the number of samples per second (in Hz).
y, sample_rate = librosa.load(sample_path_jazz)

print('y:', y)
# 'y' contains the amplitude values of the audio waveform.

print('y shape:', np.shape(y))
# The shape tells you the number of samples. For mono audio, it's (num_samples,).

print('Sample rate (Hz):', sample_rate)
# Note: sample_rate is in Hz (e.g., 22050 Hz means 22.05 KHz).
# If you want the value in KHz, you could divide it by 1000.

print(f'Length of audio (s): {np.shape(y)[0]/sample_rate}')
# The duration (in seconds) is computed by dividing the total number of samples by the sample rate.
```

```
y: [0.09072876 0.13885498 0.08273315 ... 0.16775513 0.17504883 0.16122437]
y shape: (661794,)
Sample rate (Hz): 22050
Length of audio (s): 30.013333333333332
```

```
def plot_music_representations(y, sample_rate):
    melspec = librosa.feature.melspectrogram(
        y=y,                  # Audio time series.
        sr=sample_rate,       # Sampling rate of the audio.
        n_fft=2048,           # Number of samples per FFT window; larger value gives better frequency
  resolution.
        hop_length=512,       # Number of audio samples between successive FFT frames; controls overlap
  between frames.
```

```
    n_mels=128            # Number of Mel bands to generate; determines the resolution in the Mel scale.
)

melspec_db = librosa.power_to_db(
    melspec,            # Input power spectrogram.
    ref=np.max          # Reference value for scaling; here using the maximum value in the spectrogram.
)

mfcc = librosa.feature.mfcc(
    y=y,                  # Audio time series.
    sr=sample_rate,       # Sampling rate of the audio.
    n_fft=2048,           # FFT window size used to compute the underlying spectrogram.
    hop_length=512,       # Number of samples between successive frames.
    n_mels=128            # Number of Mel bands used; influences the number of MFCC coefficients computed.
)

oenv = librosa.onset.onset_strength(
    y=y,                  # Audio time series.
    sr=sample_rate,       # Sampling rate of the audio.
    hop_length=512        # Number of samples between frames for computing the onset envelope.
)

tempogram = librosa.feature.tempogram(
    onset_envelope=oenv,  # Precomputed onset strength envelope.
    sr=sample_rate,       # Sampling rate; used to convert lag bins to tempo (beats per minute).
    hop_length=512,       # Hop length; should match that used for the onset envelope.
    norm=None             # Normalization method for the tempogram; None means no normalization is applied.
)

chromagram = librosa.feature.chroma_stft(
    y=y,                  # Audio time series.
    sr=sample_rate,       # Sampling rate of the audio.
    n_fft=2048,           # FFT window size for the STFT.
    hop_length=512,       # Number of samples between successive frames.
    n_chroma=24           # Number of chroma bins; represents the resolution for the 12 pitch classes
(here, 24 provides a finer resolution).
)

fig, axes = plt.subplots(nrows=5, ncols=1, figsize=(12, 11))

# Plot the waveform
librosa.display.waveshow(y=y, sr=sample_rate, ax=axes[0])
axes[0].set_title("Waveform", fontsize=20)

# Plot MFCC
img1 = librosa.display.specshow(mfcc, sr=sample_rate, ax=axes[1], x_axis='time', y_axis='mel')
axes[1].set_title("MFCC", fontsize=20)
fig.colorbar(img1, ax=axes[1], format="%+2.f dB")

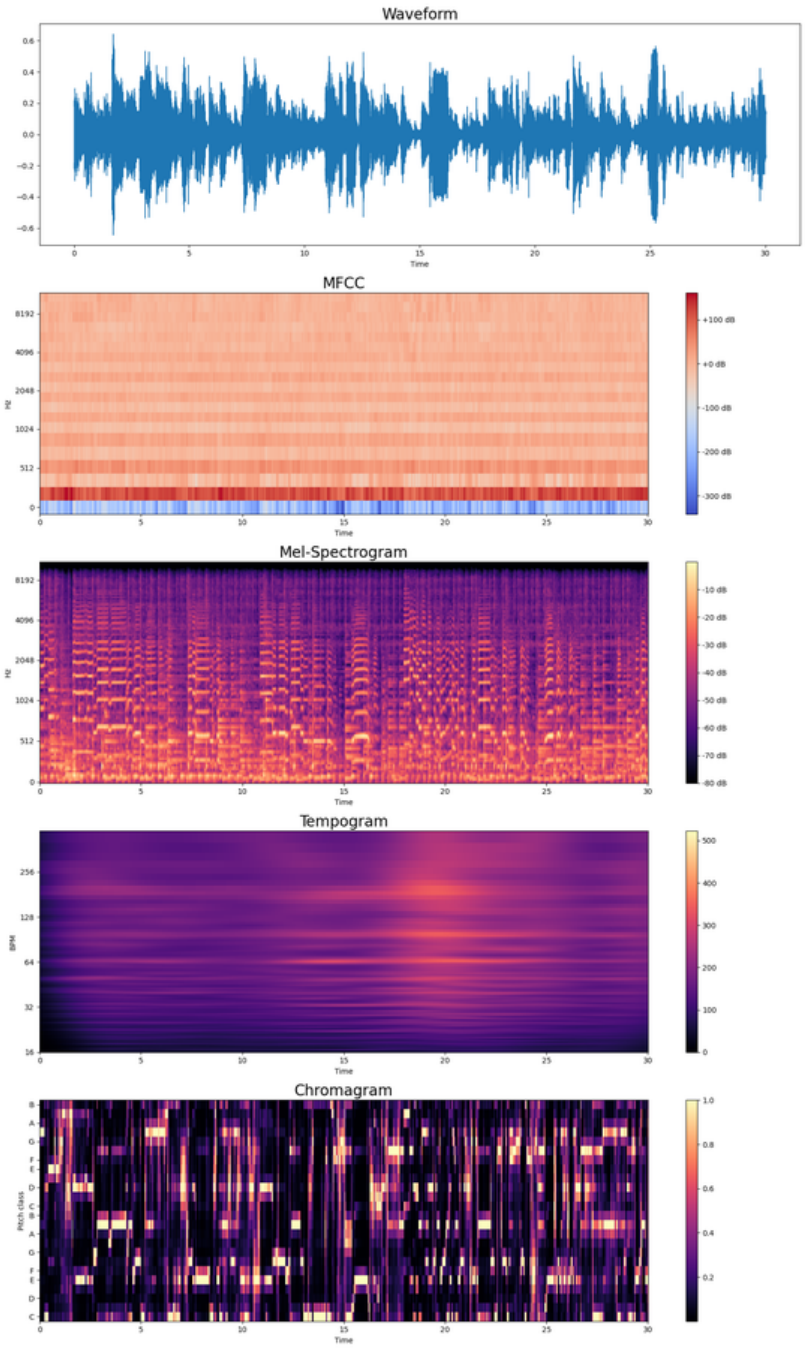
# Plot Mel-Spectrogram
img2 = librosa.display.specshow(melspec_db, sr=sample_rate, ax=axes[2], x_axis='time', y_axis='mel')
axes[2].set_title("Mel-Spectrogram", fontsize=20)
fig.colorbar(img2, ax=axes[2], format="%+2.f dB")

# Plot Tempogram
img3 = librosa.display.specshow(tempogram, sr=sample_rate, hop_length=512, ax=axes[3],
                                x_axis='time', y_axis='tempo')
axes[3].set_title("Tempogram", fontsize=20)
fig.colorbar(img3, ax=axes[3])
```

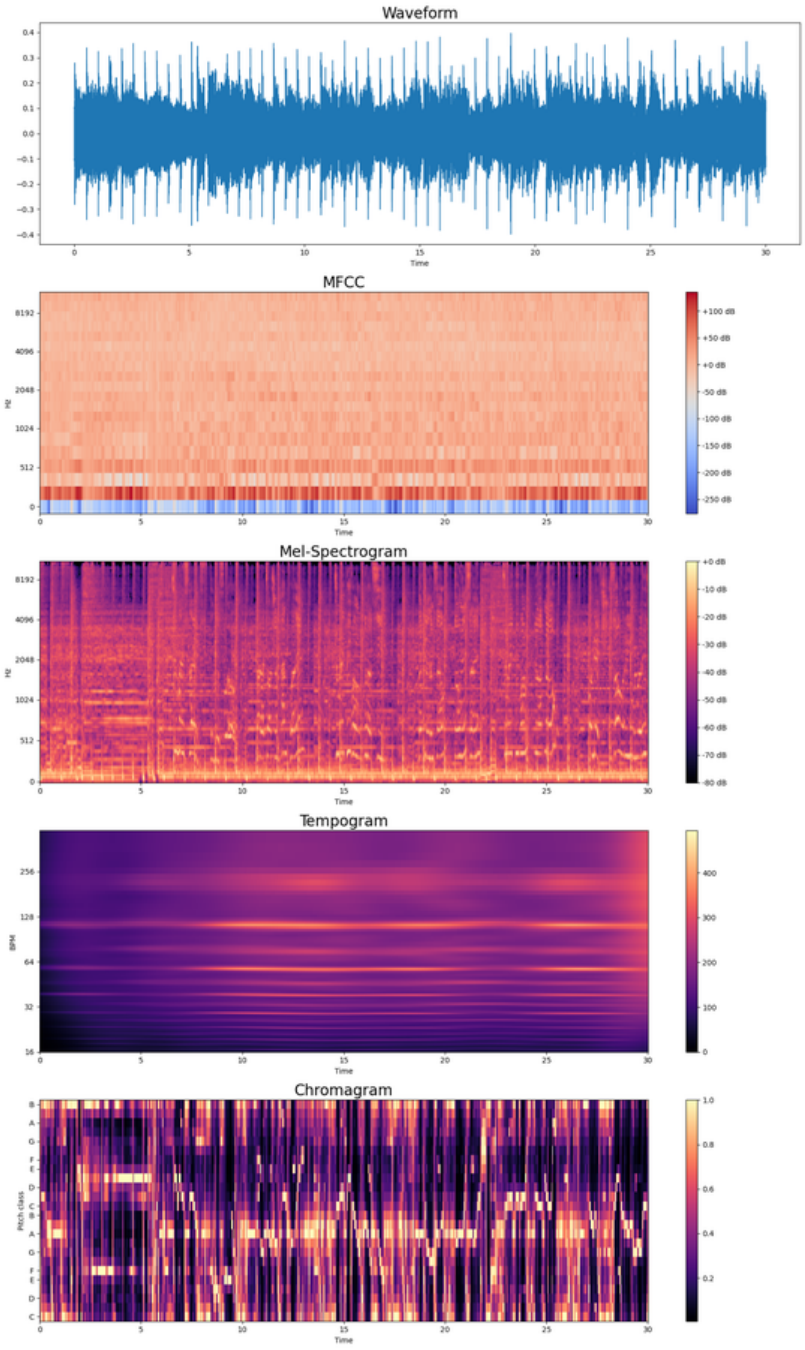
```
    # Plot Chromagram
    img4 = librosa.display.specshow(chromagram, sr=sample_rate, hop_length=512, ax=axes[4],
                                    x_axis='time', y_axis='chroma')
    axes[4].set_title("Chromagram", fontsize=20)
    fig.colorbar(img4, ax=axes[4])

    plt.tight_layout()
    plt.show()
```

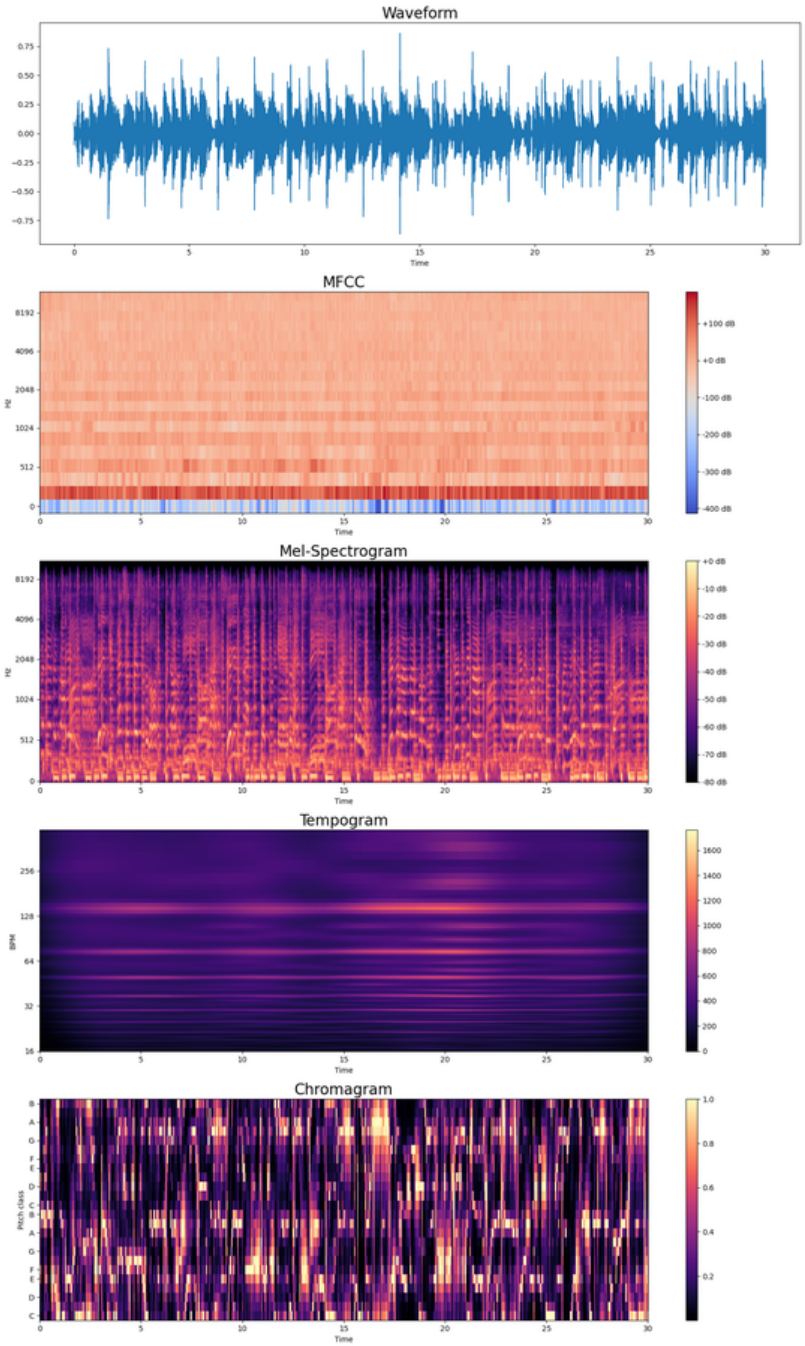
```
plot_music_representations(y, sample_rate)
```



```
y, sample_rate = librosa.load(sample_path_metal)
plot_music_representations(y, sample_rate)
```




```
y, sample_rate = librosa.load(sample_path_reggae)
plot_music_representations(y, sample_rate)
```



9.3. Sticking with Short-Time Fourier Transform

```
# - n_fft=2048: Size of FFT window, which determines the frequency resolution.
# - hop_length=512: Number of samples between successive frames, affecting time resolution.
# Taking the absolute value gives the magnitude spectrum (ignoring phase).
D = np.abs(librosa.stft(y, n_fft=2048, hop_length=512))
print('Shape of D object:', np.shape(D))  # (frequency bins x time frames).

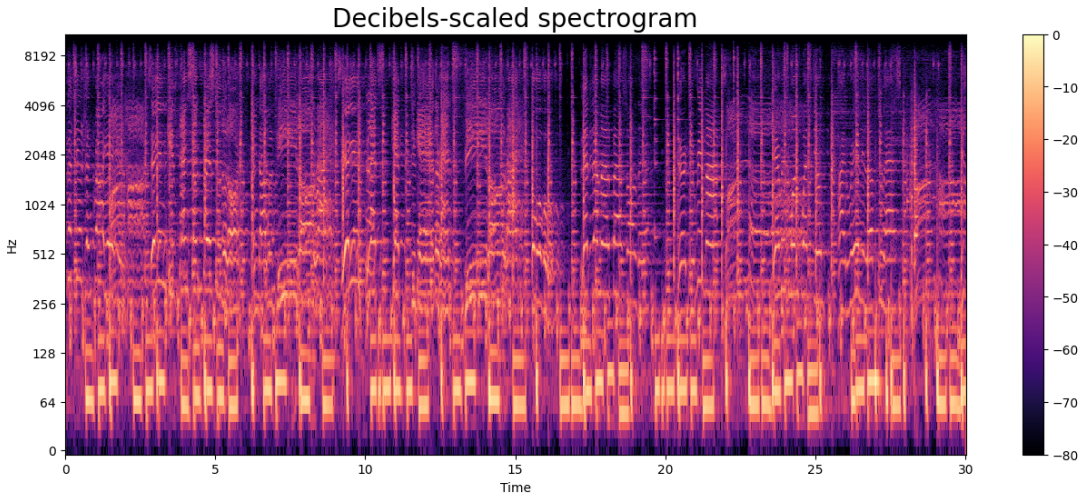
# Convert the amplitude spectrogram to a decibel (dB) scale.
# - ref=np.max: The reference value is the maximum amplitude in D, setting that value to 0 dB.
```

```
# This logarithmic conversion is useful because human perception of loudness is logarithmic.
DB = librosa.amplitude_to_db(D, ref=np.max)

plt.figure(figsize=(16, 6))
librosa.display.specshow(DB, sr=sample_rate, hop_length=512, x_axis='time', y_axis='log')

plt.colorbar()
plt.title('Decibels-scaled spectrogram', fontsize=20)
plt.show()
```

```
Shape of D object: (1025, 1293)
```



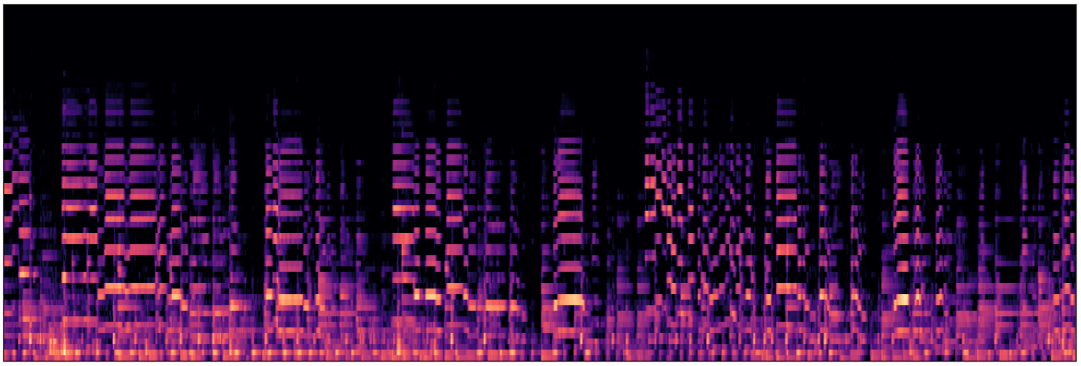
A **Mel-scaled spectrogram** uses the **Mel scale** instead of a linear frequency scale: read more here.

In short, the **Mel scale** is a **perceptual scale of pitches**, designed to match how humans perceive sound. Unlike the linear frequency scale, the Mel scale spaces frequencies in a way that aligns more closely with human hearing.

```
y, sr = librosa.load(sample_path_jazz)

S = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=64)
S_DB = librosa.amplitude_to_db(S, ref=np.max)
plt.figure(figsize=(15, 5))
librosa.display.specshow(S_DB, sr=sr, hop_length=512)

plt.show()
```



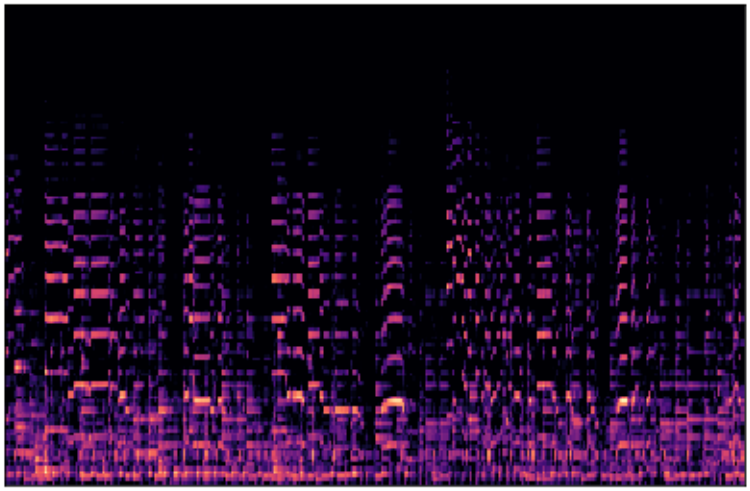
9.4. Looking at our Mel Spectrograms in the Dataset

```
img_path = 'data/images_original/jazz/jazz00011.png'
img = imageio.v2.imread(img_path)
print(img.shape)

plt.imshow(img, interpolation='nearest')
plt.axis('off')

plt.gca().spines['top'].set_visible(False)
plt.gca().spines['right'].set_visible(False)
plt.gca().spines['bottom'].set_visible(False)
plt.gca().spines['left'].set_visible(False)
plt.savefig("./test_image.png", bbox_inches='tight', pad_inches=0)
plt.show()
```

```
(288, 432, 4)
```



Note: If we examine our images closely, we can see that they have a **white pixel border**. This could potentially complicate training for classification models. However, for the purposes of this task, we will ignore this issue and proceed without addressing it.

9.4.1. Creating Datasets and Dataloaders

Before loading the spectrogram images, it is good practice to normalize them using `PyTorch transforms`. Unlike standard image datasets, spectrograms **do not have predefined mean and standard deviation**

values for each channel. Therefore, we will first load the entire dataset, compute these values manually, and then use them to normalize the data consistently.

```
temp_transform = transforms.Compose([
    transforms.ToTensor()
])
temp_dataset = datasets.ImageFolder(root="data/images_original", transform=temp_transform)
loader = DataLoader(temp_dataset, batch_size=64, shuffle=False, num_workers=2)

mean = 0.0
std = 0.0
nb_samples = 0

for data, _ in loader:
    batch_samples = data.size(0)
    data = data.view(batch_samples, data.size(1), -1)

    mean += data.mean(2).sum(0)
    std += data.std(2).sum(0)
    nb_samples += batch_samples

mean /= nb_samples
std /= nb_samples

print("Computed Mean per channel:", mean)
print("Computed Std per channel:", std)
```

```
Computed Mean per channel: tensor([0.4994, 0.4420, 0.5124])
Computed Std per channel: tensor([0.4564, 0.4717, 0.4318])
```

After loading the temporary dataset with preliminary transforms, we can now **split it into training, validation, and test sets**. For this task, we use an **80% training, 10% validation, and 10% test** split.

The `DataLoader` in PyTorch is a powerful tool designed to handle the complexities of loading and managing data during the training process. It is particularly useful when working with large datasets, as it helps to streamline the data loading process by handling batching, shuffling, and parallel data loading.

9.4.2. Key Features of `DataLoader`

1. **Batching**: The `DataLoader` allows you to split your dataset into smaller batches. This is crucial for training models on large datasets, as it enables the processing of smaller subsets of data at a time, reducing memory usage and speeding up training.
2. **Shuffling**: To prevent the model from learning the order of the data, which could lead to overfitting, the `DataLoader` can shuffle the data at every epoch. This ensures that the model generalizes better to unseen data.
3. **Parallel Loading with Multiple Workers**: The `DataLoader` can utilize multiple subprocesses to load data in parallel using the `num_workers` parameter. This is particularly beneficial when dealing with large datasets or complex preprocessing, as it reduces the data loading time by parallelizing the process.

4. **Custom Samplers**: The `DataLoader` allows the use of custom sampling strategies by providing different `Sampler` classes. This is useful when you need to apply specific sampling methods, such as weighted sampling, which can help in dealing with imbalanced datasets.
5. **Support for Iterable Datasets**: Besides regular datasets, the `DataLoader` can work with iterable datasets, which are particularly useful when the data is generated on-the-fly or when working with streams of data.
6. **Collate Function**: The `DataLoader` provides an option to define a custom `collate_fn`, which allows you to specify how the individual data samples should be merged into a single batch. This is useful for handling variable-length sequences or more complex data structures.

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=mean.tolist(), std=std.tolist())
])

full_dataset = datasets.ImageFolder(root="data/images_original", transform=transform)

indices = list(range(len(full_dataset)))
labels = [full_dataset.targets[i] for i in indices]

train_valid_idx, test_idx = train_test_split(
    indices, test_size=0.1, stratify=labels, random_state=42
)
train_idx, valid_idx = train_test_split(
    train_valid_idx, test_size=0.1111, stratify=[labels[i] for i in train_valid_idx], random_state=42
)

train_dataset = Subset(full_dataset, train_idx)
valid_dataset = Subset(full_dataset, valid_idx)
test_dataset  = Subset(full_dataset, test_idx)

train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True, num_workers=2)
valid_loader = DataLoader(valid_dataset, batch_size=16, shuffle=False, num_workers=2)
test_loader  = DataLoader(test_dataset, batch_size=16, shuffle=False, num_workers=2)

print(f"Training samples: {len(train_dataset)}")
print(f"Validation samples: {len(valid_dataset)}")
print(f"Testing samples: {len(test_dataset)}")
```

```
Training samples: 799
Validation samples: 100
Testing samples: 100
```

9.5. Checking Class Distribution

Scikit-learn's `train_test_split` provides an option for **stratified sampling** based on labels. This ensures that each of the **train, validation, and test splits** contains the **same proportion of samples from each class** (each music genre).

In the next code block, we will **verify this stratification** by printing the class distribution percentages for each split.

```python
def check_class_distribution(dataset_loader, class_names=None):
    """
    Computes the percentage distribution of classes in a dataset loader.

    Args:
        dataset_loader (DataLoader): The data loader containing the dataset.
        class_names (list, optional): List of class names corresponding to the labels.
                                      If provided, the output will map class names to percentages.

    Returns:
        dict: A dictionary mapping class labels or class names to their percentage in the dataset.
    """
    class_counts = {}
    total_samples = 0
    for _, labels in dataset_loader:
        for label in labels:
            label = label.item()
            class_counts[label] = class_counts.get(label, 0) + 1
            total_samples += 1

    class_percentages = {}
    for label, count in class_counts.items():
        class_percentages[label] = (count / total_samples) * 100

    return {class_names[label]: percentage for label, percentage in class_percentages.items()}

def print_class_distribution(name, distribution):
    """
    Prints the class distribution in a readable format.

    Args:
        name (str): Name of the dataset split (e.g., 'Training', 'Validation', 'Testing').
        distribution (dict): Dictionary with class names (or labels) as keys and percentages as values.
    """
    print(f"{name} Set Class Distribution:")
    for cls, perc in distribution.items():
        print(f"  {cls}: {perc:.2f}%")
    print()

class_names = full_dataset.classes

train_percentages = check_class_distribution(train_loader, class_names=class_names)
valid_percentages = check_class_distribution(valid_loader, class_names=class_names)
test_percentages  = check_class_distribution(test_loader, class_names=class_names)

print_class_distribution("Training", train_percentages)
print_class_distribution("Validation", valid_percentages)
print_class_distribution("Testing", test_percentages)
```

```
Training Set Class Distribution:
  hiphop: 10.01%
  rock: 10.01%
  metal: 10.01%
  classical: 10.01%
  reggae: 10.01%
  blues: 10.01%
  pop: 10.01%
  jazz: 9.89%
  country: 10.01%
  disco: 10.01%
```

```
Validation Set Class Distribution:
  classical: 10.00%
  reggae: 10.00%
  blues: 10.00%
  metal: 10.00%
  rock: 10.00%
  hiphop: 10.00%
  country: 10.00%
  disco: 10.00%
  pop: 10.00%
  jazz: 10.00%

Testing Set Class Distribution:
  disco: 10.00%
  hiphop: 10.00%
  classical: 10.00%
  blues: 10.00%
  rock: 10.00%
  pop: 10.00%
  country: 10.00%
  reggae: 10.00%
  jazz: 10.00%
  metal: 10.00%
```

9.6. Train / Validation / Test splits and data leakage

GTZAN is small, 1 000 clips over 10 genres, so the test set is the single strongest check we have on whether the model is learning music or learning the quirks of the collection. Three practical rules keep the evaluation honest.

1. **Split once, commit.** Pick a split seed at the start of the project and never re-split to improve a result. A tiny amount of seed-shopping is enough to publish a non-reproducible accuracy.
2. **No per-clip leakage across splits.** GTZAN clips are 30 s long. If you slice each clip into 3 s chunks *before* the split, chunks from the same clip will end up in both training and test. The model then recognises the specific recording, not the genre. Split at the clip level, never at the chunk level.
3. **Keep test set untouched until the final number.** Use the validation set to tune hyper-parameters, compare architectures, pick early-stopping epochs. Touch the test set once, at the end, to report the number that goes in the paper or report.

```python
# Example: clip-level split. `filenames` is a list of relative paths like
# 'jazz/jazz.00012.wav'. We split at the clip level BEFORE any chunking.
from sklearn.model_selection import train_test_split

# placeholder list reflecting a typical GTZAN structure
filenames = [f'{genre}/{genre}.{i:05d}.wav'
             for genre in ['jazz', 'metal', 'reggae']
             for i in range(100)]
labels = [f.split('/')[0] for f in filenames]

train_fn, temp_fn, train_y, temp_y = train_test_split(
    filenames, labels, test_size=0.30, stratify=labels, random_state=42,
)
val_fn, test_fn, _, _ = train_test_split(
    temp_fn, temp_y, test_size=0.50, stratify=temp_y, random_state=42,
```

```
)
print(f'train: {len(train_fn):3d} clips, '
      f'val: {len(val_fn):3d} clips, '
      f'test: {len(test_fn):3d} clips')
```

9.7. Creating our Model

Our model is a **simple CNN** consisting of **four convolutional blocks**.

9.7.1. Architecture Overview:

- Each **convolutional block** contains **two sequential layers** following the pattern: **(Conv -> BatchNorm -> ReLU)**
- A **MaxPooling layer** at the end of each block **reduces spatial dimensions by half**.
- Finally, a **Global Average Pooling (GAP) layer** is applied, followed by a **fully connected linear layer** that outputs logits for classification.

```
class MusicNet(nn.Module):
    def __init__(self, num_classes=10):
        """
        Improved CNN architecture for spectrogram classification.

        The network is divided into several convolutional blocks.
        Each block contains two convolutional layers with Batch Normalization
        and ReLU activations, followed by a max pooling layer.
        Global average pooling is used at the end to aggregate spatial features,
        followed by dropout and a fully connected layer for classification.

        Args:
            num_classes (int): Number of target classes.
        """
        super(MusicNet, self).__init__()

        # Block 1
        self.block1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2)
        )

        # Block 2
        self.block2 = nn.Sequential(
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2)
        )

        # Block 3
```

```
        self.block3 = nn.Sequential(
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2)
        )

        # Block 4
        self.block4 = nn.Sequential(
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2)
        )

        # Global average pooling to collapse the spatial dimensions.
        self.global_avg_pool = nn.AdaptiveAvgPool2d((1, 1))
        self.dropout = nn.Dropout(p=0.5)
        self.fc = nn.Linear(256, num_classes)

    def forward(self, x):
        x = self.block1(x)
        x = self.block2(x)
        x = self.block3(x)
        x = self.block4(x)

        x = self.global_avg_pool(x)  # [B, 256, 1, 1]
        x = x.view(x.size(0), -1)    # [B, 256]

        x = self.dropout(x)
        x = self.fc(x)
        return x
```

9.8. Training our Model

An alternative and commonly used approach in PyTorch is to **structure training loops** by separating **one training epoch** and **model validation** into two distinct functions. These functions are then called sequentially within a larger **training loop**.

9.8.1. Why Use This Approach?

- **Cleaner code structure:** The main training loop remains uncluttered, while the underlying logic is handled in separate functions.
- **Easier debugging and modification:** Training and validation processes can be adjusted independently.

9.8.2. Tracking Performance & Early Stopping

- We will **track both training and validation losses** along with **accuracy**.

- **Early stopping** is enabled on the **validation loss**:
  - If the validation loss does not improve for **10 consecutive epochs**, training will be stopped to prevent overfitting.

```
def train_one_epoch(model, device, dataloader, criterion, optimizer):
    """Train the model for one epoch."""
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for data, target in dataloader:
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        outputs = model(data)
        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()

        # Accumulate loss.
        running_loss += loss.item() * data.size(0)

        # Compute accuracy.
        _, predicted = torch.max(outputs, 1)
        total += target.size(0)
        correct += (predicted == target).sum().item()

    epoch_loss = running_loss / total
    epoch_acc = correct / total
    return epoch_loss, epoch_acc

def validate(model, device, dataloader, criterion):
    """Evaluate the model on a validation set."""
    model.eval()
    running_loss = 0.0
    correct = 0
    total = 0
    with torch.no_grad():
        for data, target in dataloader:
            data, target = data.to(device), target.to(device)
            outputs = model(data)
            loss = criterion(outputs, target)
            running_loss += loss.item() * data.size(0)

            _, predicted = torch.max(outputs, 1)
            total += target.size(0)
            correct += (predicted == target).sum().item()

    epoch_loss = running_loss / total
    epoch_acc = correct / total
    return epoch_loss, epoch_acc
```

9.9. Loss function

`CrossEntropyLoss` is a commonly used loss function in PyTorch for classification tasks. It combines `nn.LogSoftmax` and `nn.NLLLoss` in one single class. This loss function is particularly useful when

training models for multi-class classification, where the target is expected to be in the form of class indices (not one-hot encoded).

Key Points: - **Input**: The input to `nn.CrossEntropyLoss` should be raw, unnormalized scores (also known as logits) of shape `(N, C)` where `N` is the batch size and `C` is the number of classes. - **Target**: The target should contain the class indices, with shape `(N)` for each sample in the batch. - **Output**: The output is a scalar that represents the average negative log-likelihood across the batch.

Example input and target: input = torch.randn(3, 5, requires_grad=True) # 3 samples, 5 classes target = torch.tensor([1, 0, 4]) # Class indices for each sample

9.9.1. Optimizers and their options

There are some pre-built Optimizers in PyTorch, they are sufficient in most cases, especially if their parameters are well set. The two most well-known are ADAM and SGD, both of which originate from Gradient Descent, which we implemented earlier.

- **S**tochastic **G**radient **D**escent (SGD)
- **ADA**ptive **M**oment optimizer (ADAM)
- A good general overview

```
def train_model(model, device, train_loader, val_loader, epochs=50, patience=5):
    """
    Train the model with early stopping based on validation loss.

    Args:
        model: The CNN model.
        device: Device to run on (CPU or GPU).
        train_loader: DataLoader for training data.
        val_loader: DataLoader for validation data.
        epochs: Maximum number of training epochs.
        patience: Number of epochs to wait for improvement before stopping.

    Returns:
        model: The best model (with lowest validation loss).
        train_losses: List of training losses per epoch.
        train_accs: List of training accuracies per epoch.
        val_losses: List of validation losses per epoch.
        val_accs: List of validation accuracies per epoch.
    """
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
    scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', factor=0.5, patience=2)

    train_losses, train_accs = [], []
    val_losses, val_accs = [], []

    best_val_loss = float('inf')
    patience_counter = 0
    best_model_wts = None

    for epoch in range(epochs):
        train_loss, train_acc = train_one_epoch(model, device, train_loader, criterion, optimizer)
        val_loss, val_acc = validate(model, device, val_loader, criterion)

        train_losses.append(train_loss)
```

```
        train_accs.append(train_acc)
        val_losses.append(val_loss)
        val_accs.append(val_acc)

        print(f"Epoch [{epoch+1}/{epochs}]  "
              f"Train Loss: {train_loss:.4f}  Train Acc: {train_acc:.4f}  |  "
              f"Val Loss: {val_loss:.4f}  Val Acc: {val_acc:.4f}")

        scheduler.step(val_loss)

        # Check for improvement.
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            best_model_wts = model.state_dict()
            patience_counter = 0
        else:
            patience_counter += 1

        # Early stopping trigger.
        if patience_counter >= patience:
            print("Early stopping triggered.")
            break

    # Load the best model weights.
    if best_model_wts is not None:
        model.load_state_dict(best_model_wts)
    return model, train_losses, train_accs, val_losses, val_accs
```

```
model = MusicNet().to(device)

model, train_losses, train_accs, val_losses, val_accs = train_model(
    model, device, train_loader, valid_loader, epochs=50, patience=10
)
```

```
Epoch [1/50]  Train Loss: 1.9714  Train Acc: 0.2591  |  Val Loss: 2.3034  Val
Acc: 0.1800
Epoch [2/50]  Train Loss: 1.6269  Train Acc: 0.4018  |  Val Loss: 5.0673  Val
Acc: 0.1000
Epoch [3/50]  Train Loss: 1.5025  Train Acc: 0.4743  |  Val Loss: 2.4380  Val
Acc: 0.2300
Epoch [4/50]  Train Loss: 1.3935  Train Acc: 0.5094  |  Val Loss: 1.6903  Val
Acc: 0.3200
Epoch [5/50]  Train Loss: 1.3369  Train Acc: 0.5332  |  Val Loss: 2.3599  Val
Acc: 0.2000
Epoch [6/50]  Train Loss: 1.2779  Train Acc: 0.5519  |  Val Loss: 2.9637  Val
Acc: 0.1500
Epoch [7/50]  Train Loss: 1.2153  Train Acc: 0.5782  |  Val Loss: 1.4785  Val
Acc: 0.5300
Epoch [8/50]  Train Loss: 1.1352  Train Acc: 0.6095  |  Val Loss: 1.3605  Val
Acc: 0.5200
Epoch [9/50]  Train Loss: 1.1207  Train Acc: 0.6320  |  Val Loss: 1.6421  Val
Acc: 0.4600
Epoch [10/50]  Train Loss: 1.1276  Train Acc: 0.6408  |  Val Loss: 1.3805  Val
Acc: 0.5400
Epoch [11/50]  Train Loss: 1.0473  Train Acc: 0.6345  |  Val Loss: 1.7692  Val
Acc: 0.4000
Epoch [12/50]  Train Loss: 0.9944  Train Acc: 0.6721  |  Val Loss: 1.4030  Val
```

```
Acc: 0.5200
Epoch [13/50]  Train Loss: 0.8988  Train Acc: 0.7309  |  Val Loss: 1.7183  Val
Acc: 0.4600
Epoch [14/50]  Train Loss: 0.9022  Train Acc: 0.7247  |  Val Loss: 1.1288  Val
Acc: 0.6100
Epoch [15/50]  Train Loss: 0.8519  Train Acc: 0.7297  |  Val Loss: 1.9496  Val
Acc: 0.3900
Epoch [16/50]  Train Loss: 0.8457  Train Acc: 0.7259  |  Val Loss: 1.5372  Val
Acc: 0.4900
Epoch [17/50]  Train Loss: 0.8017  Train Acc: 0.7472  |  Val Loss: 1.6161  Val
Acc: 0.5200
Epoch [18/50]  Train Loss: 0.7690  Train Acc: 0.7522  |  Val Loss: 1.2373  Val
Acc: 0.6400
Epoch [19/50]  Train Loss: 0.7160  Train Acc: 0.7910  |  Val Loss: 1.0121  Val
Acc: 0.6600
Epoch [20/50]  Train Loss: 0.6935  Train Acc: 0.7885  |  Val Loss: 1.3253  Val
Acc: 0.5500
[... 20 lines omitted ...]
Epoch [41/50]  Train Loss: 0.4964  Train Acc: 0.8673  |  Val Loss: 0.8147  Val
Acc: 0.7500
Epoch [42/50]  Train Loss: 0.5118  Train Acc: 0.8698  |  Val Loss: 0.7900  Val
Acc: 0.7600
Epoch [43/50]  Train Loss: 0.4688  Train Acc: 0.8824  |  Val Loss: 0.7928  Val
Acc: 0.7600
Epoch [44/50]  Train Loss: 0.4976  Train Acc: 0.8686  |  Val Loss: 0.7988  Val
Acc: 0.7700
Epoch [45/50]  Train Loss: 0.4839  Train Acc: 0.8661  |  Val Loss: 0.7951  Val
Acc: 0.7900
Epoch [46/50]  Train Loss: 0.4811  Train Acc: 0.8636  |  Val Loss: 0.7868  Val
Acc: 0.7500
Epoch [47/50]  Train Loss: 0.4818  Train Acc: 0.8798  |  Val Loss: 0.7927  Val
Acc: 0.7700
Epoch [48/50]  Train Loss: 0.4893  Train Acc: 0.8748  |  Val Loss: 0.7968  Val
Acc: 0.7800
Epoch [49/50]  Train Loss: 0.4765  Train Acc: 0.8698  |  Val Loss: 0.7825  Val
Acc: 0.7600
Epoch [50/50]  Train Loss: 0.4755  Train Acc: 0.8836  |  Val Loss: 0.7842  Val
Acc: 0.7700
```

Saving our Model

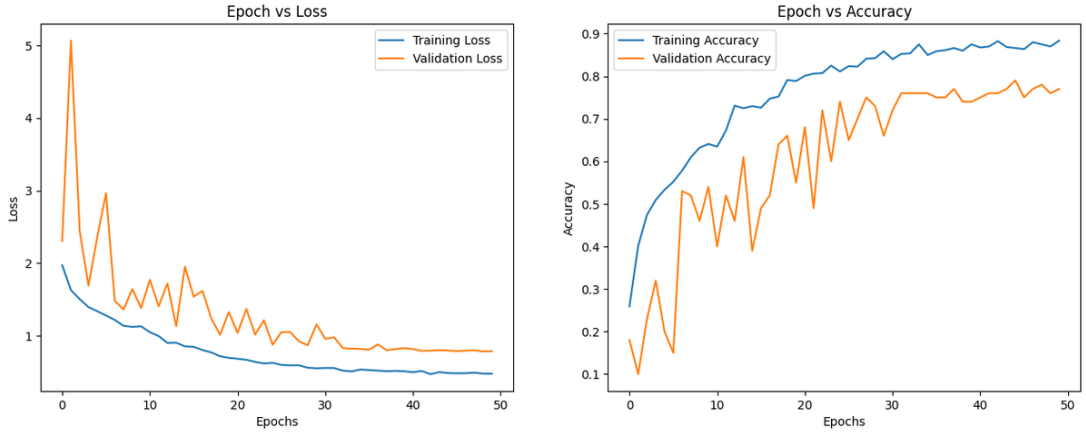
```
torch.save(model.state_dict(), 'best_model_wts.pth')
```

9.9.2. Plotting our Losses and Accuracies

```
def plot_loss_accuracy(train_loss, train_acc, validation_loss, validation_acc):
    epochs = len(train_loss)
    fig, (ax1, ax2) = plt.subplots(1, 2)
    ax1.plot(list(range(epochs)), train_loss, label='Training Loss')
    ax1.plot(list(range(epochs)), validation_loss, label='Validation Loss')
    ax1.set_xlabel('Epochs')
    ax1.set_ylabel('Loss')
    ax1.set_title('Epoch vs Loss')
    ax1.legend()
```

```
    ax2.plot(list(range(epochs)), train_acc, label='Training Accuracy')
    ax2.plot(list(range(epochs)), validation_acc, label='Validation Accuracy')
    ax2.set_xlabel('Epochs')
    ax2.set_ylabel('Accuracy')
    ax2.set_title('Epoch vs Accuracy')
    ax2.legend()
    fig.set_size_inches(15.5, 5.5)
    plt.show()
```

```
plot_loss_accuracy(train_losses, train_accs, val_losses, val_accs)
```



9.10. Testing our Model

```
def inference(model, device, test_loader):
    """
    Run inference on the test set and calculate accuracy.

    Args:
        model (nn.Module): The trained model.
        device (torch.device): Device to run inference on (CPU or GPU).
        test_loader (DataLoader): DataLoader for the test set.

    Returns:
        float: Test accuracy (between 0 and 1).
    """
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)

            _, predicted = torch.max(output, dim=1)
            total += target.size(0)
            correct += (predicted == target).sum().item()
```

```
    accuracy = correct / total
    return accuracy
```

```
loaded_model = MusicNet().to(device)
loaded_model.load_state_dict(torch.load('best_model_wts.pth', weights_only=True))

test_accuracy = inference(loaded_model, device, test_loader)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
```

```
Test Accuracy: 85.00%
```

9.10.1. Data Augmentation for Spectrograms

Since standard image augmentation techniques (e.g., random cropping, flipping, or rotation) may not be applicable here, **as they disrupt the temporal correlations in the audio signal**, we will instead use augmentation methods specifically designed for spectrograms.

Instead, we will apply one or more augmentation techniques proposed in **Park et al., 2019** (`SpecAugment`). These methods are **designed for spectrograms** and retain the structural integrity of the data. Some of these include: - **Time masking**: Randomly masking a section of the spectrogram in the time domain, simulating dropped information. - **Frequency masking**: Masking specific frequency bands to simulate real-world variations in the sound spectrum. - **Time warping**: Stretching or compressing parts of the spectrogram to create slight variations in tempo.

```
class SpecAugmentTransform(object):
    """
    Applies SpecAugment to a spectrogram tensor.

    Args:
        freq_mask_param (int): Maximum width (in frequency bins) for frequency masking.
        time_mask_param (int): Maximum width (in time steps) for time masking.
        num_freq_masks (int): Number of frequency masks to apply.
        num_time_masks (int): Number of time masks to apply.
    """
    def __init__(self, freq_mask_param=10, time_mask_param=40, num_freq_masks=2, num_time_masks=2):
        self.freq_mask_param = freq_mask_param
        self.time_mask_param = time_mask_param
        self.num_freq_masks = num_freq_masks
        self.num_time_masks = num_time_masks

    def __call__(self, tensor):
        augmented = tensor.clone()
        C, H, W = augmented.shape

        for _ in range(self.num_freq_masks):
            f = random.randint(0, self.freq_mask_param)
            if f == 0:
                continue
            f0 = random.randint(0, max(0, H - f))
            augmented[:, f0:f0+f, :] = 0

        for _ in range(self.num_time_masks):
            t = random.randint(0, self.time_mask_param)
            if t == 0:
                continue
            t0 = random.randint(0, max(0, W - t))
```

```
            augmented[:, :, t0:t0+t] = 0

        return augmented
```

```
train_transforms = transforms.Compose([
    transforms.ToTensor(),
    SpecAugmentTransform(freq_mask_param=5, time_mask_param=20, num_freq_masks=1, num_time_masks=1),
    transforms.Normalize(mean=mean.tolist(), std=std.tolist()),
])

test_transforms = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=mean.tolist(), std=std.tolist())
])

full_train_dataset = datasets.ImageFolder(root="data/images_original", transform=train_transforms)
full_test_dataset  = datasets.ImageFolder(root="data/images_original", transform=test_transforms)

indices = list(range(len(full_train_dataset)))
labels = [full_train_dataset.targets[i] for i in indices]

train_valid_idx, test_idx = train_test_split(indices, test_size=0.1, stratify=labels, random_state=42)
train_idx, valid_idx = train_test_split(
    train_valid_idx,
    test_size=0.11111,
    stratify=[labels[i] for i in train_valid_idx],
    random_state=42
)

train_dataset = Subset(full_train_dataset, train_idx)
valid_dataset = Subset(full_train_dataset, valid_idx)
test_dataset  = Subset(full_test_dataset, test_idx)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True, num_workers=2)
valid_loader = DataLoader(valid_dataset, batch_size=64, shuffle=False, num_workers=2)
test_loader  = DataLoader(test_dataset, batch_size=64, shuffle=False, num_workers=2)

print(f"Training samples: {len(train_dataset)}")
print(f"Validation samples: {len(valid_dataset)}")
print(f"Testing samples: {len(test_dataset)}")
```

```
Training samples: 799
Validation samples: 100
Testing samples: 100
```

Checking Class Distribution

```
train_percentages = check_class_distribution(train_loader, class_names=class_names)
valid_percentages = check_class_distribution(valid_loader, class_names=class_names)
test_percentages  = check_class_distribution(test_loader, class_names=class_names)

print_class_distribution("Training", train_percentages)
print_class_distribution("Validation", valid_percentages)
print_class_distribution("Testing", test_percentages)
```

```
Training Set Class Distribution:
  blues: 10.01%
  rock: 10.01%
  classical: 10.01%
  reggae: 10.01%
```

```
  metal: 10.01%
  disco: 10.01%
  hiphop: 10.01%
  country: 10.01%
  jazz: 9.89%
  pop: 10.01%

Validation Set Class Distribution:
  classical: 10.00%
  reggae: 10.00%
  blues: 10.00%
  metal: 10.00%
  rock: 10.00%
  hiphop: 10.00%
  country: 10.00%
  disco: 10.00%
  pop: 10.00%
  jazz: 10.00%

Testing Set Class Distribution:
  disco: 10.00%
  hiphop: 10.00%
  classical: 10.00%
  blues: 10.00%
  rock: 10.00%
  pop: 10.00%
  country: 10.00%
  reggae: 10.00%
  jazz: 10.00%
  metal: 10.00%
```

9.11. Training a New Model

```
model_augment = MusicNet().to(device)

model_augment, train_losses, train_accs, val_losses, val_accs = train_model(
    model_augment, device, train_loader, valid_loader, epochs=50, patience=10
)
```

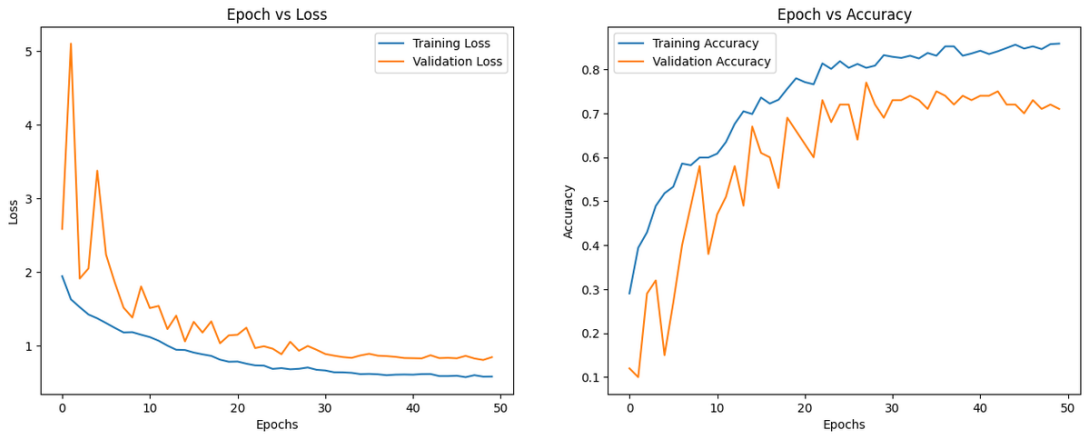
```
Epoch [1/50]  Train Loss: 1.9425  Train Acc: 0.2904  |  Val Loss: 2.5840  Val
Acc: 0.1200
Epoch [2/50]  Train Loss: 1.6273  Train Acc: 0.3942  |  Val Loss: 5.1021  Val
Acc: 0.1000
Epoch [3/50]  Train Loss: 1.5215  Train Acc: 0.4293  |  Val Loss: 1.9097  Val
Acc: 0.2900
Epoch [4/50]  Train Loss: 1.4215  Train Acc: 0.4894  |  Val Loss: 2.0473  Val
Acc: 0.3200
Epoch [5/50]  Train Loss: 1.3695  Train Acc: 0.5181  |  Val Loss: 3.3753  Val
Acc: 0.1500
Epoch [6/50]  Train Loss: 1.3055  Train Acc: 0.5332  |  Val Loss: 2.2356  Val
Acc: 0.2700
Epoch [7/50]  Train Loss: 1.2399  Train Acc: 0.5857  |  Val Loss: 1.8559  Val
Acc: 0.4000
Epoch [8/50]  Train Loss: 1.1775  Train Acc: 0.5820  |  Val Loss: 1.5144  Val
Acc: 0.4900
Epoch [9/50]  Train Loss: 1.1811  Train Acc: 0.5995  |  Val Loss: 1.3806  Val
Acc: 0.5800
```

```
Epoch [10/50]  Train Loss: 1.1465  Train Acc: 0.5995  |  Val Loss: 1.8022  Val
Acc: 0.3800
Epoch [11/50]  Train Loss: 1.1145  Train Acc: 0.6083  |  Val Loss: 1.5090  Val
Acc: 0.4700
Epoch [12/50]  Train Loss: 1.0649  Train Acc: 0.6345  |  Val Loss: 1.5378  Val
Acc: 0.5100
Epoch [13/50]  Train Loss: 0.9988  Train Acc: 0.6758  |  Val Loss: 1.2232  Val
Acc: 0.5800
Epoch [14/50]  Train Loss: 0.9423  Train Acc: 0.7046  |  Val Loss: 1.4062  Val
Acc: 0.4900
Epoch [15/50]  Train Loss: 0.9388  Train Acc: 0.6984  |  Val Loss: 1.0558  Val
Acc: 0.6700
Epoch [16/50]  Train Loss: 0.9044  Train Acc: 0.7359  |  Val Loss: 1.3222  Val
Acc: 0.6100
Epoch [17/50]  Train Loss: 0.8802  Train Acc: 0.7222  |  Val Loss: 1.1764  Val
Acc: 0.6000
Epoch [18/50]  Train Loss: 0.8589  Train Acc: 0.7309  |  Val Loss: 1.3277  Val
Acc: 0.5300
Epoch [19/50]  Train Loss: 0.8078  Train Acc: 0.7559  |  Val Loss: 1.0302  Val
Acc: 0.6900
Epoch [20/50]  Train Loss: 0.7794  Train Acc: 0.7797  |  Val Loss: 1.1377  Val
Acc: 0.6600
[... 20 lines omitted ...]
Epoch [41/50]  Train Loss: 0.6040  Train Acc: 0.8423  |  Val Loss: 0.8279  Val
Acc: 0.7400
Epoch [42/50]  Train Loss: 0.6118  Train Acc: 0.8348  |  Val Loss: 0.8256  Val
Acc: 0.7400
Epoch [43/50]  Train Loss: 0.6128  Train Acc: 0.8411  |  Val Loss: 0.8678  Val
Acc: 0.7500
Epoch [44/50]  Train Loss: 0.5848  Train Acc: 0.8486  |  Val Loss: 0.8293  Val
Acc: 0.7200
Epoch [45/50]  Train Loss: 0.5847  Train Acc: 0.8561  |  Val Loss: 0.8327  Val
Acc: 0.7200
Epoch [46/50]  Train Loss: 0.5894  Train Acc: 0.8473  |  Val Loss: 0.8266  Val
Acc: 0.7000
Epoch [47/50]  Train Loss: 0.5701  Train Acc: 0.8523  |  Val Loss: 0.8596  Val
Acc: 0.7300
Epoch [48/50]  Train Loss: 0.5970  Train Acc: 0.8461  |  Val Loss: 0.8245  Val
Acc: 0.7100
Epoch [49/50]  Train Loss: 0.5773  Train Acc: 0.8573  |  Val Loss: 0.8037  Val
Acc: 0.7200
Epoch [50/50]  Train Loss: 0.5786  Train Acc: 0.8586  |  Val Loss: 0.8419  Val
Acc: 0.7100
```

Loading and Visualizing Losses and Accuracies

```
torch.save(model.state_dict(), 'best_model_augment_wts.pth')
```

```
plot_loss_accuracy(train_losses, train_accs, val_losses, val_accs)
```




```
loaded_aug_model = MusicNet().to(device)
loaded_aug_model.load_state_dict(torch.load('best_model_augment_wts.pth', weights_only=True))

test_accuracy = inference(loaded_aug_model, device, test_loader)
print(f"Test Accuracy: {test_accuracy * 100:.2f}%")
```

```
Test Accuracy: 85.00%
```

9.12. Visualizing Feature Vectors

Instead of **classification**, we will use the **previously trained neural network** to **extract useful feature vectors**, which should be the **inputs to the last dense layer**.

Then, we will **project these feature vectors into two dimensions** and **visualize them on a plot** to observe **how well the music genres cluster together**.

For better clarity, we will **assign the same color to vectors belonging to the same music genre** in the visualization.

```
def extract_features(model, dataloader, device):
    """
    Extracts feature vectors from the model just before the final dense layer.

    Args:
        model (nn.Module): The pretrained model.
        dataloader (DataLoader): DataLoader for the dataset.
        device (torch.device): Device on which to run inference.

    Returns:
        features (Tensor): Extracted feature vectors of shape (N, feature_dim).
        labels (Tensor): Corresponding genre labels.
    """
    model.eval()
    features_list = []
    labels_list = []

    # Define a hook to capture the output from the dropout layer.
    def hook_fn(module, input, output):
        features_list.append(output.detach().cpu()) # (batch_size, feature_dim)
```

```
    # Register the hook on the dropout layer.
    hook_handle = model.dropout.register_forward_hook(hook_fn)

    with torch.no_grad():
        for data, target in dataloader:
            data = data.to(device)
            _ = model(data)
            labels_list.append(target)

    hook_handle.remove()
    features = torch.cat(features_list, dim=0)
    labels = torch.cat(labels_list, dim=0)
    return features, labels
```

t-SNE for Feature Projection

`t-Distributed Stochastic Neighbor Embedding` (t-SNE) is a non-linear dimensionality reduction technique used primarily for visualizing high-dimensional data in two or three dimensions.

It **preserves local structures** by modeling pairwise similarities between data points in the original space and mapping them to a lower-dimensional representation, making it particularly effective for visualizing clusters and latent structures.

```
features, labels = extract_features(loaded_aug_model, test_loader, device)

# Project the features into 2 dimensions using t-SNE.
tsne = TSNE(n_components=2, random_state=42)
features_2d = tsne.fit_transform(features.numpy())
```

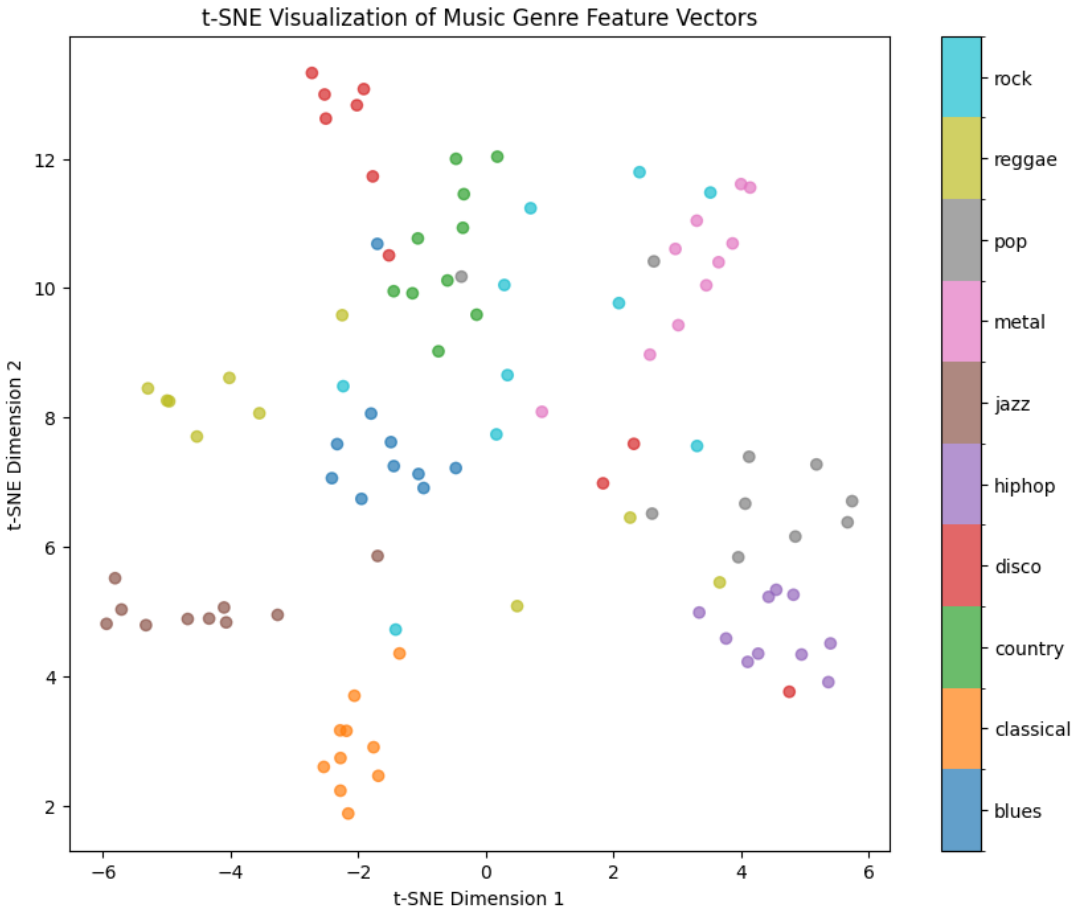
```
classes = full_dataset.classes
num_classes = len(classes)

cmap = matplotlib.colormaps.get_cmap("tab10")
norm = matplotlib.colors.BoundaryNorm(boundaries=np.arange(num_classes + 1) - 0.5, ncolors=num_classes)

plt.figure(figsize=(10, 8))
scatter = plt.scatter(features_2d[:, 0], features_2d[:, 1], c=labels, cmap=cmap, norm=norm, alpha=0.7)

cb = plt.colorbar(scatter, ticks=np.arange(num_classes))
cb.ax.set_yticklabels(classes)

plt.xlabel("t-SNE Dimension 1")
plt.ylabel("t-SNE Dimension 2")
plt.title("t-SNE Visualization of Music Genre Feature Vectors")
plt.show()
```



The `t-SNE` projection shows that the final-layer feature vectors are **not linearly separable in two dimensions**, which is expected given the complexity and high dimensionality of the underlying data.

However, these features can still be effectively distinguished by **non-linear decision boundaries in the original feature space**, where local relationships are preserved more accurately.

9.13. Music Genre Similarities

In the next step we will quantify how similar the music genres are by **calculating the cosine similarity** between the feature vectors of samples belonging to different genres.

A **similarity measure** (also called a similarity function or metric) quantifies how alike two entities are, typically yielding higher values for more similar entities and lower values for less similar ones. Conversely, a **distance measure** assesses dissimilarity, assigning larger values to entities that are further apart in the given feature space.

Three widely used similarity and distance metrics in vector space models are **Euclidean Distance**, **Dot Product Similarity**, and **Cosine Similarity**.

9.13.1. Euclidean Distance

Euclidean distance represents the straight-line distance between two vectors in a multi-dimensional space, considering both **magnitude** and **direction**. It is computed as:

$$d(a, b) = \sqrt{\sum_{i=1}^{n} (a_i - b_i)^2}$$

where: - a and b are vectors in $\mathbb{R}^n$, - a_i and b_i are the components of the vectors.

Euclidean distance provides a straightforward way to measure how far two points (or vectors) are from each other in space.

9.13.2. Dot Product Similarity

The **dot product similarity** measures how much two vectors align with each other. It is calculated as:

$$S(a, b) = \sum_{i=1}^{n} a_i b_i$$

where: - $a \cdot b$ represents the dot product, - a_i and b_i are the vector components.

The dot product is sensitive to both vector magnitude and direction, making it useful for quantifying similarity in raw vector space.

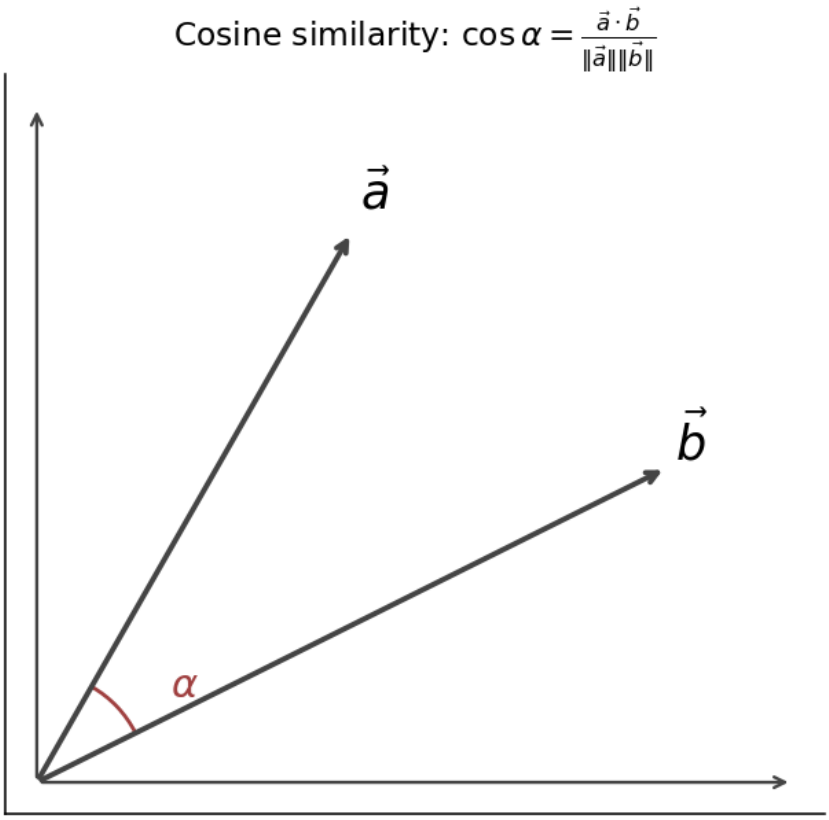
9.13.3. Cosine Similarity

Cosine similarity measures the **angle** between two vectors rather than their magnitude. It evaluates whether two vectors point in the same direction, making it a common choice in **text analysis** and **word embeddings**.

$$\cos(a, b) = \frac{a \cdot b}{\|a\| \|b\|}$$

where: - $a \cdot b$ is the dot product of the two vectors, - $\|a\|$ and $\|b\|$ are the magnitudes (norms) of the vectors.

Cosine similarity ranges from **-1 to 1**: - 1 indicates vectors pointing in the **same direction** (high similarity), - 0 means the vectors are **orthogonal** (unrelated), - -1 suggests vectors are **pointing in opposite directions** (complete dissimilarity).

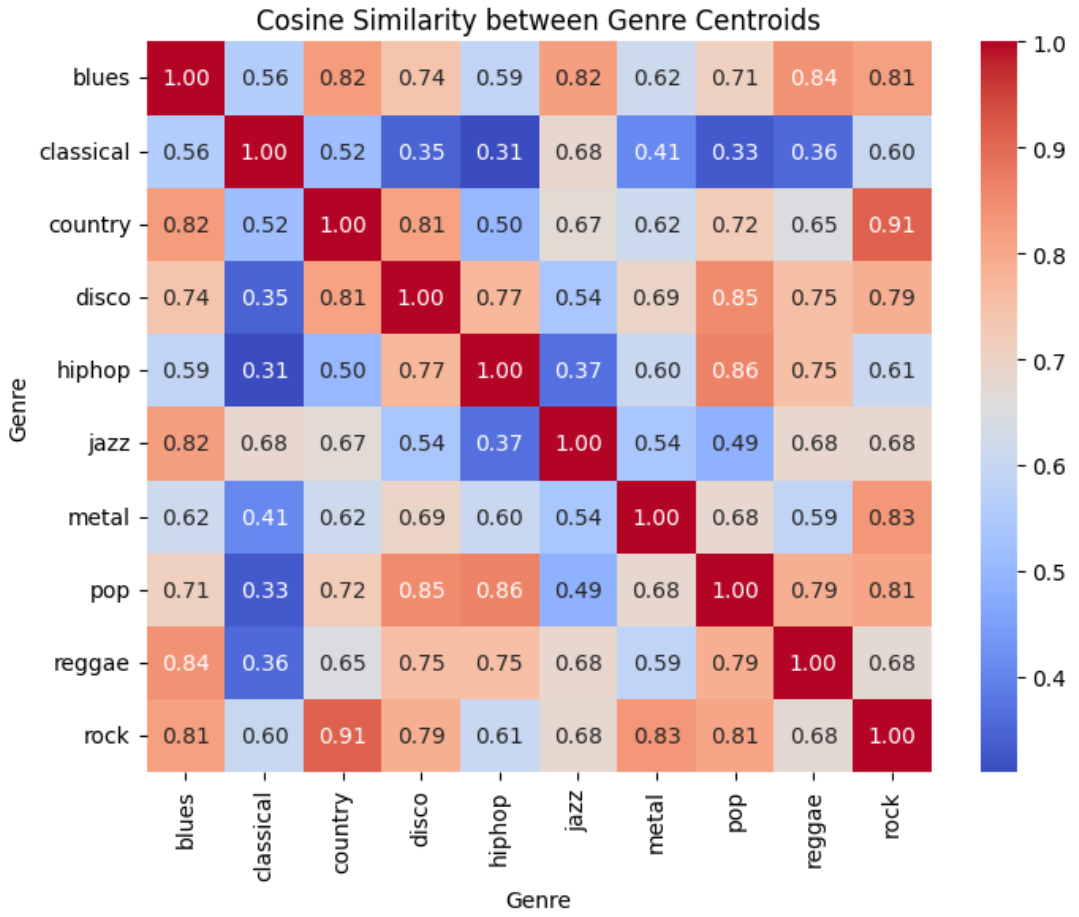



```
features_np = features.numpy()
labels_np = labels.numpy()

unique_labels = np.unique(labels_np)
centroids = []
for label in unique_labels:
    genre_features = features_np[labels_np == label]
    centroid = np.mean(genre_features, axis=0)
    centroids.append(centroid)
centroids = np.array(centroids)

cosine_sim_matrix = cosine_similarity(centroids)

plt.figure(figsize=(8, 6))
sns.heatmap(cosine_sim_matrix, annot=True, fmt=".2f", cmap='coolwarm',
            xticklabels=class_names, yticklabels=class_names)
plt.title('Cosine Similarity between Genre Centroids')
plt.xlabel('Genre')
plt.ylabel('Genre')
plt.show()
```



9.14. Integrated Gradients

9.14.1. Task – Integrated Gradients Visualization

9.15. Integrated Gradients Visualization

Integrated Gradients is an attribution method introduced by Sundararajan et al. in 2017 (Axiomatic Attribution for Deep Networks), designed to quantify the contribution of each input feature to a model's prediction. By integrating the gradients of the model's output with respect to the inputs along a path from a baseline to the actual input, this method provides insights into which features are most influential in the model's decision-making process.

To visualize what the network detects in a randomly chosen test set image, we apply the **Integrated Gradients method** as follows:

1. **Select a Test Image**: Randomly choose an image from the test dataset.
2. **Define a Baseline**: Typically, a baseline image (such as a black image) is used as a reference point.
3. **Compute Integrated Gradients**: Calculate the integrated gradients by interpolating between the baseline and the input image, computing the gradients at each step, and averaging them.

4. **Visualize Attributions**: Overlay the computed attributions on the original image to highlight the regions that significantly influence the model's prediction.

```
def unnormalize(tensor, mean, std):
    """
    Reverses the normalization applied to a tensor image using the given mean & std.
    tensor: shape [C, H, W]
    mean, std: lists or tensors of shape [3] for RGB images.
    """
    if not torch.is_tensor(mean):
        mean = torch.tensor(mean)
    if not torch.is_tensor(std):
        std = torch.tensor(std)
    mean = mean.view(-1, 1, 1)
    std = std.view(-1, 1, 1)
    return tensor * std + mean

def integrated_gradients_visualization(model, dataset, device, mean, std, n_steps=50):
    """
    1) Randomly select an image & label from dataset.
    2) Use a black baseline (all zeros).
    3) Compute IG for the predicted class.
    4) Clip attributions below 0 to 0, so everything <=0 is black.
    5) Plot the unnormalized image (left) & the clipped IG heatmap (right) using a 'gray' colormap
       where 0=black and max=white.

    Args:
        model (nn.Module): Trained model (e.g., MusicNet).
        dataset (torch.utils.data.Dataset): Dataset or Subset for sampling.
        device (torch.device): CPU or GPU.
        mean (list): Channel-wise mean used during normalization.
        std (list): Channel-wise std used during normalization.
        n_steps (int): Number of steps for the IG integration path.
    """
    model.eval()

    # 1) Randomly pick an example from the dataset
    idx = random.randint(0, len(dataset) - 1)
    x, true_label = dataset[idx]
    x = x.unsqueeze(0).to(device)

    with torch.no_grad():
        logits = model(x)
        pred_label = logits.argmax(dim=1).item()

    # 2) Define a black baseline
    baseline = torch.zeros_like(x)

    # 3) Compute Integrated Gradients for the predicted class
    ig = IntegratedGradients(model)
    attributions = ig.attribute(
        inputs=x,
        baselines=baseline,
        target=pred_label,
        n_steps=n_steps
    )

    attributions = attributions.squeeze(0)
    ig_heatmap = attributions.mean(dim=0).cpu().numpy()
```

```
    # 4) Clip attributions below 0 to 0 => everything <= 0 is black
    ig_clipped = np.clip(ig_heatmap, a_min=0, a_max=None)
    ig_min, ig_max = 0, ig_clipped.max()

    unnorm_x = unnormalize(x.squeeze(0).cpu(), mean, std)
    unnorm_x = torch.clamp(unnorm_x, 0, 1)
    original_img = unnorm_x.permute(1, 2, 0).numpy()

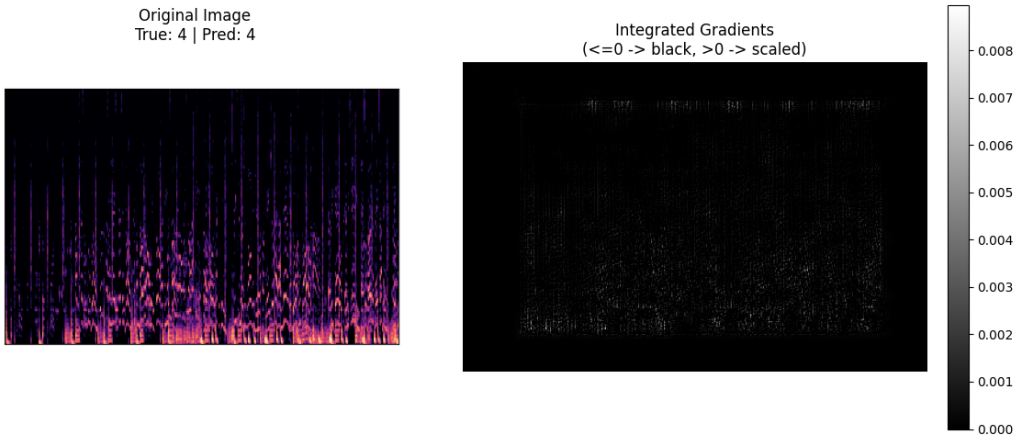
    # 5) Plot side-by-side
    fig, axes = plt.subplots(1, 2, figsize=(12, 5))

    axes[0].imshow(original_img)
    axes[0].set_title(f"Original Image\nTrue: {true_label} | Pred: {pred_label}")
    axes[0].axis("off")

    im = axes[1].imshow(ig_clipped, cmap="gray", vmin=ig_min, vmax=ig_max)
    axes[1].set_title("Integrated Gradients\n(<=0 -> black, >0 -> scaled)")
    axes[1].axis("off")
    fig.colorbar(im, ax=axes[1], fraction=0.046, pad=0.04)

    plt.tight_layout()
    plt.show()
    print(f"Clipped IG Heatmap range: [{ig_min:.4f}, {ig_max:.4f}]")
```

```
loaded_aug_model.to(device)
integrated_gradients_visualization(loaded_aug_model, test_dataset, device, mean, std, 15)
```




```
Clipped IG Heatmap range: [0.0000, 0.0090]
```

9.16. Uploading your Own Music

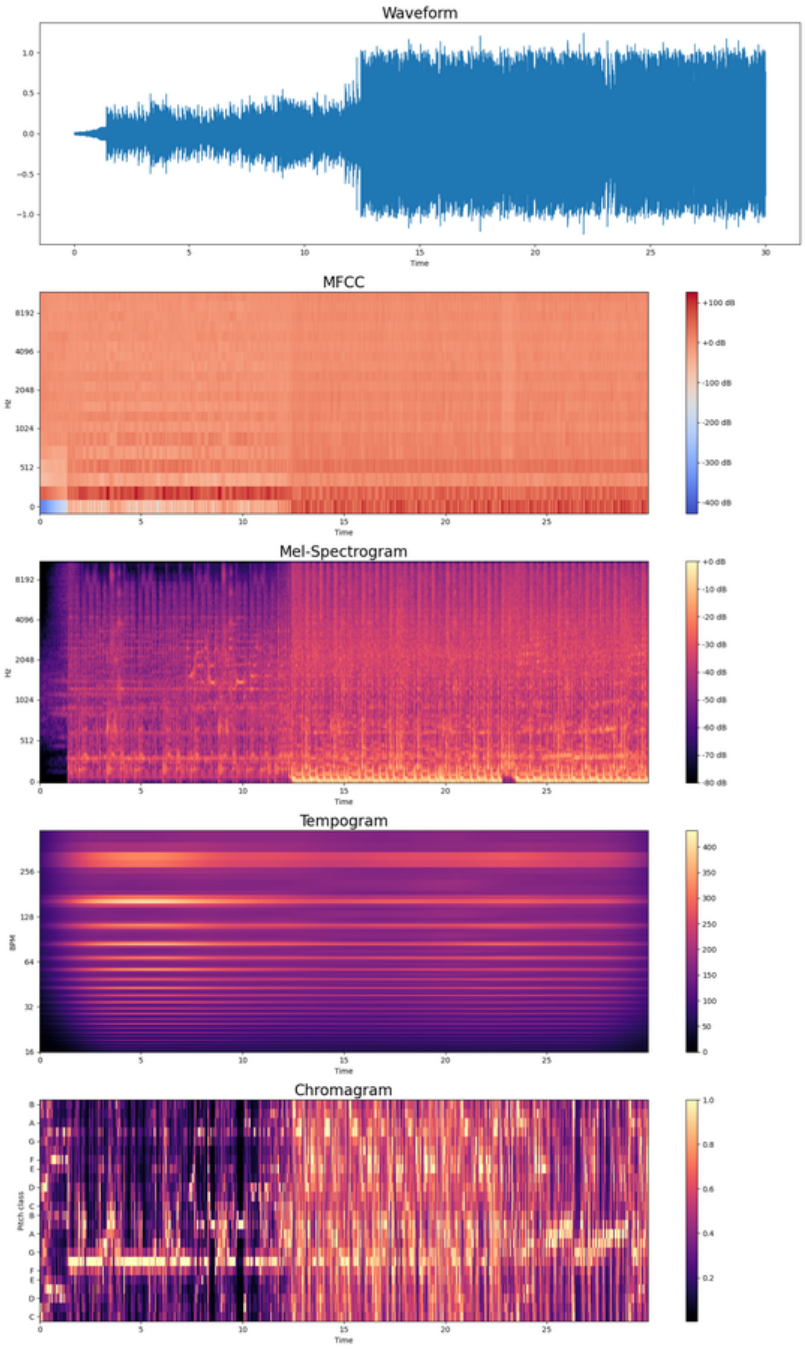
Note: make sure you upload a `.wav` file.

```
from google.colab import files
uploaded = files.upload()
```

```
Saving test.wav to test.wav
```

```
test_audio_path = '/content/test.wav' # Make sure to change this according to your audio file name
display.Audio(test_audio_path)
```

```
y, sample_rate = librosa.load(test_audio_path, duration=30) # Set to 30 seconds
plot_music_representations(y, sample_rate)
```




```
D = np.abs(librosa.stft(y, n_fft=2048, hop_length=512))
print('Shape of D object:', np.shape(D))  # (frequency bins x time frames).

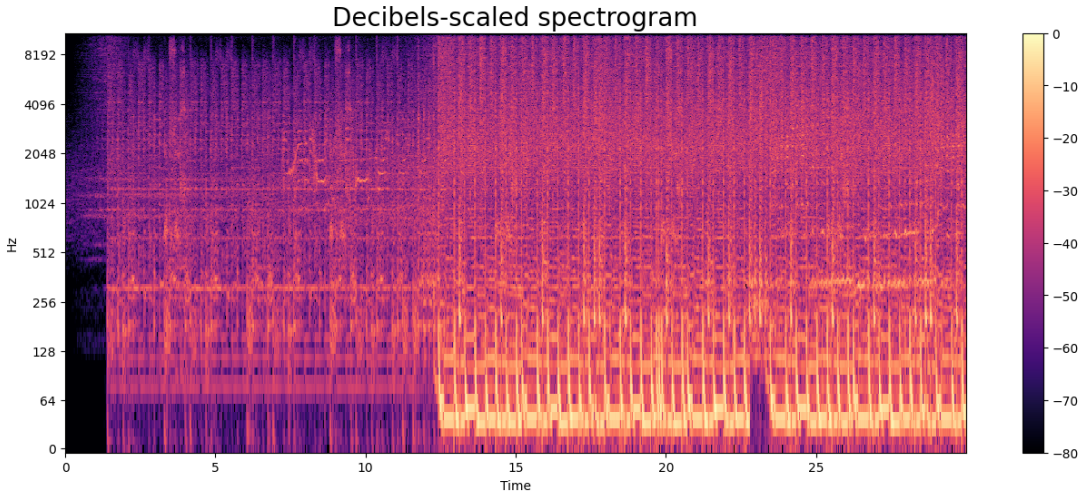
# Convert the amplitude spectrogram to a decibel (dB) scale.
# - ref=np.max: The reference value is the maximum amplitude in D, setting that value to 0 dB.
# This logarithmic conversion is useful because human perception of loudness is logarithmic.
```

```
DB = librosa.amplitude_to_db(D, ref=np.max)

plt.figure(figsize=(16, 6))
librosa.display.specshow(DB, sr=sample_rate, hop_length=512, x_axis='time', y_axis='log')

plt.colorbar()
plt.title('Decibels-scaled spectrogram', fontsize=20)
plt.show()
```

```
Shape of D object: (1025, 1292)
```




```
DB = librosa.amplitude_to_db(D, ref=np.max)
DB_clipped = np.clip(DB, a_min=-80, a_max=0)
norm_DB = (DB_clipped - DB_clipped.min()) / (DB_clipped.max() - DB_clipped.min())

colormap = matplotlib.colormaps.get_cmap('viridis')
rgb_array = colormap(norm_DB)
rgb_array = (rgb_array[..., :3] * 255).astype(np.uint8)

spectrogram_img = Image.fromarray(rgb_array)
```

```
transform_pipeline = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=mean.tolist(), std=std.tolist())
])
input_tensor = transform_pipeline(spectrogram_img).unsqueeze(0)

model.eval()
with torch.no_grad():
    outputs = model(input_tensor.to(device))
    predicted_idx = outputs.argmax(dim=1).item()

probabilities = torch.softmax(outputs, dim=1)
probabilities = probabilities.squeeze(0).tolist()

for idx, prob in enumerate(probabilities):
    print(f"{full_dataset.classes[idx]}: {prob*100:.2f}%")
```

```
blues: 0.00%
classical: 0.00%
country: 0.00%
disco: 0.02%
hiphop: 0.00%
jazz: 0.22%
metal: 0.00%
pop: 0.00%
reggae: 99.77%
rock: 0.00%
```

9.17. Homework (Optional)

1. **Mel vs. linear spectrogram.** Train the same CNN architecture on linear-scaled STFT spectrograms and on Mel spectrograms. Report validation accuracy for each and explain in one sentence why one tends to outperform the other on music.
2. **Genre-pair confusion.** Build the confusion matrix of the trained model. Identify the two most-confused genre pairs and listen to one sample of each to form a hypothesis about why the model confuses them.
3. **SpecAugment [30] ablation.** Train three variants: no augmentation, frequency-masking only, time-masking only. Plot validation accuracy curves on the same figure and comment on which masking mode helps more.
4. **Transfer from a larger audio dataset.** Replace the MusicNet backbone with a `torchvision.models.resnet18` modified for single-channel (or three-channel replicated) spectrogram inputs, initialised from ImageNet weights. Compare final accuracy and training epochs against the from-scratch baseline.
5. **Class-imbalance robustness.** Artificially drop half the training samples from one genre of your choice. Retrain and report per-class accuracy. Discuss what changed and how weighted loss functions could mitigate the degradation.
6. **Inference on a new audio file.** Record or download a 30-second clip in a genre from GTZAN, feed it through the trained model, and display the top-3 predicted genres with their softmax probabilities. Comment on whether the prediction matches the clip's actual genre.

Chapter 10

Transfer Learning

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1Urp9rVDzZPQ8Zdz0PoVKZtaPZuAOwkqF?usp=sharing

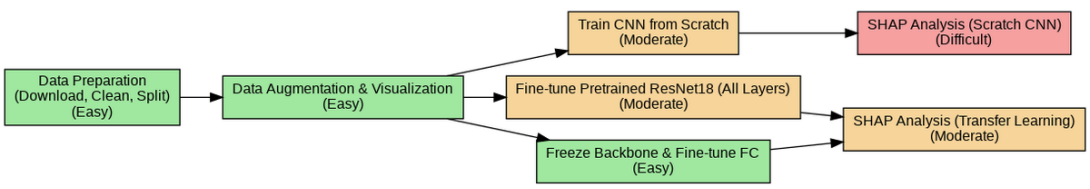
Transfer learning is a method where a pre-trained model, developed on a large and diverse dataset, is repurposed for a new, related task, thereby leveraging previously learned features to improve performance and reduce training time. This approach is particularly advantageous in scenarios with limited domain-specific data, as it enables the model to generalize effectively by building upon established representations.

The problem we're going to solve today is to train a model to classify **ants** and **bees**. We have about 120 training images each for ants and bees. There are 75 validation images for each class. Usually, this is a very small dataset to generalize upon, if trained from scratch. Since we are using transfer learning, we should be able to generalize reasonably well.

10.1. Necessary Imports

```
import time
import os
import copy
import shap
import random
import shutil
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
import torchvision
from IPython.display import Image as IMG
from torch.optim import lr_scheduler
from torch.quantization import FakeQuantize, get_default_qat_qconfig, prepare_qat, convert
from torchvision.models import resnet18, ResNet18_Weights
from torchvision import datasets, models, transforms
from torch.ao.quantization import QuantStub, DeQuantStub, prepare_qat, convert

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```



10.2. Loading the Data

We will use `torchvision` and `torch.utils.data` packages for loading the data. We are gonna use the `wget` bash command to download the data from the official PyTorch website, then uzing `unzip` to unzip it into a `data` folder. We subsequently remove the downloaded zip file for cleanliness.

Since the original data contains corrupted files, we do a short data cleaning utilzing the find command, to delete all non jpg files.

```
%%bash
if [ -d "data" ]; then
    echo "Data folder already exists. Skipping download and extraction."
else
    echo "Data folder does not exist. Downloading and extracting..."
    wget https://download.pytorch.org/tutorial/hymenoptera_data.zip --no-check-certificate --quiet
    unzip -q hymenoptera_data.zip -d data
    rm hymenoptera_data.zip
fi
```

```
Data folder does not exist. Downloading and extracting...
```

```
!find data -type f ! -iname "*.jpg" -delete

!echo "Number of training images (ants):" $(find data/hymenoptera_data/train/ants -type f | wc -l)
!echo "Number of training images (bees):" $(find data/hymenoptera_data/train/bees -type f | wc -l)
!echo ""
!echo "Number of validation images (ants):" $(find data/hymenoptera_data/val/ants -type f | wc -l)
!echo "Number of validation images (bees):" $(find data/hymenoptera_data/val/bees -type f | wc -l)
```

```
Number of training images (ants): 122
Number of training images (bees): 121

Number of validation images (ants): 70
Number of validation images (bees): 83
```

We will establish a **dedicated test set** by relocating 40% of the images from the `validation set` into a newly created `test` directory.

```
val_dir = 'data/hymenoptera_data/val'
test_dir = 'data/hymenoptera_data/test'

for cls in ['ants', 'bees']:
    # for creating the subdirectories inside the test directory
    os.makedirs(os.path.join(test_dir, cls), exist_ok=True)

for cls in ['ants', 'bees']:
    cls_val_dir = os.path.join(val_dir, cls)
    cls_test_dir = os.path.join(test_dir, cls)

    files = [f for f in os.listdir(cls_val_dir) if os.path.isfile(os.path.join(cls_val_dir, f))]
    total = len(files)

    # calculating the number of images to move from val to test
    test_count = int(total * 0.4)
    print(f"Moving {test_count} out of {total} '{cls}' images from val to test.")

    random.shuffle(files)
    selected_files = files[:test_count]
```

```
    # moving utilizing shutil
    for file in selected_files:
        src_path = os.path.join(cls_val_dir, file)
        dst_path = os.path.join(cls_test_dir, file)
        shutil.move(src_path, dst_path)
```

```
Moving 28 out of 70 'ants' images from val to test.
Moving 33 out of 83 'bees' images from val to test.
```

```
!echo "Number of training images (ants):" $(find data/hymenoptera_data/train/ants -type f | wc -l)
!echo "Number of training images (bees):" $(find data/hymenoptera_data/train/bees -type f | wc -l)
!echo ""
!echo "Number of validation images (ants):" $(find data/hymenoptera_data/val/ants -type f | wc -l)
!echo "Number of validation images (bees):" $(find data/hymenoptera_data/val/bees -type f | wc -l)
!echo ""
!echo "Number of test images (ants):" $(find data/hymenoptera_data/test/ants -type f | wc -l)
!echo "Number of test images (bees):" $(find data/hymenoptera_data/test/bees -type f | wc -l)
```

```
Number of training images (ants): 122
Number of training images (bees): 121

Number of validation images (ants): 42
Number of validation images (bees): 50

Number of test images (ants): 28
Number of test images (bees): 33
```

10.3. Data Augmentation

PyTorch's `torchvision.transforms` module provides a comprehensive set of transformations that are commonly used for data augmentation and preprocessing in computer vision tasks. These transforms can be applied to images to enhance model training by artificially expanding the training dataset or by preprocessing the data to better suit the learning algorithms.

Here's a list of different transformation methods provided by PyTorch's `torchvision.transforms`:

10.3.1. Geometric Transformations

- **Resize**: Resize the input image to a specified size.
- **CenterCrop**: Crop the image at the center.
- **RandomCrop**: Crop the image at a random location.
- **RandomResizedCrop**: Crop a random portion of the image and resize it.
- **RandomHorizontalFlip**: Horizontally flip the image with a given probability.
- **RandomVerticalFlip**: Vertically flip the image with a given probability.
- **RandomRotation**: Rotate the image by an angle.
- **RandomAffine**: Apply random affine transformations such as rotation, translation, scaling, and shearing.
- **RandomPerspective**: Apply a random perspective transformation.
- **FiveCrop**: Crop the image into four corners and the central crop.
- **TenCrop**: Similar to `FiveCrop`, but also includes flipped versions of the crops.
- **Pad**: Pad the image on all sides with a specified value.

10.3.2. Color Transformations

- **Grayscale**: Convert the image to grayscale.
- **RandomGrayscale**: Convert the image to grayscale with a given probability.
- **ColorJitter**: Randomly change the brightness, contrast, saturation, and hue of the image.
- **RandomInvert**: Randomly invert the colors of the image.
- **RandomPosterize**: Reduce the number of bits for each color channel.
- **RandomSolarize**: Invert all pixel values above a certain threshold.
- **RandomAdjustSharpness**: Adjust the sharpness of the image.
- **RandomAutocontrast**: Apply an autocontrast operation on the image.
- **RandomEqualize**: Equalize the histogram of the image.

10.3.3. Miscellaneous Transformations

- **GaussianBlur**: Apply Gaussian blur to the image.
- **RandomErasing**: Randomly erase a rectangle region in the image.
- **Normalize**: Normalize a tensor image with mean and standard deviation.
- **ToTensor**: Convert a `PIL.Image` or `numpy.ndarray` to a tensor.
- **Lambda**: Apply a user-defined lambda function as a transformation.

These transforms can be combined using `transforms.Compose` to create complex data augmentation pipelines that are crucial for training robust machine learning models.

For more detailed information, you can explore the official PyTorch documentation.

```
batch_size = 64
mean = [0.485, 0.456, 0.406]  # Mean used in transforms.Normalize
std = [0.229, 0.224, 0.225]   # Std used in transforms.Normalize
data_transforms = {
    'train': transforms.Compose([
        transforms.Resize((256, 256)),  # Resize all images to 256x256
        transforms.RandomHorizontalFlip(),
        transforms.RandomVerticalFlip(),
        transforms.RandomAffine(degrees=(5, 50), translate=(0.1, 0.3), scale=(0.9, 1.1)),
        transforms.RandomPerspective(distortion_scale=0.2),
        transforms.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ]),
    'val': transforms.Compose([
        transforms.Resize((256, 256)),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ]),
    'test': transforms.Compose([
        transforms.Resize((256, 256)),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ]),
}

data_dir = 'data/hymenoptera_data'
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x), data_transforms[x]) for x in ['train',
  'val', 'test']}
```

```
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_size, shuffle=True,
   num_workers=0) for x in ['train', 'val', 'test']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val', 'test']}

class_names = image_datasets['train'].classes

print(f'List of classes: {class_names}')
print(f'Number of training images: {dataset_sizes["train"]}')
print(f'Number of validation images: {dataset_sizes["val"]}')
print(f'Number of test images: {dataset_sizes["test"]}')
```

```
List of classes: ['ants', 'bees']
Number of training images: 243
Number of validation images: 92
Number of test images: 61
```

10.3.4. Visualizing a few images

We will visualize a few training images to inspect the applied data augmentations. A smaller batch size allows for clearer viewing of individual images.

```
def denormalize(tensor, mean, std):
    """
    Denormalizes a tensor normalized by transforms.Normalize.

    Args:
        tensor (torch.Tensor): Normalized tensor.
        mean (list or tuple): Mean values used for normalization (one for each channel).
        std (list or tuple): Std values used for normalization (one for each channel).

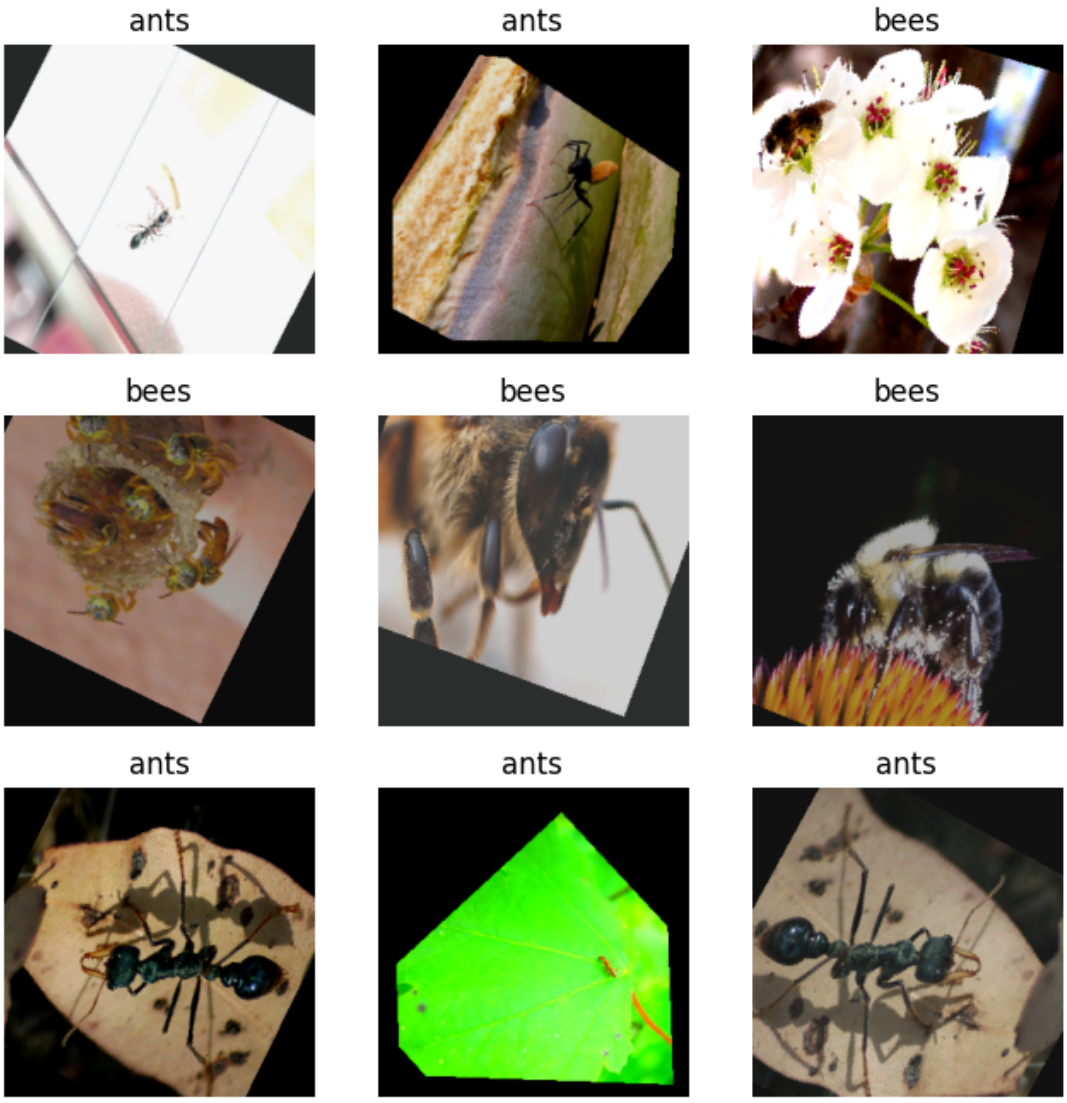
    Returns:
        torch.Tensor: Denormalized tensor.
    """
    if not isinstance(mean, (list, tuple)):
        mean = [mean]
    if not isinstance(std, (list, tuple)):
        std = [std]

    mean = torch.tensor(mean).view(-1, 1, 1)
    std = torch.tensor(std).view(-1, 1, 1)

    return tensor * std + mean
```

```
def plot_images(dataset,mean, std):
    figure = plt.figure(figsize=(8, 8))
    cols, rows = 3, 3
    for i in range(1, cols * rows + 1):
        sample_idx = torch.randint(len(dataset), size=(1,)).item()
        img, label = dataset[sample_idx]
        img = denormalize(img, mean, std) # unnormalize since it was normalized during loading
        npimg = img.numpy()
        img = np.transpose(npimg, (1, 2, 0))
        # Clip the values to ensure they're between [0, 1] for display
        img = np.clip(img, 0, 1)
        figure.add_subplot(rows, cols, i)
        plt.title(image_datasets['train'].classes[label])
        plt.axis("off")
        plt.imshow(img)
    plt.show()
```

```
plot_images(image_datasets['train'],mean, std)
```



10.4. Training and Testing the Model

In this section, we will develop a generic training function that demonstrates two important aspects of model training in PyTorch:

- **Learning Rate Scheduling**: Dynamically adjust the learning rate during training using a scheduler from `torch.optim.lr_scheduler`.
- **Model Checkpointing**: Save the best performing model based on validation metrics to ensure that the optimal weights are preserved.

Note: In the following function, the parameter scheduler is an LR scheduler object from the `torch.optim.lr_scheduler` module.

```python
def train_test_model(model, criterion, optimizer, scheduler, num_epochs=25):
    """
    Train and test a PyTorch model with learning rate scheduling and model checkpointing.

    This function performs training and validation over a specified number of epochs.
    During training, it schedules the learning rate using a provided scheduler and saves
    the best performing model (based on validation accuracy). After training, the best model
    is evaluated on the test set.

    Args:
        model (torch.nn.Module): The neural network model to be trained.
        criterion (callable): The loss function.
        optimizer (torch.optim.Optimizer): The optimizer for model parameters.
        scheduler (torch.optim.lr_scheduler._LRScheduler): Learning rate scheduler.
        num_epochs (int): Number of epochs for training (default is 25).

    Returns:
        model (torch.nn.Module): The trained model with the best validation weights loaded.
        val_acc_h (list): History of validation accuracy for each epoch.
        train_acc_h (list): History of training accuracy for each epoch.
        test_acc (torch.Tensor): Accuracy on the test dataset.
    """
    import time, copy

    since = time.time()
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    val_acc_h = []
    train_acc_h = []

    for epoch in range(num_epochs):
        print(f'Epoch {epoch}/{num_epochs - 1} LR {scheduler.get_last_lr()[0]:.6f}')
        print('-' * 10)

        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()
            else:
                model.eval()

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)

                optimizer.zero_grad()
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
```

```
            if phase == 'train':
                scheduler.step()

            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]
            print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')

            if phase == 'train':
                train_acc_h.append(epoch_acc)
            elif phase == 'val':
                val_acc_h.append(epoch_acc)
                if epoch_acc > best_acc:
                    best_acc = epoch_acc
                    best_model_wts = copy.deepcopy(model.state_dict())

    print('Testing started')
    phase = 'test'
    model.load_state_dict(best_model_wts)
    model.eval()
    running_loss = 0.0
    running_corrects = 0

    with torch.no_grad():
        for inputs, labels in dataloaders[phase]:
            inputs = inputs.to(device)
            labels = labels.to(device)
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            loss = criterion(outputs, labels)
            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

    test_loss = running_loss / dataset_sizes[phase]
    test_acc = running_corrects.double() / dataset_sizes[phase]

    time_elapsed = time.time() - since
    print('Training and testing complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print(f'Best val Acc: {best_acc:.4f}')
    print(f'Test Acc: {test_acc:.4f}')

    return model, val_acc_h, train_acc_h, test_acc
```

10.4.1. Visualizing the model predictions

We will create a generic function for visualizing the performance of our model.

```
def visualize_model(model, dataloaders, device, mean, std, num_images=9):
    """
    Visualizes model predictions by plotting a grid of images from the dataloader,
    displaying the true label, predicted label, and confidence score.

    Args:
        model: Trained PyTorch model.
        dataloaders: Dictionary containing 'train', 'val', and 'test' DataLoaders.
        device: Device to run the model on (e.g., 'cpu' or 'cuda').
        mean: Mean used to normalize the images.
        std: Standard deviation used to normalize the images.
        num_images: Number of images to display in the grid (default: 9).
```

```
    """
    model.eval()
    figure = plt.figure(figsize=(8, 8))
    cols, rows = int(np.sqrt(num_images)), int(np.sqrt(num_images))

    images_shown = 0

    with torch.no_grad():
        for i, (inputs, labels) in enumerate(dataloaders):
            inputs = inputs.to(device)
            labels = labels.to(device)

            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)

            confidences = torch.nn.functional.softmax(outputs, dim=1)

            for j in range(inputs.size(0)):
                if images_shown >= num_images:
                    break

                img = inputs[j].cpu()
                img = denormalize(img, mean, std)

                pred_label = preds[j].item()
                true_label = labels[j].item()
                confidence = confidences[j][pred_label].item()

                npimg = img.numpy()
                img = np.transpose(npimg, (1, 2, 0))

                img = np.clip(img, 0, 1)

                ax = figure.add_subplot(rows, cols, images_shown + 1)
                ax.imshow(img)

                true_class = dataloaders.dataset.classes[true_label]
                pred_class = dataloaders.dataset.classes[pred_label]
                title = f"True: {true_class}\nPred: {pred_class} ({confidence:.2f})"
                ax.set_title(title, fontsize=8)

                plt.axis('off')
                images_shown += 1

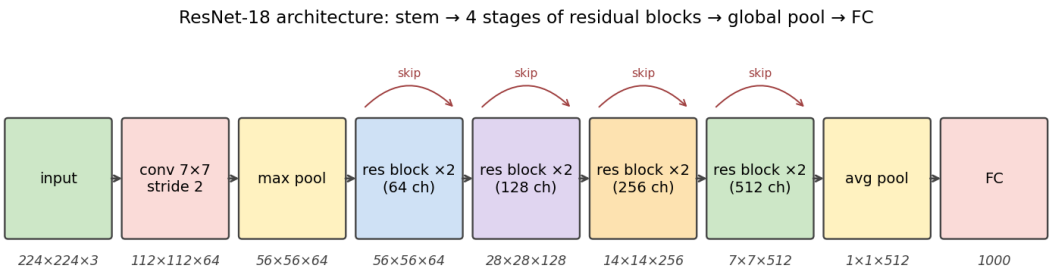
            if images_shown >= num_images:
                break

    plt.show()
```

10.5. Model overview: ResNet18

We will utilize the well-known ResNet18 architecture, using it both to train models from scratch and to fine-tune pre-trained weights for transfer learning. For additional information about **ResNet18**, please refer to this link.

ResNet18 Architecture



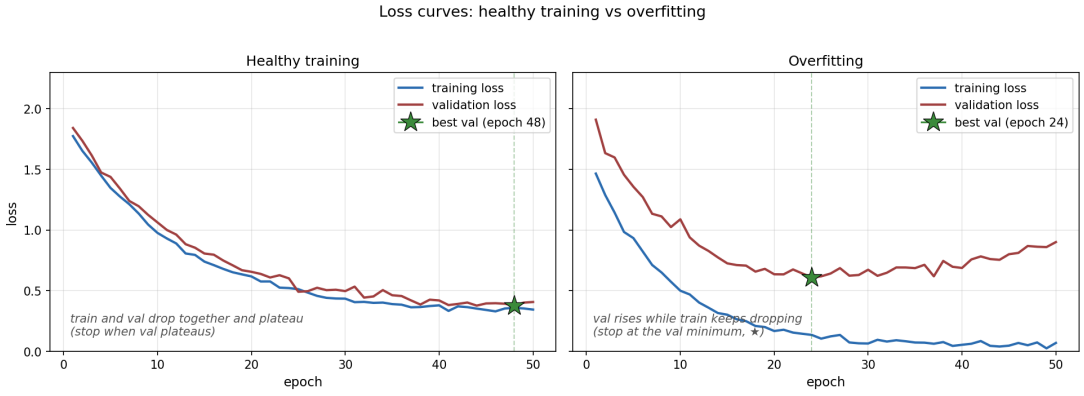

```
model = models.resnet18()
model
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
[... 54 lines omitted ...]
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
```

```
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

10.6. Reading training curves

When you stare at training and validation loss curves, two patterns to recognise:



In the **healthy** case both curves drop together and plateau, and you can stop whenever validation stops improving. In the **overfitting** case validation rises while training keeps falling. The right action is to stop at the validation minimum (green star) and apply more regularisation (weight decay, dropout, augmentation) or use a smaller model.

10.7. Training a CNN from scratch

We are going to initialize a **ResNet18** model and customize it for our classification task. Specifically, we replace the final fully connected layer with a new linear layer that outputs logits for each target class. After moving the model to the appropriate device, we define a cross-entropy loss function and configure an `SGD` optimizer with an initial `learning rate` of 0.001 to update all parameters. Additionally, we create a step learning rate scheduler that decays the `learning rate` by a factor of 0.1 every 5 epochs.

```
model = models.resnet18()
numrs = model.fc.in_features
model.fc = nn.Linear(numrs, len(class_names))
model = model.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized
optimizer = optim.AdamW(model.parameters(), lr=0.001)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
```

```
model, val_acc_h, train_acc_h, test_acc = train_test_model(model, criterion, optimizer, exp_lr_scheduler,
   num_epochs=25)
```

```
Epoch 0/24 LR 0.001000
----------
train Loss: 1.8346 Acc: 0.5309
val Loss: 8.6375 Acc: 0.4565
Epoch 1/24 LR 0.001000
----------
train Loss: 1.0817 Acc: 0.4733
val Loss: 28.2736 Acc: 0.4565
Epoch 2/24 LR 0.001000
----------
train Loss: 0.7321 Acc: 0.5309
val Loss: 8.7784 Acc: 0.4674
Epoch 3/24 LR 0.001000
----------
train Loss: 0.7210 Acc: 0.5103
val Loss: 8.0549 Acc: 0.4565
Epoch 4/24 LR 0.001000
----------
train Loss: 0.6765 Acc: 0.5761
val Loss: 0.9902 Acc: 0.5109
[... 74 lines omitted ...]
train Loss: 0.6392 Acc: 0.6255
val Loss: 0.6511 Acc: 0.6413
Epoch 24/24 LR 0.000001
----------
train Loss: 0.5991 Acc: 0.6502
val Loss: 0.6509 Acc: 0.6304
Testing started
Training and testing complete in 1m 44s
Best val Acc: 0.6522
Test Acc: 0.6066
```

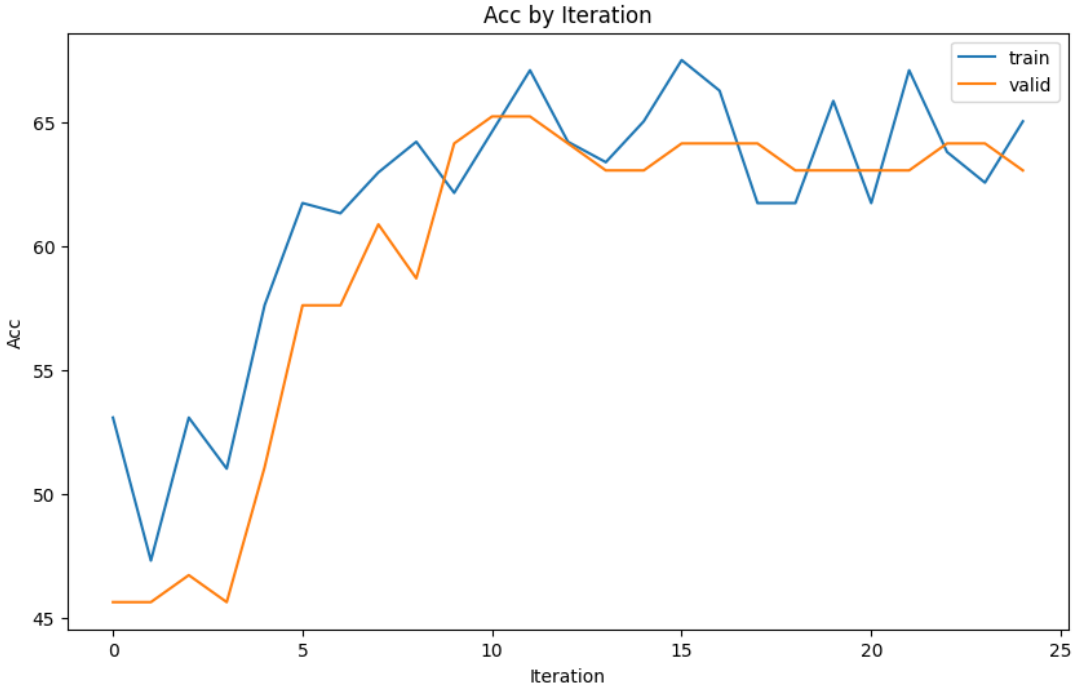
10.8. Saving our Model

```
model_save_path = 'model_weights.pth'

# Save the model's state dictionary
torch.save(model.state_dict(), model_save_path)
```

```
def plot_acc(train_acc_h,val_acc_h):
    plt.figure(figsize=(10,6))
    plt.xlabel('Iteration')
    plt.ylabel('Acc')
    plt.title('Acc by Iteration')
    plt.plot(torch.tensor(train_acc_h, device = 'cpu')*100)
    plt.plot(torch.tensor(val_acc_h, device = 'cpu')*100)
    plt.legend(['train', 'valid'])
```

```
plot_acc(train_acc_h,val_acc_h)
```




```
visualize_model(model, dataloaders['test'],device, mean, std)
```



10.9. SHAP Analysis

In this code block, we prepare a batch of test images for **SHAP visualization**. First, we denormalize the images (undoing the normalization applied during data loading) so that their pixel values return to the `[0, 255]` range. Next, we define a prediction function that converts input numpy images to normalized PyTorch tensors, passes them through our model, and returns the resulting logits. We then set up a SHAP masker tailored for image data and initialize a SHAP explainer using our prediction function and the masker. Finally, we compute and visualize the SHAP values for a couple of test images to interpret our model's predictions.

```
def denormalize_images(image_tensor, mean, std):
    """
    Denormalizes a batch of images from normalized tensor values to the original [0, 255] scale.

    Args:
```

```
        image_tensor (torch.Tensor): A tensor of shape (batch_size, channels, height, width).
        mean (np.ndarray): Array of mean values used for normalization.
        std (np.ndarray): Array of standard deviation values used for normalization.

    Returns:
        np.ndarray: Denormalized images as a numpy array of shape (batch_size, height, width, channels) with
   uint8 type.
    """
    images_np = image_tensor.permute(0, 2, 3, 1).cpu().numpy()
    images_np = (images_np * std + mean) * 255.0
    images_np = np.clip(images_np, 0, 255).astype(np.uint8)

    # Ensure images have 3 channels
    assert images_np.shape[-1] == 3, f"Expected 3-channel images, but got shape {images_np.shape}"
    return images_np

def predict_fn(x, model):
    """
    Prediction function for the SHAP explainer. Converts input images from numpy array to a normalized PyTorch
   tensor,
    feeds them through the model, and returns the logits as a numpy array.

    Args:
        x (np.ndarray): Input images with pixel values in [0, 255] of shape (batch_size, H, W, C).
        model (torch.nn.Module): The PyTorch model for making predictions.

    Returns:
        np.ndarray: Model logits as a numpy array.
    """
    x_tensor = torch.tensor(x, dtype=torch.float32).permute(0, 3, 1, 2) / 255.0
    x_tensor = transforms.Normalize(mean, std)(x_tensor)
    x_tensor = x_tensor.to(device)

    with torch.no_grad():
        logits = model(x_tensor)
    return logits.cpu().numpy()

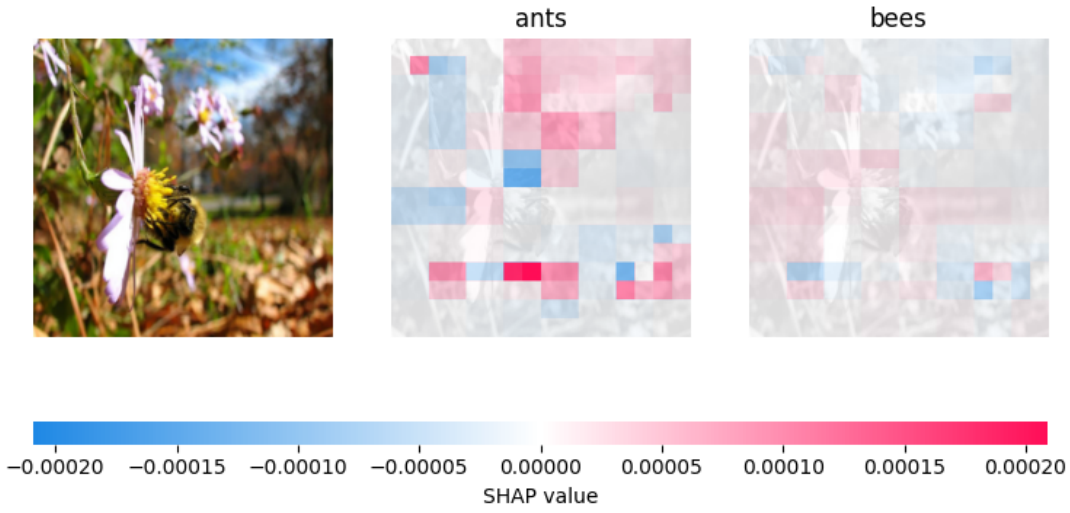
test_images, test_labels = next(iter(dataloaders['test']))
denorm_test_images = denormalize_images(test_images, np.array(mean), np.array(std))

# Create a SHAP masker for images using the shape of a single image
shap_masker = shap.maskers.Image("inpaint_telea", denorm_test_images[0].shape)
# Initialize the SHAP explainer with the model's prediction function and output class names
explainer = shap.Explainer(lambda x: predict_fn(x, model), shap_masker, output_names=class_names)
# Compute SHAP values for the first 1 test image
shap_values = explainer(denorm_test_images[0:1], max_evals=500, batch_size=1)

# Visualize the SHAP explanations for these test images
num_samples = 1
true_label = class_names[test_labels[0]]
opposite_label = 1 - test_labels[0]
opposite_label_name = class_names[opposite_label]
true_labels = np.array([opposite_label_name, true_label])
labels = np.tile(true_labels, (num_samples, 1))
shap.image_plot(shap_values, pixel_values=denorm_test_images[0:1], labels=labels)
```

```
0%|          | 0/498 [00:00<?, ?it/s]
```

```
PartitionExplainer explainer: 2it [00:33, 33.05s/it]
```



10.10. When transfer learning works (and when it does not)

Transfer learning looks like a free lunch when the target dataset is small, fine-tune a backbone pretrained on ImageNet and pick up ten accuracy points with two epochs of training. In practice it works well, poorly, or not at all depending on three factors.

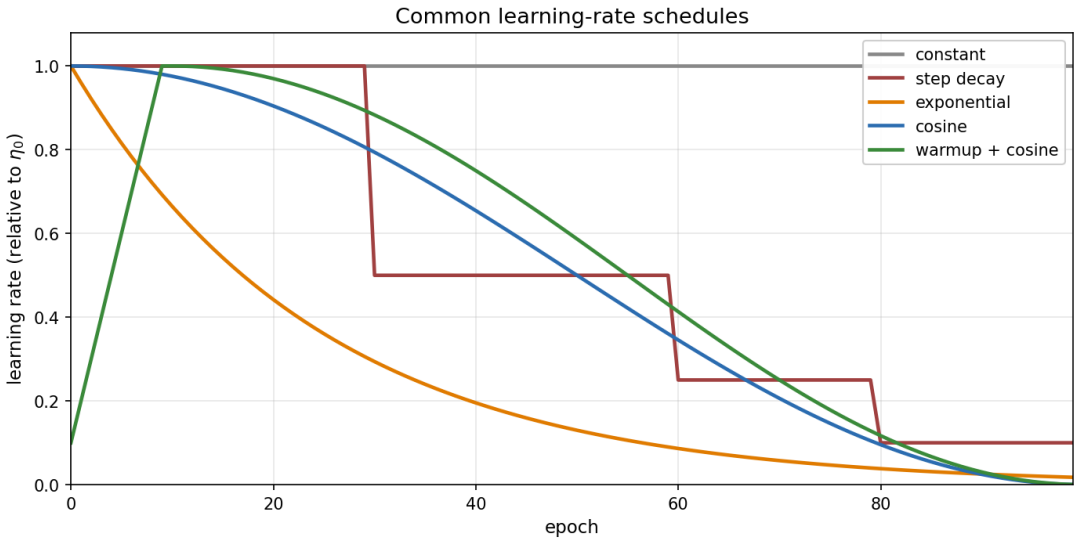
Domain gap. The pretrained model learned features useful for the source distribution. If the target images are visually similar (natural images with similar colour statistics), the early convolutional features transfer almost losslessly. If the target is X-ray scans or satellite imagery, the low-level filters still transfer but the mid-level features are mostly useless, and fine-tuning from ImageNet can actually hurt compared to training from scratch.

Target dataset size. With fewer than a few hundred images per class, freezing the backbone and training only the classification head is usually the safest bet, the head has fewer parameters and is less likely to overfit. With several thousand per class, unfreezing and fine-tuning the entire network typically pays off.

Learning rate and schedule. Fine-tuning almost always needs a learning rate 10-100x smaller than the one you would use for training from scratch, to avoid destroying the pretrained features in the first few steps.

10.11. Learning-rate schedules

Fine-tuning a pretrained model usually wants a **smaller** initial learning rate that decays over training. The most common choices:



Cosine decay (with optional **linear warmup**) is the modern default for vision and NLP. Step decay is simple and works well when you know in advance which epochs need a drop. Exponential decay is rarely the best choice but is included for completeness.

10.12. Finetuning the ResNet18 Model

We will be loading in a pretrained version of **ResNet18** and replacing its final fully connected layer to match our target classes. In this approach, we fine-tune the entire network, both the convolutional layers and the fully connected layers, using the pretrained weights as a starting point.

```
model_ft = resnet18(weights=ResNet18_Weights.IMAGENET1K_V1)
num_ftrs = model_ft.fc.in_features
model_ft.fc = nn.Linear(num_ftrs, 2)
model_ft = model_ft.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized. We are using SGD here, since it is more appropiate for
   fine-tuning tasks.
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)
```

```
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to
/root/.cache/torch/hub/checkpoints/resnet18-f37072fd.pth
100%|##########| 44.7M/44.7M [00:00<00:00, 183MB/s]
```

```
model_ft = resnet18(weights=ResNet18_Weights.IMAGENET1K_V1)

for param in model_ft.parameters():
    param.requires_grad = True

num_ftrs = model_ft.fc.in_features
model_ft.fc = nn.Linear(num_ftrs, 2)
model_ft = model_ft.to(device)
```

```
criterion_ft = nn.CrossEntropyLoss()

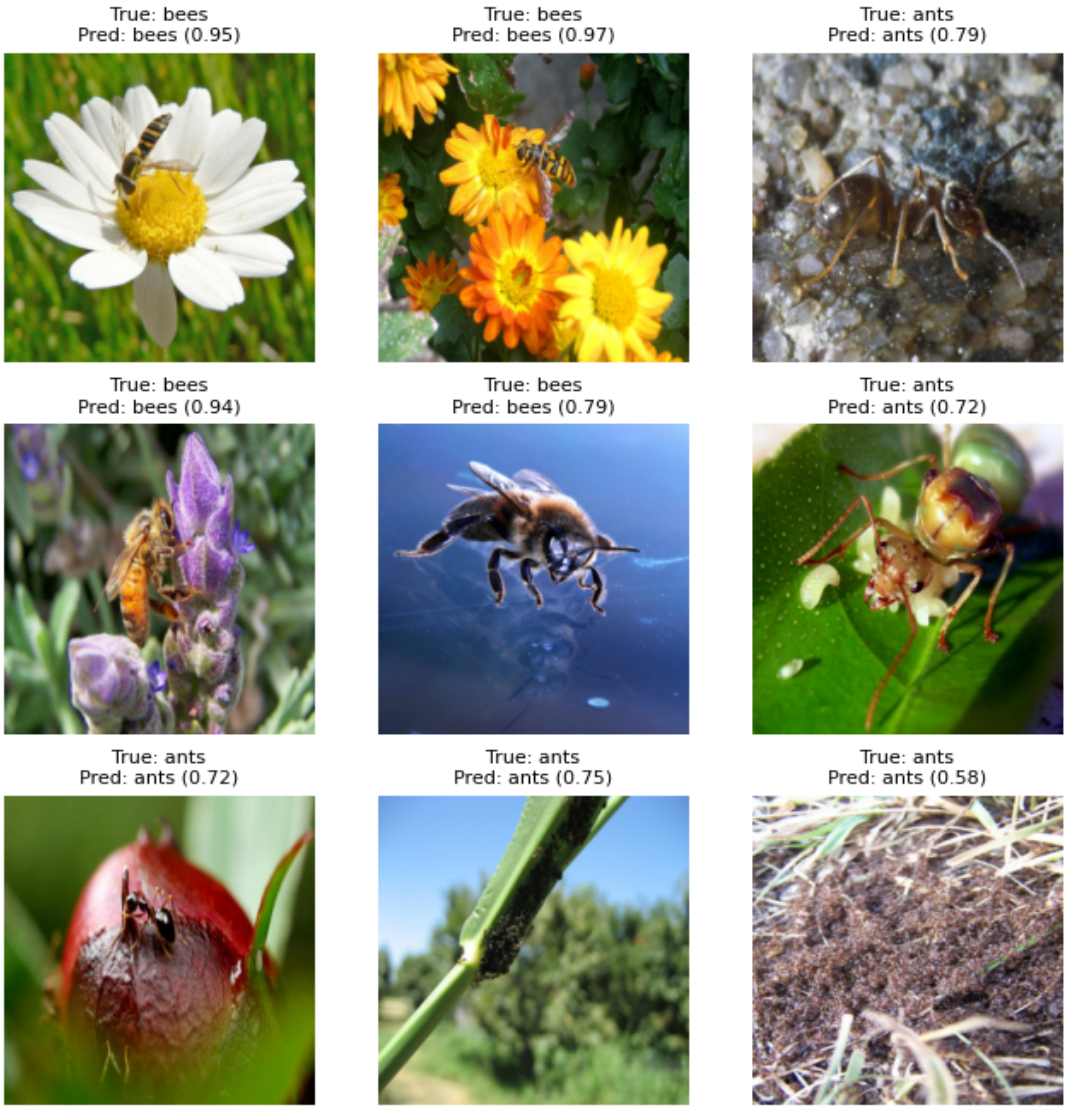
# Observe that all parameters are being optimized. We are using SGD here, since it is more appropiate for
   fine-tuning tasks.
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler_ft = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)
```

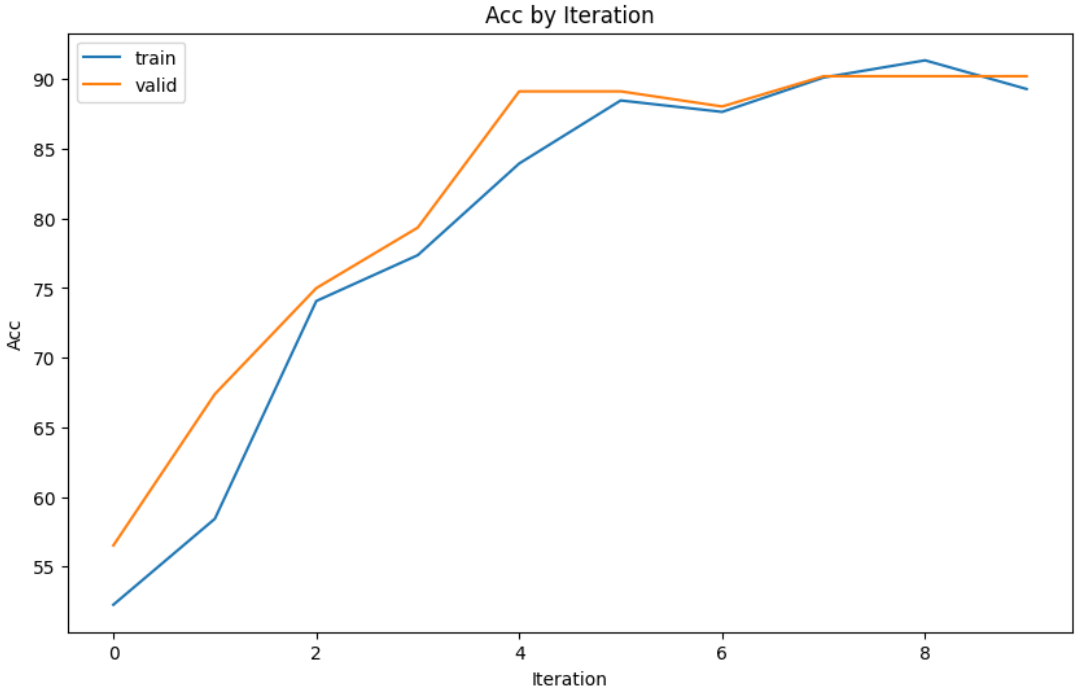
```
model_ft, val_acc_h, train_acc_h, test_acc = train_test_model(model_ft, criterion_ft, optimizer_ft,
   exp_lr_scheduler_ft, num_epochs=10)
```

```
Epoch 0/9 LR 0.001000
----------
train Loss: 0.7501 Acc: 0.5226
val Loss: 0.6968 Acc: 0.5652
Epoch 1/9 LR 0.001000
----------
train Loss: 0.6389 Acc: 0.5844
val Loss: 0.6189 Acc: 0.6739
Epoch 2/9 LR 0.001000
----------
train Loss: 0.5528 Acc: 0.7407
val Loss: 0.5236 Acc: 0.7500
Epoch 3/9 LR 0.001000
----------
train Loss: 0.4974 Acc: 0.7737
val Loss: 0.4483 Acc: 0.7935
Epoch 4/9 LR 0.001000
----------
train Loss: 0.4380 Acc: 0.8395
val Loss: 0.3692 Acc: 0.8913
[... 14 lines omitted ...]
train Loss: 0.2891 Acc: 0.9136
val Loss: 0.3087 Acc: 0.9022
Epoch 9/9 LR 0.000100
----------
train Loss: 0.3014 Acc: 0.8930
val Loss: 0.3076 Acc: 0.9022
Testing started
Training and testing complete in 0m 39s
Best val Acc: 0.9022
Test Acc: 0.9180
```

```
visualize_model(model_ft, dataloaders['test'], device, mean, std)
```




```
plot_acc(train_acc_h, val_acc_h)
```



10.13. Keeping the ResNet18 Backbone Intact

In this step, we freeze all layers of the network except for the final layer by setting `requires_grad = False` for all other parameters. This prevents gradients from being computed for the frozen layers during backpropagation, thereby reducing computational overhead and concentrating the training on the final layer.

You can read more about this in the documentation here: PyTorch Autograd.

```
model_conv = resnet18(weights=ResNet18_Weights.IMAGENET1K_V1)
for param in model_conv.parameters():
    param.requires_grad = False

num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 2)
model_conv = model_conv.to(device)

criterion_conv = nn.CrossEntropyLoss()

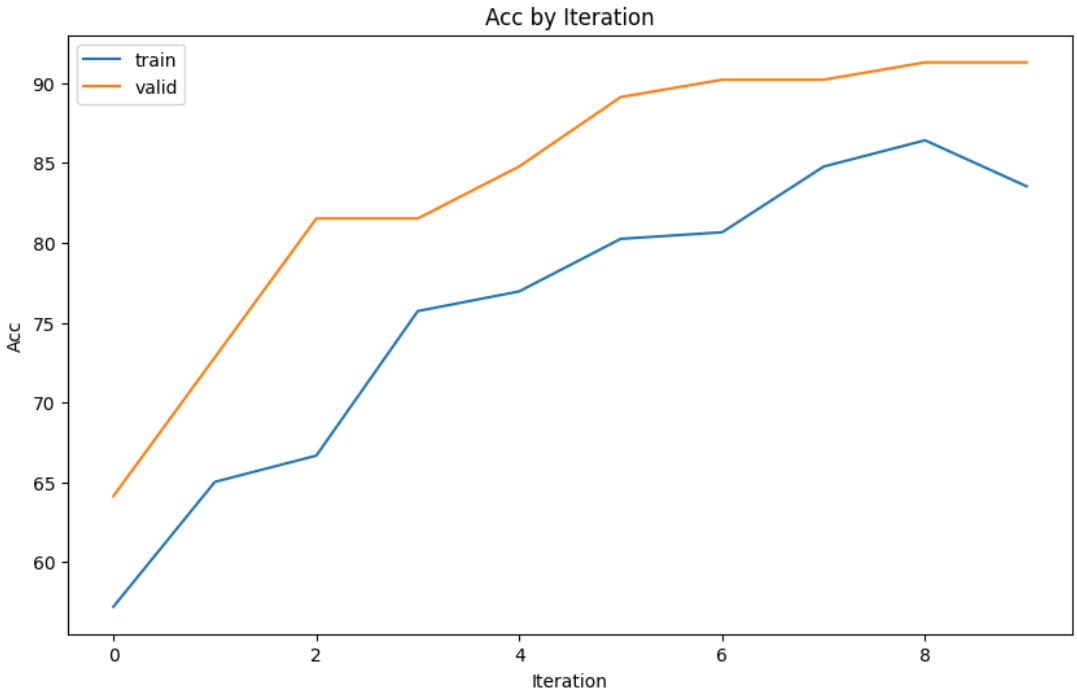
# Observe that only parameters of final layer are being optimized
optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler_conv = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)
```

```
model_conv, val_acc_h, train_acc_h, test_acc = train_test_model(model_conv, criterion_conv, optimizer_conv,
   exp_lr_scheduler_conv, num_epochs=10)
```

```
Epoch 0/9 LR 0.001000
----------
train Loss: 0.6689 Acc: 0.5720
val Loss: 0.6528 Acc: 0.6413
Epoch 1/9 LR 0.001000
----------
train Loss: 0.6506 Acc: 0.6502
val Loss: 0.5878 Acc: 0.7283
Epoch 2/9 LR 0.001000
----------
train Loss: 0.6205 Acc: 0.6667
val Loss: 0.5249 Acc: 0.8152
Epoch 3/9 LR 0.001000
----------
train Loss: 0.5556 Acc: 0.7572
val Loss: 0.4780 Acc: 0.8152
Epoch 4/9 LR 0.001000
----------
train Loss: 0.5249 Acc: 0.7695
val Loss: 0.4258 Acc: 0.8478
[... 14 lines omitted ...]
train Loss: 0.4140 Acc: 0.8642
val Loss: 0.3599 Acc: 0.9130
Epoch 9/9 LR 0.000100
----------
train Loss: 0.4282 Acc: 0.8354
val Loss: 0.3561 Acc: 0.9130
Testing started
Training and testing complete in 0m 35s
Best val Acc: 0.9130
Test Acc: 0.8361
```

```
plot_acc(train_acc_h, val_acc_h)
```



```
visualize_model(model_conv, dataloaders['test'], device, mean, std)
```



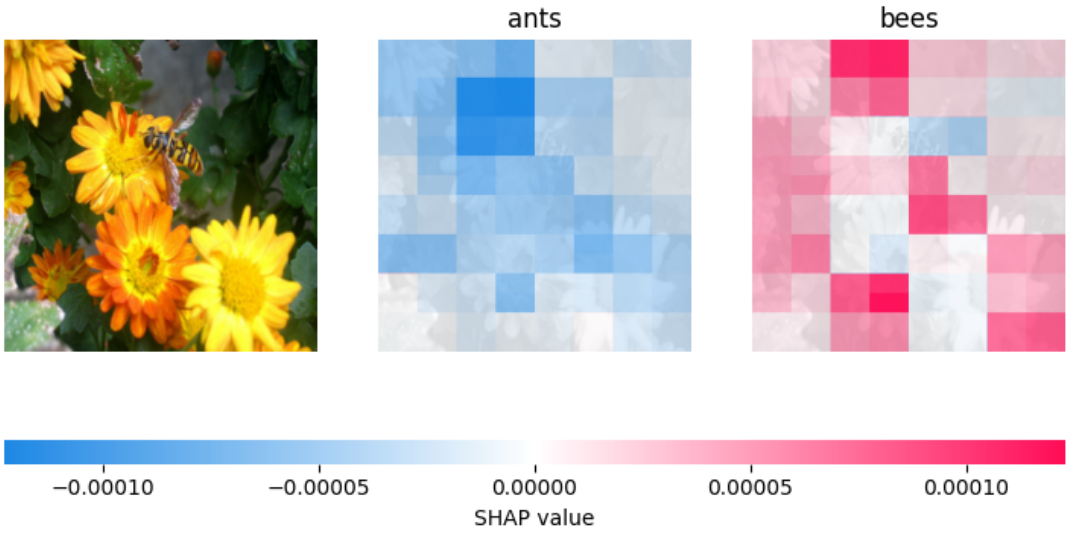

```
test_images, test_labels = next(iter(dataloaders['test']))
denorm_test_images = denormalize_images(test_images, np.array(mean), np.array(std))

shap_masker = shap.maskers.Image("inpaint_telea", denorm_test_images[0].shape)
explainer = shap.Explainer(lambda x: predict_fn(x, model_conv.to(device)), shap_masker,
   output_names=class_names)
shap_values = explainer(denorm_test_images[0:1], max_evals=500, batch_size=1)

# Visualize the SHAP explanations for these test images
num_samples = 1
true_label = class_names[test_labels[0]]
opposite_label = 1 - test_labels[0]
opposite_label_name = class_names[opposite_label]
true_labels = np.array([opposite_label_name, true_label])
labels = np.tile(true_labels, (num_samples, 1))
shap.image_plot(shap_values, pixel_values=denorm_test_images[0:1], labels=labels)
```

```
0%|          | 0/498 [00:00<?, ?it/s]
```

```
PartitionExplainer explainer: 2it [00:26, 26.51s/it]
```



10.14. Another Way to Finetune the Entire Network

```
model_conv_ft = model_conv
for param in model_conv_ft.parameters():
    param.requires_grad = True

model_conv_ft = model_conv_ft.to(device)

criterion_conv_ft = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized
optimizer_conv_ft = optim.SGD(model_conv_ft.parameters(), lr=0.0001, momentum=0.99)

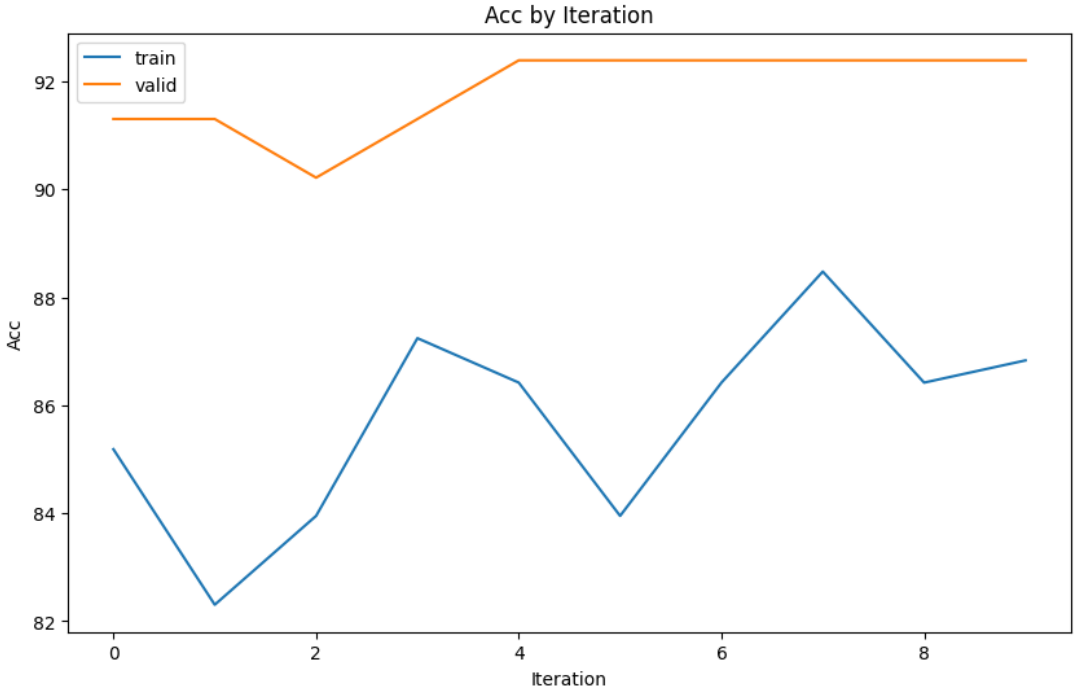
# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler_conv_ft = lr_scheduler.StepLR(optimizer_conv_ft, step_size=7, gamma=0.1)
```

```
model_conv_ft, val_acc_h, train_acc_h, test_acc = train_test_model(model_conv_ft, criterion_conv_ft,
   optimizer_conv_ft, exp_lr_scheduler_conv_ft, num_epochs=10)
```

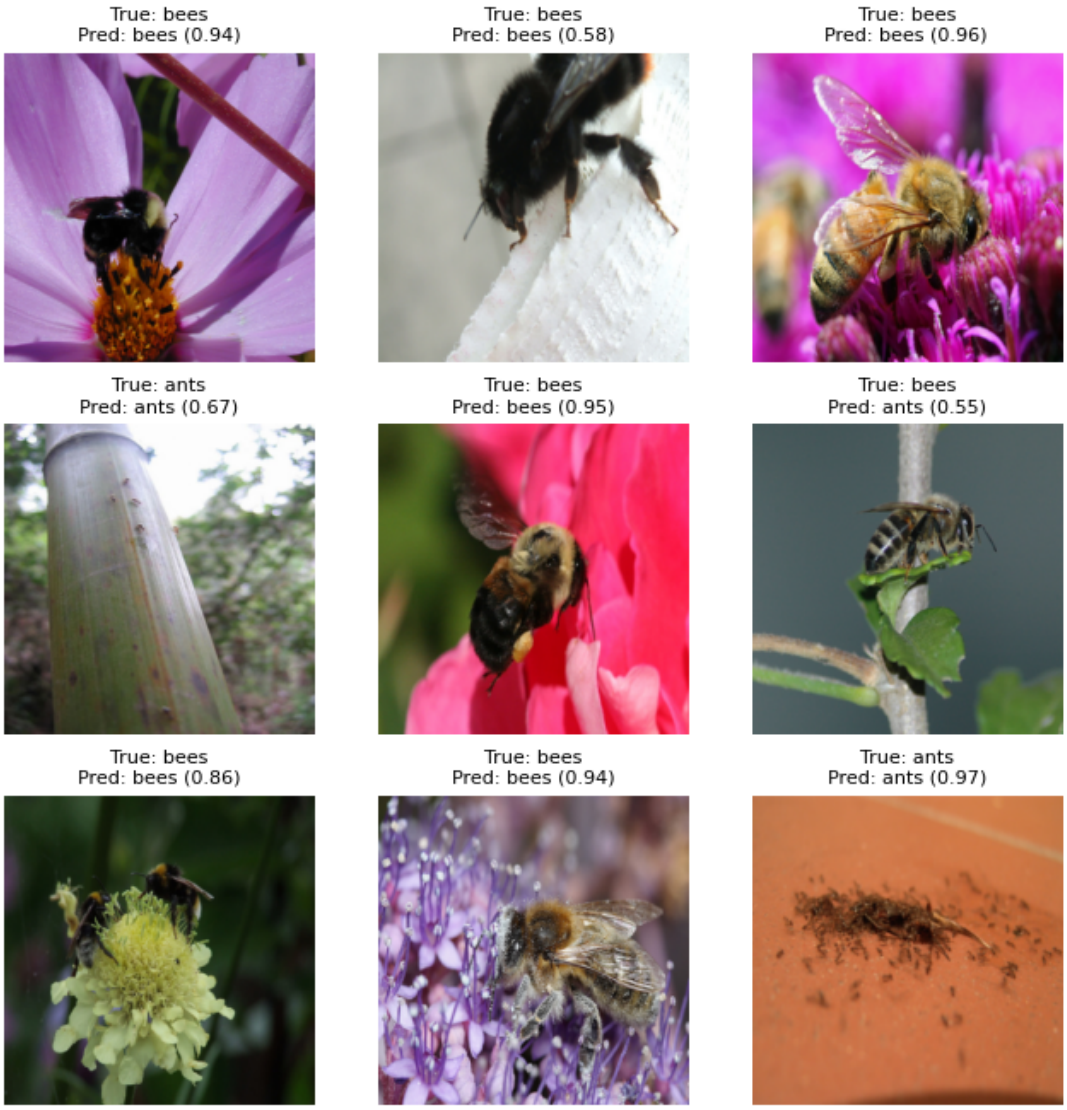
```
Epoch 0/9 LR 0.000100
----------
train Loss: 0.4306 Acc: 0.8519
val Loss: 0.3559 Acc: 0.9130
Epoch 1/9 LR 0.000100
----------
train Loss: 0.4201 Acc: 0.8230
val Loss: 0.3541 Acc: 0.9130
Epoch 2/9 LR 0.000100
----------
train Loss: 0.4331 Acc: 0.8395
val Loss: 0.3481 Acc: 0.9022
Epoch 3/9 LR 0.000100
```

```
----------
train Loss: 0.3845 Acc: 0.8724
val Loss: 0.3420 Acc: 0.9130
Epoch 4/9 LR 0.000100
----------
train Loss: 0.3996 Acc: 0.8642
val Loss: 0.3335 Acc: 0.9239
[... 14 lines omitted ...]
train Loss: 0.3742 Acc: 0.8642
val Loss: 0.3136 Acc: 0.9239
Epoch 9/9 LR 0.000010
----------
train Loss: 0.3690 Acc: 0.8683
val Loss: 0.3127 Acc: 0.9239
Testing started
Training and testing complete in 0m 38s
Best val Acc: 0.9239
Test Acc: 0.8525
```

```
plot_acc(train_acc_h, val_acc_h)
```




```
visualize_model(model_conv_ft, dataloaders['test'], device, mean, std)
```



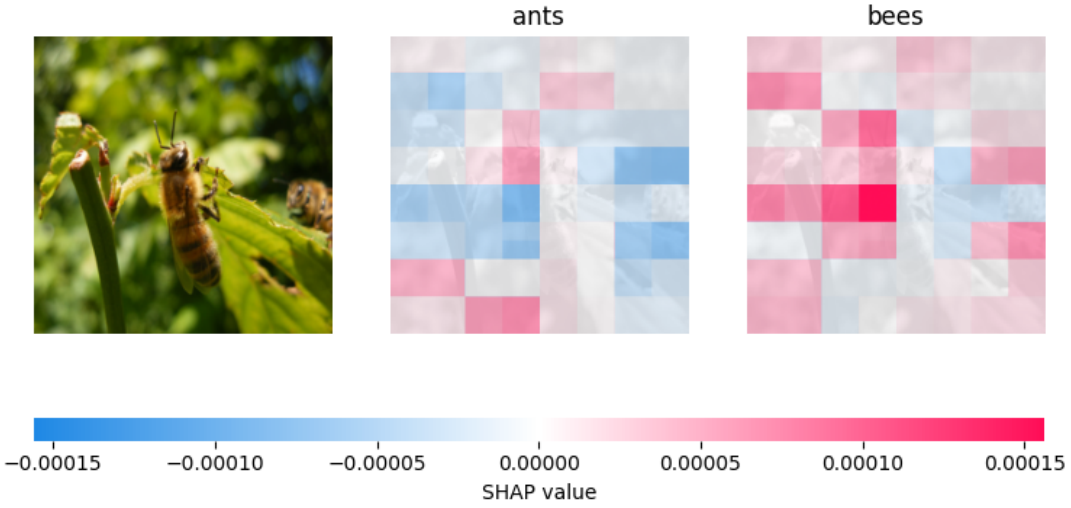

```
test_images, test_labels = next(iter(dataloaders['test']))
denorm_test_images = denormalize_images(test_images, np.array(mean), np.array(std))

shap_masker = shap.maskers.Image("inpaint_telea", denorm_test_images[0].shape)
explainer = shap.Explainer(lambda x: predict_fn(x, model_conv_ft), shap_masker, output_names=class_names)
shap_values = explainer(denorm_test_images[0:1], max_evals=500, batch_size=1)

# Visualize the SHAP explanations for these test images
num_samples = 1
true_label = class_names[test_labels[0]]
opposite_label = 1 - test_labels[0]
opposite_label_name = class_names[opposite_label]
true_labels = np.array([opposite_label_name, true_label])
labels = np.tile(true_labels, (num_samples, 1))
shap.image_plot(shap_values, pixel_values=denorm_test_images[0:1], labels=labels)
```

```
0%|          | 0/498 [00:00<?, ?it/s]
```

```
PartitionExplainer explainer: 2it [00:24, 24.60s/it]
```



10.15. Homework (Optional)

1. **Freeze vs. fine-tune comparison.** Train three variants on the Hymenoptera dataset: (a) train from scratch, (b) fine-tune the full pretrained ResNet-18, (c) freeze all but the final layer. Report test accuracy and training time for each. Argue which approach is best for this dataset.
2. **Learning rate for fine-tuning.** Fine-tune a pretrained ResNet-18 with three learning rates: `1e-2`, `1e-3`, `1e-4`. Report validation accuracy curves. Explain why a much smaller learning rate than pretraining is recommended.
3. **Discriminative learning rates.** Use `torch.optim.SGD([ {'params': base.parameters(), 'lr': 1e-4}, {'params': head.parameters(), 'lr': 1e-3} ])` so the backbone trains more slowly than the head. Compare with a single global learning rate.
4. **Partial unfreezing.** Freeze all layers, then incrementally unfreeze `layer3`, `layer2`, `layer1`, `conv1`, one at a time, training for two epochs each. Plot accuracy vs. layers-unfrozen. At what depth does unfreezing stop helping?
5. **Different backbone.** Swap ResNet-18 for a smaller MobileNetV3 and a larger ResNet-50. Report test accuracy and parameter count for all three. Discuss the capacity vs. overfit trade-off on this small dataset.
6. **SHAP interpretability.** For a correctly classified and a misclassified test image each, generate SHAP explanations and identify which pixels drove the prediction. Comment on whether the model is attending to sensible image regions.

Chapter 11

Quantization Techniques

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1iYEshq_vYtBHVYnlA8xzPOEMOIFWQ78W?usp=sharing

11.1. Choosing between dynamic, static, and quantization-aware training

PyTorch ships three distinct quantization workflows. Picking the right one saves hours of debugging and avoids an accuracy cliff.

Dynamic quantization is the simplest: weights are converted to INT8 ahead of time, activations are quantized on-the-fly at inference. Works well for models dominated by linear and LSTM layers, transformers, language models, NLP backbones, where matrix multiplies are the bottleneck. Convolutional networks get almost no benefit because dynamic quantization does not currently support `Conv2d`.

Static (post-training) quantization computes activation statistics on a calibration set, then bakes them into the model. Works well for convolutional networks and requires a small representative calibration dataset (a few hundred images is enough). Accuracy loss is typically 1-3% versus the float baseline.

Quantization-aware training (QAT) inserts fake-quantize operators into the graph at training time, so the model learns to be robust to the quantization error. More expensive (you retrain), but gives the smallest accuracy gap, often within 0.5% of the float baseline. Use it when static quantization drops accuracy unacceptably.

11.2. Quantization in PyTorch

Project: Transfer Learning

Course: Deep Network Development

Authors: Tamás Takács, Imre Molnár (PhD students, Department of Artificial Intelligence, Eötvös Loránd University)

The problem we're going to solve today is to train a model to classify **ants** and **bees** utilizing a pre-trained **ResNet-18 [13]** model and **quantization techniques**. We have about 120 training images each for ants and bees. There are 75 validation images for each class. Usually, this is a very small dataset to generalize upon, if trained from scratch. Since we are using transfer learning, we should be able to generalize reasonably well.

Quantization is an important technique for optimizing neural networks, particularly for deployment on resource-constrained devices such as mobile phones and edge devices. PyTorch offers several methods for quantizing models, each suited to different use cases.

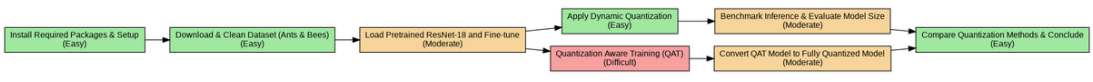
The three main types of quantization in PyTorch are:

1. **Dynamic Quantization**: In dynamic quantization, only the model weights are quantized (typically from 32-bit floating point to 8-bit integers) at runtime. This form of quantization is relatively simple to implement and is effective in models with large fully connected layers, such as NLP models like LSTMs and Transformers.
2. **Static Quantization (Post-Training Quantization)**: Both weights and activations are quantized statically. This method requires calibration on a sample dataset after training. It is beneficial for models with both convolutional and fully connected layers. Static quantization improves performance by reducing the precision of activations and weights but typically requires calibration data to adjust the scale.
3. **Quantization Aware Training (QAT)**: In QAT, quantization is simulated during training. The model learns to adjust its weights and activations to account for the reduced precision. This approach generally provides the best post-quantization accuracy and is especially effective in models where maintaining accuracy is critical, such as for vision tasks or when quantizing convolutional layers.

11.3. Necessary Imports

```
import time
import os
import copy
import random
import shutil
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import numpy as np
import torchvision
from IPython.display import Image as IMG
from torch.optim import lr_scheduler
from torch.quantization import FakeQuantize, get_default_qat_qconfig, prepare_qat, convert
from torchvision.models import resnet18, ResNet18_Weights
from torchvision import datasets, models, transforms
from torch.ao.quantization import QuantStub, DeQuantStub, prepare_qat, convert

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```



11.4. Loading the Data

We will use `torchvision` and `torch.utils.data` packages for loading the data. We are gonna use the `wget` bash command to download the data from the official PyTorch website, then uzing `unzip` to unzip it into a `data` folder. We subsequently remove the downloaded zip file for cleanliness.

Since the original data contains corrupted files, we do a short data cleaning utilzing the find command, to delete all non jpg files.

```
%%bash
if [ -d "data" ]; then
    echo "Data folder already exists. Skipping download and extraction."
else
    echo "Data folder does not exist. Downloading and extracting..."
    wget https://download.pytorch.org/tutorial/hymenoptera_data.zip --no-check-certificate --quiet
    unzip -q hymenoptera_data.zip -d data
    rm hymenoptera_data.zip
fi
```

```
Data folder does not exist. Downloading and extracting...
```

```
!find data -type f ! -iname "*.jpg" -delete

!echo "Number of training images (ants):" $(find data/hymenoptera_data/train/ants -type f | wc -l)
!echo "Number of training images (bees):" $(find data/hymenoptera_data/train/bees -type f | wc -l)
!echo ""
!echo "Number of validation images (ants):" $(find data/hymenoptera_data/val/ants -type f | wc -l)
!echo "Number of validation images (bees):" $(find data/hymenoptera_data/val/bees -type f | wc -l)
```

```
Number of training images (ants): 122
Number of training images (bees): 121

Number of validation images (ants): 70
Number of validation images (bees): 83
```

We will establish a **dedicated test set** by relocating 40% of the images from the `validation set` into a newly created `test` directory.

```
val_dir = 'data/hymenoptera_data/val'
test_dir = 'data/hymenoptera_data/test'

for cls in ['ants', 'bees']:
    # for creating the subdirectories inside the test directory
    os.makedirs(os.path.join(test_dir, cls), exist_ok=True)

for cls in ['ants', 'bees']:
    cls_val_dir = os.path.join(val_dir, cls)
    cls_test_dir = os.path.join(test_dir, cls)

    files = [f for f in os.listdir(cls_val_dir) if os.path.isfile(os.path.join(cls_val_dir, f))]
    total = len(files)

    # calculating the number of images to move from val to test
    test_count = int(total * 0.4)
    print(f"Moving {test_count} out of {total} '{cls}' images from val to test.")

    random.shuffle(files)
    selected_files = files[:test_count]
```

```
    # moving utilizing shutil
    for file in selected_files:
        src_path = os.path.join(cls_val_dir, file)
        dst_path = os.path.join(cls_test_dir, file)
        shutil.move(src_path, dst_path)
```

```
Moving 28 out of 70 'ants' images from val to test.
Moving 33 out of 83 'bees' images from val to test.
```

```
!echo "Number of training images (ants):" $(find data/hymenoptera_data/train/ants -type f | wc -l)
!echo "Number of training images (bees):" $(find data/hymenoptera_data/train/bees -type f | wc -l)
!echo ""
!echo "Number of validation images (ants):" $(find data/hymenoptera_data/val/ants -type f | wc -l)
!echo "Number of validation images (bees):" $(find data/hymenoptera_data/val/bees -type f | wc -l)
!echo ""
!echo "Number of test images (ants):" $(find data/hymenoptera_data/test/ants -type f | wc -l)
!echo "Number of test images (bees):" $(find data/hymenoptera_data/test/bees -type f | wc -l)
```

```
Number of training images (ants): 122
Number of training images (bees): 121

Number of validation images (ants): 42
Number of validation images (bees): 50

Number of test images (ants): 28
Number of test images (bees): 33
```

11.5. Data Augmentation

PyTorch's `torchvision.transforms` module provides a comprehensive set of transformations that are commonly used for data augmentation and preprocessing in computer vision tasks. These transforms can be applied to images to enhance model training by artificially expanding the training dataset or by preprocessing the data to better suit the learning algorithms.

Here's a list of different transformation methods provided by PyTorch's `torchvision.transforms`:

11.5.1. Geometric Transformations

- **Resize**: Resize the input image to a specified size.
- **CenterCrop**: Crop the image at the center.
- **RandomCrop**: Crop the image at a random location.
- **RandomResizedCrop**: Crop a random portion of the image and resize it.
- **RandomHorizontalFlip**: Horizontally flip the image with a given probability.
- **RandomVerticalFlip**: Vertically flip the image with a given probability.
- **RandomRotation**: Rotate the image by an angle.
- **RandomAffine**: Apply random affine transformations such as rotation, translation, scaling, and shearing.
- **RandomPerspective**: Apply a random perspective transformation.
- **FiveCrop**: Crop the image into four corners and the central crop.
- **TenCrop**: Similar to `FiveCrop`, but also includes flipped versions of the crops.
- **Pad**: Pad the image on all sides with a specified value.

11.5.2. Color Transformations

- **Grayscale**: Convert the image to grayscale.
- **RandomGrayscale**: Convert the image to grayscale with a given probability.
- **ColorJitter**: Randomly change the brightness, contrast, saturation, and hue of the image.
- **RandomInvert**: Randomly invert the colors of the image.
- **RandomPosterize**: Reduce the number of bits for each color channel.
- **RandomSolarize**: Invert all pixel values above a certain threshold.
- **RandomAdjustSharpness**: Adjust the sharpness of the image.
- **RandomAutocontrast**: Apply an autocontrast operation on the image.
- **RandomEqualize**: Equalize the histogram of the image.

11.5.3. Miscellaneous Transformations

- **GaussianBlur**: Apply Gaussian blur to the image.
- **RandomErasing**: Randomly erase a rectangle region in the image.
- **Normalize**: Normalize a tensor image with mean and standard deviation.
- **ToTensor**: Convert a `PIL.Image` or `numpy.ndarray` to a tensor.
- **Lambda**: Apply a user-defined lambda function as a transformation.

These transforms can be combined using `transforms.Compose` to create complex data augmentation pipelines that are crucial for training robust machine learning models.

For more detailed information, you can explore the official PyTorch documentation.

```
batch_size = 64
mean = [0.485, 0.456, 0.406]  # Mean used in transforms.Normalize
std = [0.229, 0.224, 0.225]   # Std used in transforms.Normalize
data_transforms = {
    'train': transforms.Compose([
        transforms.Resize((256, 256)),  # Resize all images to 256x256
        transforms.RandomHorizontalFlip(),
        transforms.RandomVerticalFlip(),
        transforms.RandomAffine(degrees=(5, 50), translate=(0.1, 0.3), scale=(0.9, 1.1)),
        transforms.RandomPerspective(distortion_scale=0.2),
        transforms.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ]),
    'val': transforms.Compose([
        transforms.Resize((256, 256)),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ]),
    'test': transforms.Compose([
        transforms.Resize((256, 256)),
        transforms.ToTensor(),
        transforms.Normalize(mean, std)
    ]),
}

data_dir = 'data/hymenoptera_data'
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x), data_transforms[x]) for x in ['train',
  'val', 'test']}
```

```
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size=batch_size, shuffle=True,
   num_workers=0) for x in ['train', 'val', 'test']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val', 'test']}

class_names = image_datasets['train'].classes

print(f'List of classes: {class_names}')
print(f'Number of training images: {dataset_sizes["train"]}')
print(f'Number of validation images: {dataset_sizes["val"]}')
print(f'Number of test images: {dataset_sizes["test"]}')
```

```
List of classes: ['ants', 'bees']
Number of training images: 243
Number of validation images: 92
Number of test images: 61
```

11.5.4. Visualizing a few images

We will visualize a few training images to inspect the applied data augmentations. A smaller batch size allows for clearer viewing of individual images.

```
def denormalize(tensor, mean, std):
    """
    Denormalizes a tensor normalized by transforms.Normalize.

    Args:
        tensor (torch.Tensor): Normalized tensor.
        mean (list or tuple): Mean values used for normalization (one for each channel).
        std (list or tuple): Std values used for normalization (one for each channel).

    Returns:
        torch.Tensor: Denormalized tensor.
    """
    # Check if mean and std are single values, in case of grayscale images
    if not isinstance(mean, (list, tuple)):
        mean = [mean]
    if not isinstance(std, (list, tuple)):
        std = [std]

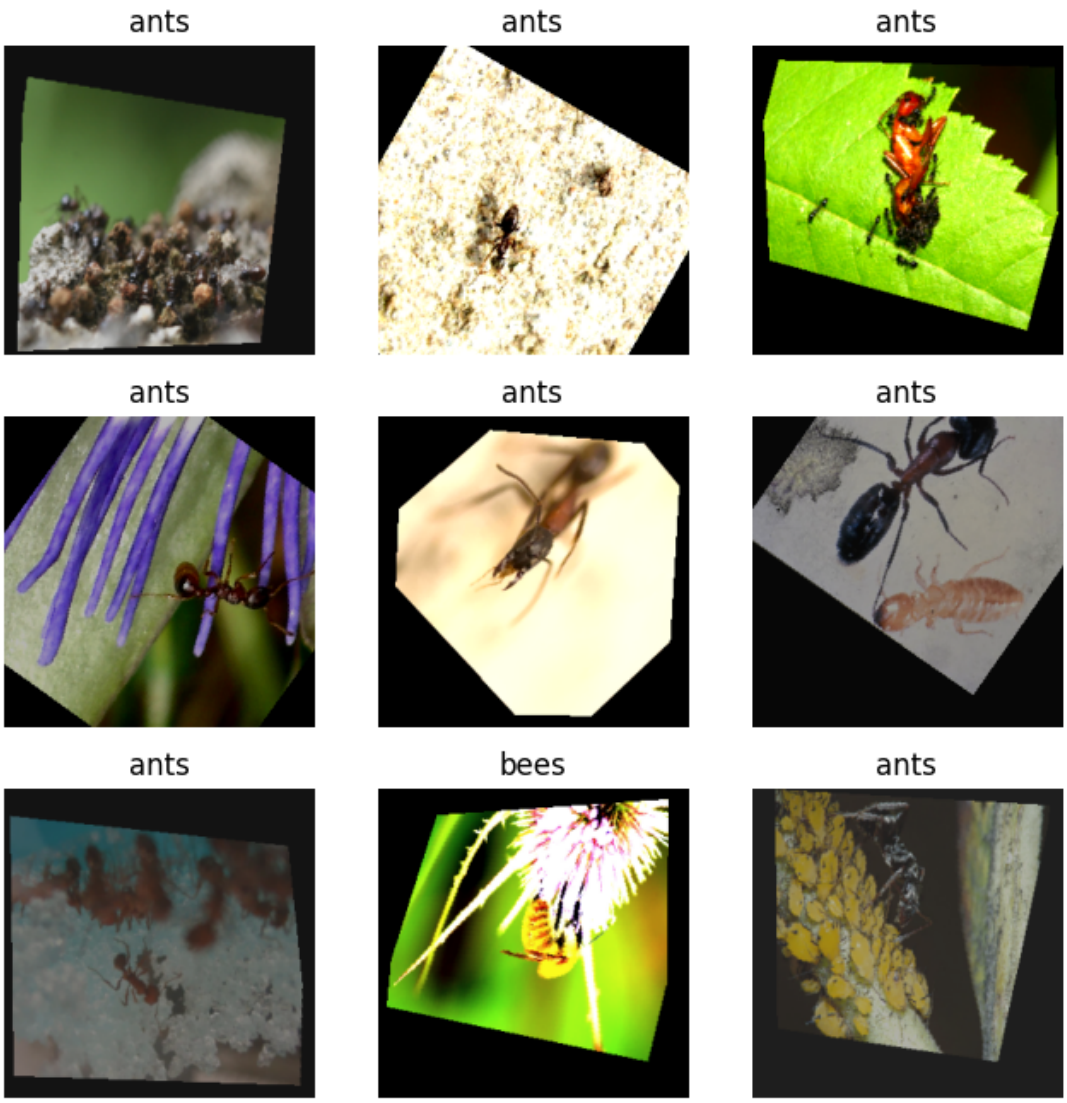
    mean = torch.tensor(mean).view(-1, 1, 1)
    std = torch.tensor(std).view(-1, 1, 1)

    return tensor * std + mean
```

```
def plot_images(dataset,mean, std):
    figure = plt.figure(figsize=(8, 8))
    cols, rows = 3, 3
    for i in range(1, cols * rows + 1):
        sample_idx = torch.randint(len(dataset), size=(1,)).item()
        img, label = dataset[sample_idx]
        img = denormalize(img, mean, std) # unnormalize since it was normalized during loading
        npimg = img.numpy()
        img = np.transpose(npimg, (1, 2, 0))
        # Clip the values to ensure they're between [0, 1] for display
        img = np.clip(img, 0, 1)
        figure.add_subplot(rows, cols, i)
        plt.title(image_datasets['train'].classes[label])
        plt.axis("off")
```

```
        plt.imshow(img)
    plt.show()
```

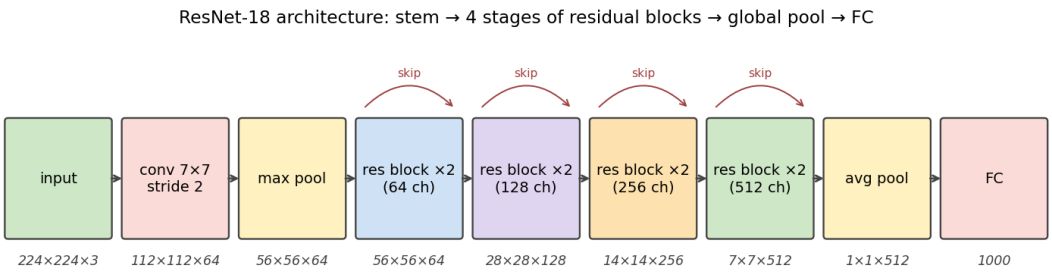
```
plot_images(image_datasets['train'],mean, std)
```



11.6. Model overview: ResNet18

We will utilize the well-known ResNet18 architecture, using it both to train models from scratch and to fine-tune pre-trained weights for transfer learning. For additional information about **ResNet18**, please refer to this link.

ResNet18 Architecture




```
model = models.resnet18()
model
```

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
[... 54 lines omitted ...]
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
```

```
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=1000, bias=True)
)
```

11.7. Finetuning the ResNet18 Model

We will be loading in a pretrained version of **ResNet18** and replacing its final fully connected layer to match our target classes. In this approach, we fine-tune the entire network, both the convolutional layers and the fully connected layers, using the pretrained weights as a starting point.

```python
def train_test_model(model, criterion, optimizer, scheduler, num_epochs=25):
    """
    Train and test a PyTorch model with learning rate scheduling and model checkpointing.

    This function performs training and validation over a specified number of epochs.
    During training, it schedules the learning rate using a provided scheduler and saves
    the best performing model (based on validation accuracy). After training, the best model
    is evaluated on the test set.

    Args:
        model (torch.nn.Module): The neural network model to be trained.
        criterion (callable): The loss function.
        optimizer (torch.optim.Optimizer): The optimizer for model parameters.
        scheduler (torch.optim.lr_scheduler._LRScheduler): Learning rate scheduler.
        num_epochs (int): Number of epochs for training (default is 25).

    Returns:
        model (torch.nn.Module): The trained model with the best validation weights loaded.
        val_acc_h (list): History of validation accuracy for each epoch.
        train_acc_h (list): History of training accuracy for each epoch.
        test_acc (torch.Tensor): Accuracy on the test dataset.
    """
    import time, copy

    since = time.time()
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    val_acc_h = []
    train_acc_h = []

    for epoch in range(num_epochs):
        print(f'Epoch {epoch}/{num_epochs - 1} LR {scheduler.get_last_lr()[0]:.6f}')
        print('-' * 10)

        # Each epoch has a training and validation phase
        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluation mode

            running_loss = 0.0
            running_corrects = 0
```

```
            # Iterate over data.
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)

                # Zero the parameter gradients
                optimizer.zero_grad()

                # Forward pass and track history only in train mode
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # Backward pass and optimize only in train mode
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # Update running statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            # Step the learning rate scheduler after the training phase
            if phase == 'train':
                scheduler.step()

            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]
            print(f'{phase} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')

            if phase == 'train':
                train_acc_h.append(epoch_acc)
            elif phase == 'val':
                val_acc_h.append(epoch_acc)
                # Save the model if validation accuracy improved
                if epoch_acc > best_acc:
                    best_acc = epoch_acc
                    best_model_wts = copy.deepcopy(model.state_dict())

    # Testing phase
    print('Testing started')
    phase = 'test'
    # Load best model weights
    model.load_state_dict(best_model_wts)
    model.eval()
    running_loss = 0.0
    running_corrects = 0

    with torch.no_grad():
        for inputs, labels in dataloaders[phase]:
            inputs = inputs.to(device)
            labels = labels.to(device)
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            loss = criterion(outputs, labels)
            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)
```

```
    test_loss = running_loss / dataset_sizes[phase]
    test_acc = running_corrects.double() / dataset_sizes[phase]

    time_elapsed = time.time() - since
    print('Training and testing complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print(f'Best val Acc: {best_acc:.4f}')
    print(f'Test Acc: {test_acc:.4f}')

    return model, val_acc_h, train_acc_h, test_acc
```

```
model_conv = resnet18(weights=ResNet18_Weights.IMAGENET1K_V1).to(device)
for param in model_conv.parameters():
    param.requires_grad = True

num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, 2)
model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model_conv.fc.parameters(), lr=0.001, momentum=0.9)
exp_lr_scheduler = lr_scheduler.StepLR(optimizer, step_size=7, gamma=0.1)
```

```
model_conv_ft, val_acc_h, train_acc_h, test_acc = train_test_model(model_conv, criterion, optimizer,
   exp_lr_scheduler, num_epochs=10)
```

```
Epoch 0/9 LR 0.001000
----------
train Loss: 0.7246 Acc: 0.5309
val Loss: 0.6976 Acc: 0.5543
Epoch 1/9 LR 0.001000
----------
train Loss: 0.7227 Acc: 0.5062
val Loss: 0.6799 Acc: 0.5761
Epoch 2/9 LR 0.001000
----------
train Loss: 0.6482 Acc: 0.6049
val Loss: 0.5723 Acc: 0.6957
Epoch 3/9 LR 0.001000
----------
train Loss: 0.5913 Acc: 0.7160
val Loss: 0.5086 Acc: 0.7391
Epoch 4/9 LR 0.001000
----------
train Loss: 0.5552 Acc: 0.7284
val Loss: 0.4567 Acc: 0.7935
[... 14 lines omitted ...]
train Loss: 0.4491 Acc: 0.8354
val Loss: 0.3772 Acc: 0.8478
Epoch 9/9 LR 0.000100
----------
train Loss: 0.4080 Acc: 0.8848
val Loss: 0.3770 Acc: 0.8587
Testing started
Training and testing complete in 0m 42s
Best val Acc: 0.8587
Test Acc: 0.9180
```

Dynamic Quantization for More Layers

Dynamic quantization in PyTorch is designed to be simple and effective. However, it primarily supports certain layer types like `nn.Linear`. For dynamic quantization, the weights are quantized during runtime, which makes it easy to apply but restricts the scope of quantization to layers that involve matrix multiplications (such as fully connected layers). It does not typically support convolutional layers (`nn.Conv2d`), which are central to models like ResNet.

Despite this limitation, we can still use dynamic quantization for the `Linear` layers in the ResNet architecture. To apply quantization to a wider range of layers, such as convolutional layers, **static quantization** or **quantization aware training (QAT)** is required.

```
# Assuming model_conv_ft is your retrained model (e.g., ResNet18)
# Apply dynamic quantization only to nn.Linear layers
model_conv_ft = model_conv_ft.to('cpu').float()
D_quantized_model = torch.quantization.quantize_dynamic(
    model_conv_ft,  # Retrained model
    {nn.Linear},    # Quantize only Linear layers
    dtype=torch.qint8  # Use 8-bit integer quantization
)

# Move the quantized model to CPU (necessary for dynamic quantization)
D_quantized_model = D_quantized_model.to('cpu')
inputs = torch.randn(1, 3, 256, 256).float().to('cpu')

with torch.no_grad():
    outputs = D_quantized_model(inputs)
    print(outputs)
```

```
tensor([[ 0.2211, -1.0220]])
```

```
# Currently dynamic quantized linear layer (quantized::linear_dynamic) does not support execution on the CUDA
   (GPU) backend.
D_quantized_model = D_quantized_model.to('cpu')
D_quantized_model.eval()

running_loss = 0.0
running_corrects = 0
with torch.no_grad():
    for inputs, labels in dataloaders['test']:
        inputs = inputs.to('cpu')
        labels = labels.to('cpu')
        outputs = D_quantized_model(inputs)
        _, preds = torch.max(outputs, 1)
        loss = criterion(outputs, labels)
        running_loss += loss.item() * inputs.size(0)
        running_corrects += torch.sum(preds == labels.data)
        test_loss = running_loss / dataset_sizes['test']
        test_acc_q = running_corrects.double() / dataset_sizes['test']

print(f'Test acc before quantization: {test_acc}')
print(f'Test acc after quantization: {test_acc_q}')
```

```
Test acc before quantization: 0.9180327868852459
Test acc after quantization: 0.9180327868852459
```

```
def benchmark_model(model, device, dataloader, num_trials=10):
    model.to(device)
    model.eval()

    start_time = time.time()
    with torch.no_grad():
        for trial in range(num_trials):
            inputs, _ = next(iter(dataloader))
            inputs = inputs.to(device)
            model(inputs)
            if trial % 2 == 0:
                print(f"{trial + 1} trials completed")

    end_time = time.time()
    average_time = (end_time - start_time) / num_trials
    return average_time
```

```
# Benchmark the original model
original_model_time = benchmark_model(model_conv_ft, "cpu", dataloaders['test'])
print(f"Average inference time of original model: {original_model_time:.6f} seconds.\n")

# Benchmark the quantized model
quantized_model_time = benchmark_model(D_quantized_model, "cpu", dataloaders['test'])
print(f"Average inference time of quantized model: {quantized_model_time:.6f} seconds.\n")
```

```
1 trials completed
3 trials completed
5 trials completed
7 trials completed
9 trials completed
Average inference time of original model: 5.262737 seconds.

1 trials completed
3 trials completed
5 trials completed
7 trials completed
9 trials completed
Average inference time of quantized model: 5.201681 seconds.
```

```
# Function to calculate model size in megabytes (MB)
def model_size(model):
    param_size = 0
    for param in model.parameters():
        param_size += param.nelement() * param.element_size()
    buffer_size = 0
    for buffer in model.buffers():
        buffer_size += buffer.nelement() * buffer.element_size()
    size_all_mb = (param_size + buffer_size) / 1024**2
    return size_all_mb

# Calculate the size of the original and quantized models
original_model_size = model_size(model_conv_ft)
quantized_model_size = model_size(D_quantized_model)

print(f"Original model size: {original_model_size:.2f} MB\n")
print(f"Quantized model size: {quantized_model_size:.2f} MB")
```

```
Original model size: 42.68 MB

Quantized model size: 42.67 MB
```

11.8. Quantization Aware Training (QAT)

Quantization Aware Training (QAT) is a powerful approach that enables the model to simulate quantized behavior during training. This process helps the model adjust to the errors introduced by quantization and thus maintain high accuracy after quantization. QAT is particularly effective for models with convolutional layers, such as ResNet, where maintaining accuracy is critical for tasks like image classification.

In QAT, both the weights and activations are quantized during training, but the gradients are computed using floating-point precision. The forward pass simulates the effect of quantization through **fake quantization** layers (also known as fake quantized operations). After the model is trained, these fake quantized layers are replaced by real quantization layers during the conversion step.

11.8.1. Steps to Perform QAT:

1. **Model Preparation**: Before starting QAT, the model must be modified by adding `QuantStub` and `DeQuantStub` layers. These stubs simulate the quantization and dequantization of the inputs and outputs of the model. Additionally, we replace `nn.Linear` and `nn.Conv2d` layers with quantization-aware versions of these layers using the PyTorch `prepare_qat` function. This function introduces the necessary fake quantization operations for simulating quantization during training.
2. **Training**: The model is trained with quantization in mind, using the same training loop as in standard training. However, during the forward pass, the activations are quantized to simulate lower precision.
3. **Model Conversion**: Once training is complete, the fake quantization layers are replaced with actual quantized operations using the `convert` function.

11.8.2. Example of Preparing a Model for QAT:

We modify a pre-trained ResNet-18 by adding the `QuantStub` and `DeQuantStub` layers, and then prepare the model for QAT.

```
class ResNet18_QAT(nn.Module):
    def __init__(self, model):
        super(ResNet18_QAT, self).__init__()
        self.quant = QuantStub()
        self.model = model
        self.dequant = DeQuantStub()

    def forward(self, x):
        x = self.quant(x)
        x = self.model(x)
        x = self.dequant(x)
        return x
```

In this example, `QuantStub` is used to quantize inputs, and `DeQuantStub` is used to dequantize the outputs after the final layer. The `prepare_qat` function introduces fake quantization operations that allow the model to simulate quantization during training.

Training the Model with QAT:

After the model is prepared, the training loop remains the same as a standard training loop. Once training is completed, we convert the model into a fully quantized version:

```
def train_test_model_q(model, criterion, optimizer, scheduler, num_epochs=25):
    """
    Train and test a quantization-aware training (QAT) model.

    This function trains the model using both training and validation datasets over a specified
    number of epochs. It computes and prints loss and accuracy for each phase (training and validation)
    and then evaluates the model on a separate test dataset. Note that the scheduler is stepped during
    the training phase on every batch.

    Args:
        model (torch.nn.Module): The model to be trained.
        criterion (callable): Loss function (e.g., CrossEntropyLoss).
        optimizer (torch.optim.Optimizer): Optimizer for updating model parameters.
        scheduler (torch.optim.lr_scheduler._LRScheduler): Learning rate scheduler.
        num_epochs (int, optional): Number of epochs to train the model. Default is 25.

    Returns:
        model (torch.nn.Module): The trained model.
        val_acc_h (list): History of validation accuracy for each epoch.
        train_acc_h (list): History of training accuracy for each epoch.
        test_acc (torch.Tensor): Test accuracy computed on the test dataset.
    """
    import time
    since = time.time()

    val_acc_h = []
    train_acc_h = []

    for epoch in range(num_epochs):
        current_lr = scheduler.get_last_lr()[0]
        print(f'Epoch {epoch}/{num_epochs - 1} | LR: {current_lr:.6f}')
        print('-' * 40)

        for phase in ['train', 'val']:
            if phase == 'train':
                model.train()
            else:
                model.eval()

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data for the current phase.
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)

                optimizer.zero_grad()
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # Backward pass and optimization only in training phase
                    if phase == 'train':
```

```
                        loss.backward()
                        optimizer.step()
                        scheduler.step()

                # Update statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)

            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]
            print(f'{phase.capitalize()} Loss: {epoch_loss:.4f} Acc: {epoch_acc:.4f}')

            if phase == 'train':
                train_acc_h.append(epoch_acc)
            elif phase == 'val':
                val_acc_h.append(epoch_acc)

    # Test phase (no gradient computation)
    print('Testing started...')
    model.eval()
    running_loss = 0.0
    running_corrects = 0

    with torch.no_grad():
        for inputs, labels in dataloaders['test']:
            inputs = inputs.to(device)
            labels = labels.to(device)
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            loss = criterion(outputs, labels)

            running_loss += loss.item() * inputs.size(0)
            running_corrects += torch.sum(preds == labels.data)

    test_loss = running_loss / dataset_sizes['test']
    test_acc = running_corrects.double() / dataset_sizes['test']

    time_elapsed = time.time() - since
    print('Training and testing complete in {:.0f}m {:.0f}s'.format(
        time_elapsed // 60, time_elapsed % 60))
    print(f'Test Loss: {test_loss:.4f} | Test Acc: {test_acc:.4f}')

    return model, val_acc_h, train_acc_h, test_acc
```

We continue setting up training by configuring a `QConfig` with custom fake quantization parameters and then applying the `prepare_qat` function to automatically insert fake quantization layers throughout the model. In doing so, it transforms a pretrained model (in this case, **ResNet18**) into a quantization-ready version.

```
model_conv_ft = model_conv_ft.to('cpu')
model_qat = ResNet18_QAT(model_conv_ft)

# Custom QAT config with quant_min and quant_max for 8-bit quantization
custom_fake_quant = FakeQuantize.with_args(quant_min=0, quant_max=255)

# Use the default QAT config and override the activation observer
qat_config = get_default_qat_qconfig('fbgemm')
qat_config = torch.quantization.QConfig(
    activation=custom_fake_quant,
    weight=qat_config.weight
```

```
)

model_qat.qconfig = qat_config
model_qat = prepare_qat(model_qat)
model_qat = model_qat.to(device)

criterion_qat = nn.CrossEntropyLoss()

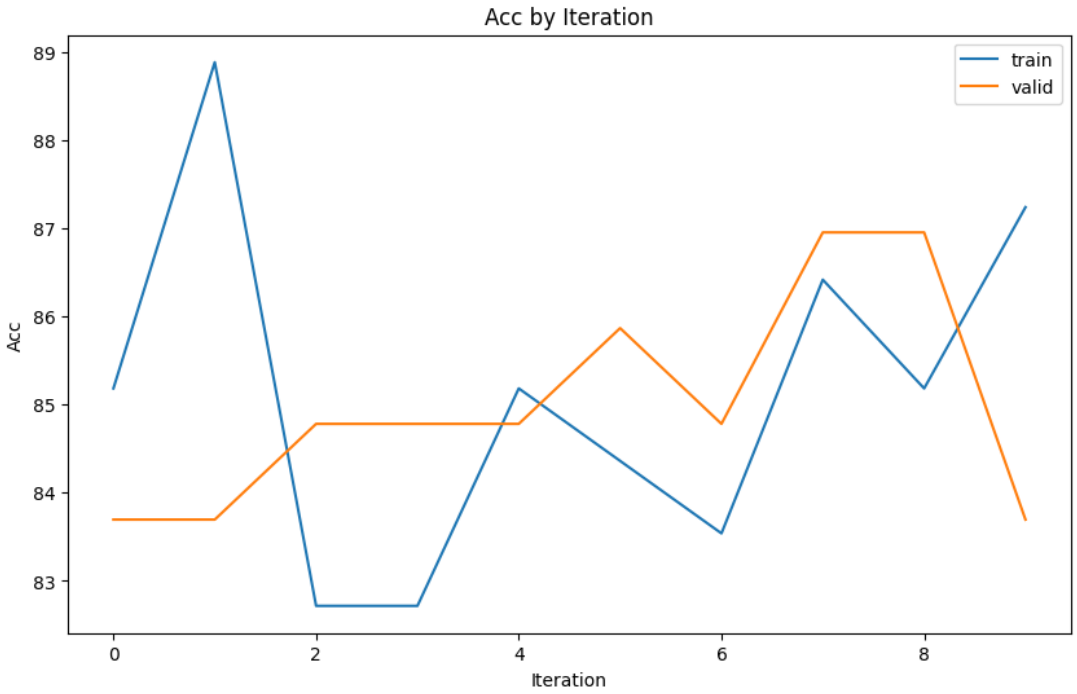
optimizer_qat = optim.SGD(model_qat.parameters(), lr=0.0001, momentum=0.25)
# Decay LR by a factor of 0.1 every 5 epochs
exp_lr_scheduler_qat = lr_scheduler.StepLR(optimizer_qat, step_size=7, gamma=0.1)
```

```
model_qat, val_acc_h, train_acc_h, test_acc = train_test_model_q(model_qat, criterion_qat, optimizer_qat,
   exp_lr_scheduler_qat, num_epochs=10)
```

```
Epoch 0/9 | LR: 0.000100
----------------------------------------
Train Loss: 0.4450 Acc: 0.8519
Val Loss: 0.3727 Acc: 0.8370
Epoch 1/9 | LR: 0.000100
----------------------------------------
Train Loss: 0.4073 Acc: 0.8889
Val Loss: 0.3676 Acc: 0.8370
Epoch 2/9 | LR: 0.000010
----------------------------------------
Train Loss: 0.4499 Acc: 0.8272
Val Loss: 0.3619 Acc: 0.8478
Epoch 3/9 | LR: 0.000010
----------------------------------------
Train Loss: 0.4333 Acc: 0.8272
Val Loss: 0.3642 Acc: 0.8478
Epoch 4/9 | LR: 0.000001
----------------------------------------
Train Loss: 0.4195 Acc: 0.8519
Val Loss: 0.3615 Acc: 0.8478
[... 13 lines omitted ...]
----------------------------------------
Train Loss: 0.4125 Acc: 0.8519
Val Loss: 0.3690 Acc: 0.8696
Epoch 9/9 | LR: 0.000000
----------------------------------------
Train Loss: 0.4196 Acc: 0.8724
Val Loss: 0.3719 Acc: 0.8370
Testing started...
Training and testing complete in 0m 45s
Test Loss: 0.4144 | Test Acc: 0.9180
```

```
def plot_acc(train_acc_h,val_acc_h):
    plt.figure(figsize=(10,6))
    plt.xlabel('Iteration')
    plt.ylabel('Acc')
    plt.title('Acc by Iteration')
    plt.plot(torch.tensor(train_acc_h, device = 'cpu')*100)
    plt.plot(torch.tensor(val_acc_h, device = 'cpu')*100)
    plt.legend(['train', 'valid'])
```

```
plot_acc(train_acc_h, val_acc_h)
```




```
model_qat.eval()
model_quantized_final = convert(model_qat.to('cpu').eval(), inplace=False)
```

```
# Function to calculate model size in megabytes (MB)
def model_size(model):
    param_size = 0
    for param in model.parameters():
        param_size += param.nelement() * param.element_size()
    buffer_size = 0
    for buffer in model.buffers():
        buffer_size += buffer.nelement() * buffer.element_size()
    size_all_mb = (param_size + buffer_size) / 1024**2
    return size_all_mb

original_model_size = model_size(model_conv_ft)
quantized_model_size = model_size(model_quantized_final)

print(f"\nOriginal model size: {original_model_size:.2f} MB")
print(f"Quantized model size: {quantized_model_size:.2f} MB")
```

```
Original model size: 42.68 MB
Quantized model size: 0.07 MB
```

```
model_quantized_final   # display value
```

```
ResNet18_QAT(
  (quant): Quantize(scale=tensor([0.0187]), zero_point=tensor([114]),
dtype=torch.quint8)
  (model): ResNet(
    (conv1): QuantizedConv2d(3, 64, kernel_size=(7, 7), stride=(2, 2),
scale=0.2902217507362366, zero_point=127, padding=(3, 3), bias=False)
    (bn1): QuantizedBatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
    (relu): ReLU(inplace=True)
    (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
    (layer1): Sequential(
      (0): BasicBlock(
        (conv1): QuantizedConv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
scale=0.10575834661722183, zero_point=181, padding=(1, 1), bias=False)
        (bn1): QuantizedBatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): QuantizedConv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
scale=0.030741408467292786, zero_point=161, padding=(1, 1), bias=False)
        (bn2): QuantizedBatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
      (1): BasicBlock(
        (conv1): QuantizedConv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
scale=0.07635319977998734, zero_point=154, padding=(1, 1), bias=False)
        (bn1): QuantizedBatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (relu): ReLU(inplace=True)
        (conv2): QuantizedConv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
scale=0.026994673535227776, zero_point=149, padding=(1, 1), bias=False)
[... 58 lines omitted ...]
        (relu): ReLU(inplace=True)
        (conv2): QuantizedConv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
scale=0.008363080210983753, zero_point=90, padding=(1, 1), bias=False)
        (bn2): QuantizedBatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
    (fc): QuantizedLinear(in_features=512, out_features=2,
scale=0.012178014032542706, zero_point=135, qscheme=torch.per_channel_affine)
  )
  (dequant): DeQuantize()
)
```

11.8.3. Conclusion on Quantization Methods

- **Dynamic Quantization**: This method is ideal for simpler models where only fully connected layers need to be quantized, providing immediate performance gains with minimal setup.
- **Static Quantization**: Useful when both weights and activations need to be quantized for models involving convolutional layers. This method requires post-training calibration.
- **Quantization Aware Training (QAT)**: The most comprehensive solution, allowing for quantization of more complex models like ResNet.

11.9. Homework (Optional)

1. **Dynamic vs. static on the same model.** Apply dynamic quantization to the fine-tuned ResNet-18 and measure latency + accuracy on the test set. Apply static quantization with a calibration of 100 images and measure the same metrics. Report the numbers in a small table.
2. **Size on disk.** For each of (float, dynamic, static, QAT) variants, serialize the model with `torch.save(model.state_dict(), ...)` and report file size in MB. Explain the storage differences.
3. **Quantization granularity.** Static quantization supports per-tensor and per-channel schemes. Retrain the QAT model with both and report which gives better accuracy on the test set.
4. **Calibration dataset size.** Quantize the same model with calibration sets of 10, 100, 1000 images. Report accuracy for each and argue when the law of diminishing returns kicks in.
5. **Target platforms.** Load your quantized model and run inference on CPU only. Compare latency to the float model on the same CPU. Discuss why GPU latency for INT8 behaves differently from CPU INT8.
6. **Custom quantizable module.** Write a short `nn.Module` subclass that inserts `quant/dequant` stubs at its boundary and supports static quantization cleanly. Demonstrate that `torch.quantization.convert` turns it into a fully quantized graph.

Chapter 12

Object Detection with YOLO11

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1JhJZSO1xVhuwBorErgrntX6t3jBo6AZt?usp=sharing

Object detection is a task that involves **identifying the location and class of objects in an image or video stream**.

The output of an object detector is a set of `bounding boxes` that enclose the objects in the image, along with `class labels` and confidence scores for each box. Object detection is a good choice when you need to identify objects of interest in a scene, but don't need to know exactly where the object is or its exact shape.

12.1. Models

YOLO11 pretrained Detect models are shown here. **Detect** models are pretrained on the **COCO** dataset, while Classify models are pretrained on the **ImageNet** dataset.

Models download automatically from the latest Ultralytics release on first use.

Model	Size (pixels)	mAPval 50-95	Speed CPU ONNX (ms)	Speed T4 TensorRT10 (ms)	Params (M)	FLOPs (B)
YOLO11n	640	39.5	56.1 ± 0.8	1.5 ± 0.0	2.6	6.5
YOLO11s	640	47.0	90.0 ± 1.2	2.5 ± 0.0	9.4	21.5
YOLO11m	640	51.5	183.2 ± 2.0	4.7 ± 0.1	20.1	68.0
YOLO11l	640	53.4	238.6 ± 1.4	6.2 ± 0.1	25.3	86.9
YOLO11x	640	54.7	462.8 ± 6.7	11.3 ± 0.2	56.9	194.9

- **mAPval** values are for single-model single-scale on COCO val2017 dataset.
  - Reproduce by: `yolo val detect data=coco -seg.yaml device=0`
- **Speed** averaged over COCO [8] val images using an Amazon EC2 P4d instance.
  - Reproduce by: `yolo val detect data=coco-seg.yaml batch=1 device=0|cpu`

12.1.1. Key Points:

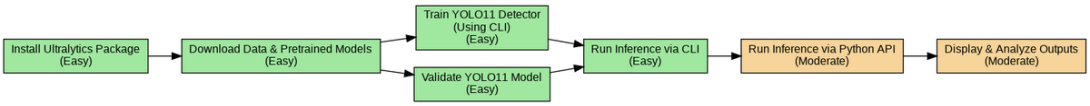
- **Different Models**:
  - The models range from **YOLO11n** (smallest, fastest, least parameters) to **YOLO11x** (largest, slowest, most parameters).

  - As you move from **n** to **x**, model size, accuracy (mAP), and computational requirements increase.
- **Why Choosing the Right Model Matters**:
  - **Speed vs. Accuracy**: Smaller models like **YOLO11n** are faster and require fewer computational resources but have lower accuracy. On the other hand, **YOLO11x** provides higher accuracy but is slower and more computationally expensive.
  - **Use Case Considerations**: Depending on your application (e.g., real-time scenarios, resource-limited devices, or cases needing high accuracy), you should choose a model that balances speed, accuracy, and hardware requirements.

12.2. Installing the Ultralytics Package

Ultralytics is the organization behind the development of YOLO (You Only Look Once) family of models for object detection, segmentation, and pose estimation, and provides tools and pretrained models for computer vision tasks.

```
!pip install -qq ultralytics
```



YOLO11 offers two distinct approaches for training, validating, and predicting with a model:

- **Command-Line Interface (CLI)**: A straightforward, non-Pythonic method that leverages the Ultralytics package to handle everything seamlessly in the background, without needing any code.
- **Pythonic Approach**: A more traditional method that allows for greater flexibility and customization, where you work directly with the Python API to manage the model's functionality programmatically.

12.3. YOLO versions, heads, and model sizes

YOLO is not a single architecture but a family of related single-stage detectors. Each generation tweaks the backbone, the neck, the detection head, and the training recipe. For course work, two dimensions matter.

Version. YOLOv5 is the most-deployed, YOLOv8 introduced anchor-free prediction, and YOLOv11 adds C3k2 and C2PSA attention blocks. Newer versions typically push the accuracy/latency frontier slightly further, but the API and training workflow stay almost identical, so code written for one version migrates to the next in a few lines.

Model size. Each version ships at least five sizes, nano, small, medium, large, extra-large, denoted by a letter suffix (`yolo11n.pt`, `yolo11s.pt`, `yolo11m.pt`, ...). The nano models are roughly 2 MB, run in real-time on CPU, and hit around 40 mAP on COCO. The extra-large models are 50+ MB, need GPU inference, and push past 55 mAP. Pick the smallest size that meets your accuracy target.

Head. All modern YOLO heads predict `(objectness, class_probs, bbox)` per spatial location on a small feature pyramid. In v8+ the anchor boxes of earlier versions are dropped. The head predicts box coordinates directly relative to the feature cell centre.

12.4. Train

You have three options to choose from when training a model: - **Build a new model from YAML**: Start from scratch by specifying model architecture in a YAML file (includes a download script for the coco8 dataset as well). - **Load a pretrained model**: Use a pretrained model to fine-tune or continue training, recommended for most use cases. - **Build from YAML and transfer weights**: Define your model architecture in YAML and load pretrained weights for a head start on training.

Downloading the `yolo11-nano` model with the `coco8-seg` dataset.

```
!wget -qq https://github.com/ultralytics/assets/releases/download/v8.3.0/yolo11n.pt
!wget -qq https://raw.githubusercontent.com/ultralytics/ultralytics/refs/heads/main/ultralytics/cfg/datasets/
  coco8.yaml
```

The `coco8-seg.yaml` has the following format:

```
## **Ultralytics  AGPL-3.0 License - <https://ultralytics.com/license**>

## **COCO8 dataset (first 8 images from COCO train2017) by Ultralytics**
## **Documentation: <https://docs.ultralytics.com/datasets/detect/coco8/**>
## **Example usage: yolo train data=coco8.yaml**
## **parent**
## **+-- ultralytics**
## **+-- datasets**
## **+-- coco8  <- downloads here (1 MB)**

## **Train/val/test sets as 1) dir: path/to/imgs, 2) file: path/to/imgs.txt, or 3) list: [path/to/imgs1, path/to/imgs2, ..]**
path: ../datasets/coco8 # dataset root dir
train: images/train # train images (relative to 'path') 4 images
val: images/val # val images (relative to 'path') 4 images
test: # test images (optional)

## **Classes**
names:
  0: person
  1: bicycle
  2: car
  3: motorcycle
  4: airplane
  5: bus
  6: train
  7: truck
  8: boat
  9: traffic light
  10: fire hydrant
  11: stop sign
  12: parking meter
  13: bench
  14: bird
  15: cat
  16: dog
  17: horse
  18: sheep
```

```
19: cow
20: elephant
21: bear
22: zebra
23: giraffe
24: backpack
25: umbrella
26: handbag
27: tie
28: suitcase
29: frisbee
30: skis
31: snowboard
32: sports ball
33: kite
34: baseball bat
35: baseball glove
36: skateboard
37: surfboard
38: tennis racket
39: bottle
40: wine glass
41: cup
42: fork
43: knife
44: spoon
45: bowl
46: banana
47: apple
48: sandwich
49: orange
50: broccoli
51: carrot
52: hot dog
53: pizza
54: donut
55: cake
56: chair
57: couch
58: potted plant
59: bed
60: dining table
61: toilet
62: tv
63: laptop
64: mouse
65: remote
66: keyboard
67: cell phone
68: microwave
69: oven
70: toaster
71: sink
72: refrigerator
73: book
74: clock
75: vase
76: scissors
77: teddy bear
78: hair drier
79: toothbrush
```

```
## **Download script/URL (optional)**
download: <https://github.com/ultralytics/assets/releases/download/v0.0.0/coco8.zip>
```

12.5. Building a New Model (CLI)

The expected `mAP50` for validation is approximately 0.

All training runs, including metrics, plot images, CSV logs, and model weights, are saved in the directory: `/content/runs/detect/train`

```
!yolo detect train data=coco8.yaml model=yolo11n.yaml epochs=100 imgsz=640
```

```
Creating new Ultralytics Settings v0.0.6 file
View Ultralytics Settings with 'yolo settings' or at
'/root/.config/Ultralytics/settings.json'
Update Settings with 'yolo settings key=value', i.e. 'yolo settings
runs_dir=path/to/dir'. For help see
https://docs.ultralytics.com/quickstart/#ultralytics-settings.
Ultralytics 8.3.70  Python-3.11.11 torch-2.5.1+cu124 CUDA:0 (Tesla T4,
15095MiB)
engine/trainer: task=detect, mode=train, model=yolo11n.yaml, data=coco8.yaml,
epochs=100, time=None, patience=100, batch=16, imgsz=640, save=True,
save_period=-1, cach ... [1377 chars truncated]

Dataset 'coco8.yaml' images not found !, missing path
'/content/datasets/coco8/images/val'
Downloading https://ultralytics.com/assets/coco8.zip to
'/content/datasets/coco8.zip'...
100% 433k/433k [00:00<00:00, 15.7MB/s]
Unzipping /content/datasets/coco8.zip to /content/datasets/coco8...: 100% 25/25
[00:00<00:00, 4854.29file/s]
Dataset download success  (0.9s), saved to /content/datasets

Downloading https://ultralytics.com/assets/Arial.ttf to
'/root/.config/Ultralytics/Arial.ttf'...
100% 755k/755k [00:00<00:00, 21.0MB/s]
WARNING: All log messages before absl::InitializeLog() is called are written to
STDERR
E0000 00:00:1738672769.711720    2173 cuda_dnn.cc:8310] Unable to register cuDNN
factory: Attempting to register factory for plugin cuDNN when one has already
been registered
E0000 00:00:1738672769.773345    2173 cuda_blas.cc:1418] Unable to register
cuBLAS factory: Attempting to register factory for plugin cuBLAS when one has
already been registered

                   from  n    params  module
arguments
  0                  -1  1       464  ultralytics.nn.modules.conv.Conv
[3, 16, 3, 2]
[... 549 lines omitted ...]
Optimizer stripped from runs/detect/train/weights/best.pt, 5.5MB
```

```
Validating runs/detect/train/weights/best.pt...
Ultralytics 8.3.70  Python-3.11.11 torch-2.5.1+cu124 CUDA:0 (Tesla T4,
15095MiB)
YOLO11n summary (fused): 238 layers, 2,616,248 parameters, 0 gradients, 6.5
GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50
mAP50-95): 100% 1/1 [00:00<00:00, 23.93it/s]
                   all          4         17          0          0          0
0
Speed: 0.3ms preprocess, 4.4ms inference, 0.0ms loss, 0.2ms postprocess per
image
Results saved to runs/detect/train
 Learn more at https://docs.ultralytics.com/modes/train
```

12.6. Loading a Pretrained Model (CLI)

The expected `mAP50` (class average precision) for validation is approximately 0.941. The mask average precision should be around `0.849`

All training runs, including metrics, plot images, CSV logs, and model weights, are saved in the directory: `/content/runs/detect/train2`

```
!yolo detect train data=coco8.yaml model=yolo11n.pt epochs=100 imgsz=640
```

```
Ultralytics 8.3.70  Python-3.11.11 torch-2.5.1+cu124 CUDA:0 (Tesla T4,
15095MiB)
engine/trainer: task=detect, mode=train, model=yolo11n.pt, data=coco8.yaml,
epochs=100, time=None, patience=100, batch=16, imgsz=640, save=True,
save_period=-1, cache= ... [1377 chars truncated]
WARNING: All log messages before absl::InitializeLog() is called are written to
STDERR
E0000 00:00:1738672925.789062    2922 cuda_dnn.cc:8310] Unable to register cuDNN
factory: Attempting to register factory for plugin cuDNN when one has already
been registered
E0000 00:00:1738672925.799787    2922 cuda_blas.cc:1418] Unable to register
cuBLAS factory: Attempting to register factory for plugin cuBLAS when one has
already been registered

                   from  n    params  module
arguments
  0                  -1  1       464  ultralytics.nn.modules.conv.Conv
[3, 16, 3, 2]
  1                  -1  1      4672  ultralytics.nn.modules.conv.Conv
[16, 32, 3, 2]
  2                  -1  1      6640  ultralytics.nn.modules.block.C3k2
[32, 64, 1, False, 0.25]
  3                  -1  1     36992  ultralytics.nn.modules.conv.Conv
[64, 64, 3, 2]
  4                  -1  1     26080  ultralytics.nn.modules.block.C3k2
[64, 128, 1, False, 0.25]
  5                  -1  1    147712  ultralytics.nn.modules.conv.Conv
```

```
[128, 128, 3, 2]
  6                  -1  1     87040  ultralytics.nn.modules.block.C3k2
[128, 128, 1, True]
  7                  -1  1    295424  ultralytics.nn.modules.conv.Conv
[128, 256, 3, 2]
  8                  -1  1    346112  ultralytics.nn.modules.block.C3k2
[256, 256, 1, True]
  9                  -1  1    164608  ultralytics.nn.modules.block.SPPF
[256, 256, 5]
 10                  -1  1    249728  ultralytics.nn.modules.block.C2PSA
[256, 256, 1]
 11                  -1  1         0  torch.nn.modules.upsampling.Upsample
[None, 2, 'nearest']
 12             [-1, 6]  1         0  ultralytics.nn.modules.conv.Concat
[1]
[... 542 lines omitted ...]
                   all          4         17      0.597      0.867      0.895
0.652
                person          3         10      0.625        0.7      0.644
0.321
                   dog          1          1      0.523          1      0.995
0.796
                 horse          1          2      0.594          1      0.995
0.676
              elephant          1          2      0.371        0.5      0.745
0.325
              umbrella          1          1      0.549          1      0.995
0.895
          potted plant          1          1      0.921          1      0.995
0.895
Speed: 0.5ms preprocess, 10.5ms inference, 0.0ms loss, 3.4ms postprocess per
image
Results saved to runs/detect/train2
 Learn more at https://docs.ultralytics.com/modes/train
```

12.7. Validation

You have two options to choose from when validating a model: - **Validate an official model**: `yolo11n.pt`, `yolo11x.pt` etc. - **Validate your trained model**: `best.pt`

12.8. Validate an Official Model (CLI)

The expected `mAP50` (class average precision) for validation is approximately 0.917. Pose `mAP50` is approximately 0.772.

All training runs, including metrics, plot images, CSV logs, and model weights, are saved in the directory: `/content/runs/detect/val`

```
!yolo detect val model=yolo11n.pt
```

```
WARNING ! 'data' argument is missing. Using default 'data=coco8.yaml'.
Ultralytics 8.3.70  Python-3.11.11 torch-2.5.1+cu124 CUDA:0 (Tesla T4,
15095MiB)
YOLO11n summary (fused): 238 layers, 2,616,248 parameters, 0 gradients, 6.5
GFLOPs
val: Scanning /content/datasets/coco8/labels/val.cache... 4 images, 0
backgrounds, 0 corrupt: 100% 4/4 [00:00<?, ?it/s]
                 Class     Images  Instances      Box(P          R      mAP50
mAP50-95): 100% 1/1 [00:00<00:00,  3.30it/s]
                   all          4         17       0.57       0.85      0.847
0.632
                person          3         10      0.557        0.6      0.585
0.272
                   dog          1          1      0.548          1      0.995
0.697
                 horse          1          2      0.531          1      0.995
0.674
              elephant          1          2      0.371        0.5      0.516
0.256
              umbrella          1          1      0.569          1      0.995
0.995
          potted plant          1          1      0.847          1      0.995
0.895
Speed: 1.2ms preprocess, 17.0ms inference, 0.0ms loss, 32.6ms postprocess per
image
Results saved to runs/detect/val
 Learn more at https://docs.ultralytics.com/modes/val
```

12.9. Inference

You have two options to choose from when testing a model: - **Test an official model**: `yolo11n.pt`, `yolo11x.pt` etc. - **Test your trained model**: `best.pt`

12.10. Infer an Official Model (CLI)

The output of YOLO's prediction, which includes an image with detected people and their corresponding keypoints, is saved in the `/content/runs/detect/predict` folder.

```
!yolo detect predict model=yolo11n.pt source='https://ultralytics.com/images/bus.jpg'
```

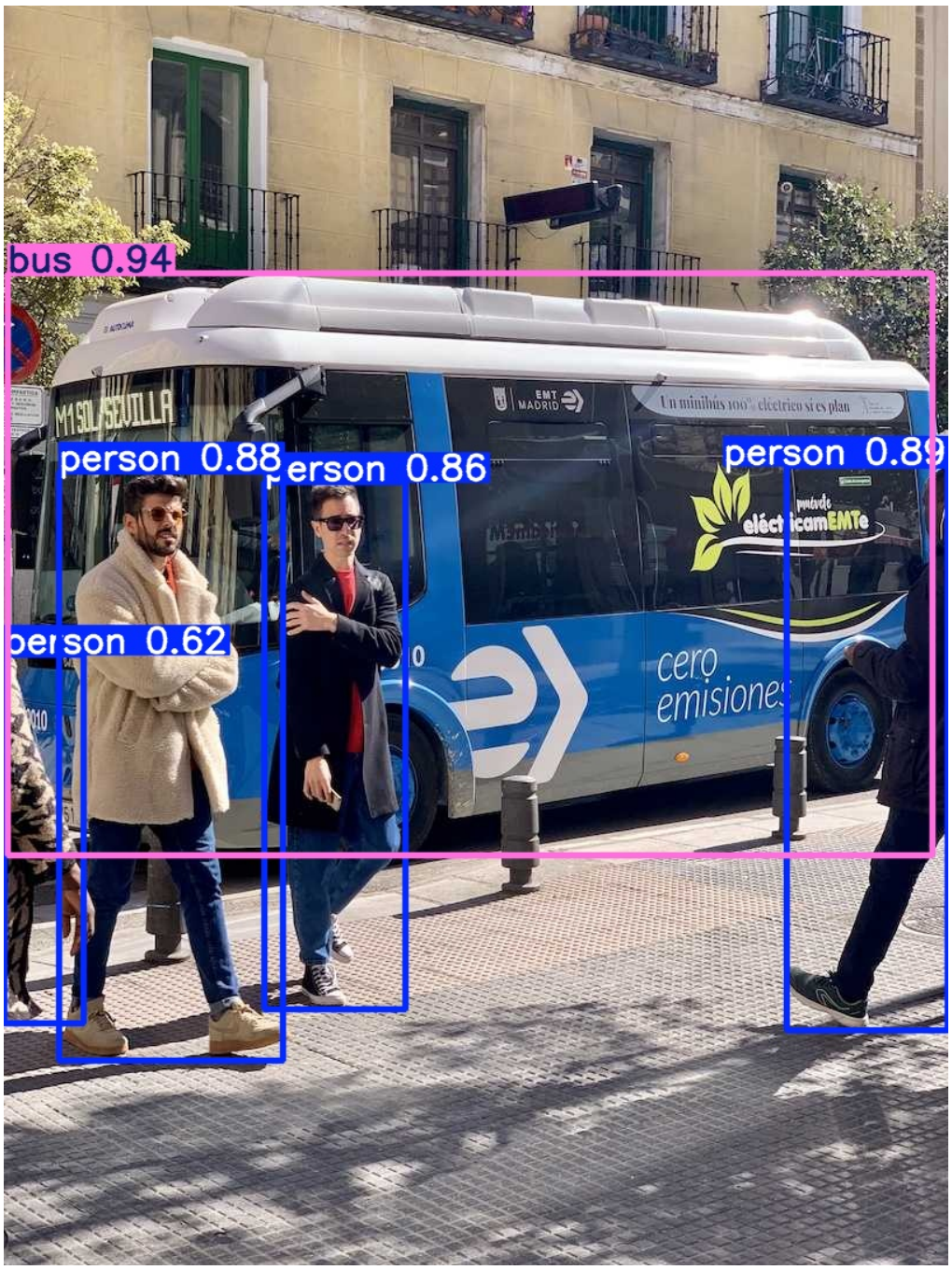
```
Ultralytics 8.3.70  Python-3.11.11 torch-2.5.1+cu124 CUDA:0 (Tesla T4,
15095MiB)
YOLO11n summary (fused): 238 layers, 2,616,248 parameters, 0 gradients, 6.5
GFLOPs

Downloading https://ultralytics.com/images/bus.jpg to 'bus.jpg'...
100% 134k/134k [00:00<00:00, 7.98MB/s]
image 1/1 /content/bus.jpg: 640x480 4 persons, 1 bus, 49.0ms
Speed: 8.6ms preprocess, 49.0ms inference, 134.1ms postprocess per image at
shape (1, 3, 640, 480)
```

```
Results saved to runs/detect/predict
 Learn more at https://docs.ultralytics.com/modes/predict
```

12.11. Displaying the output

It's important to note that **YOLO** first performs object detection as a prerequisite for segmentation. This means the model must identify objects in the image before it can accurately generate and display their segmentation masks.



12.12. Infer an Official Model (Python)

Now, let's perform inference using a larger, more advanced model, `yolo11x`, for improved results.

This step involves importing the necessary libraries and downloading the `yolo11x` model.

```
!wget -qq https://github.com/ultralytics/assets/releases/download/v8.3.0/yolo11x.pt

from PIL import Image
import ultralytics
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import torch
import numpy as np
import random
import cv2
```

```
# food.jpg and room.jpg ship with the course materials
```

```
ultralytics.checks()
print(f"Setup complete. Using torch {torch.__version__} on device ({torch.cuda.get_device_properties(0).name
   if torch.cuda.is_available() else 'CPU'})")

# load an official model
model = ultralytics.YOLO("yolo11n.pt")
# predict on multiple images
results = model(["/content/food.jpg", "/content/room.jpg"], conf=0.25)
```

```
Ultralytics 8.3.70  Python-3.11.11 torch-2.5.1+cu124 CUDA:0 (Tesla T4,
15095MiB)
Setup complete  (2 CPUs, 12.7 GB RAM, 33.1/235.7 GB disk)
Setup complete. Using torch 2.5.1+cu124 on device (Tesla T4)

0: 640x640 3 cups, 2 bowls, 2 forks, 1 dining table, 1 spoon, 1 knife, 9.7ms
1: 640x640 2 beds, 9.7ms
Speed: 4.0ms preprocess, 9.7ms inference, 1.9ms postprocess per image at shape
(1, 3, 640, 640)
```

Visualizing the model architecture

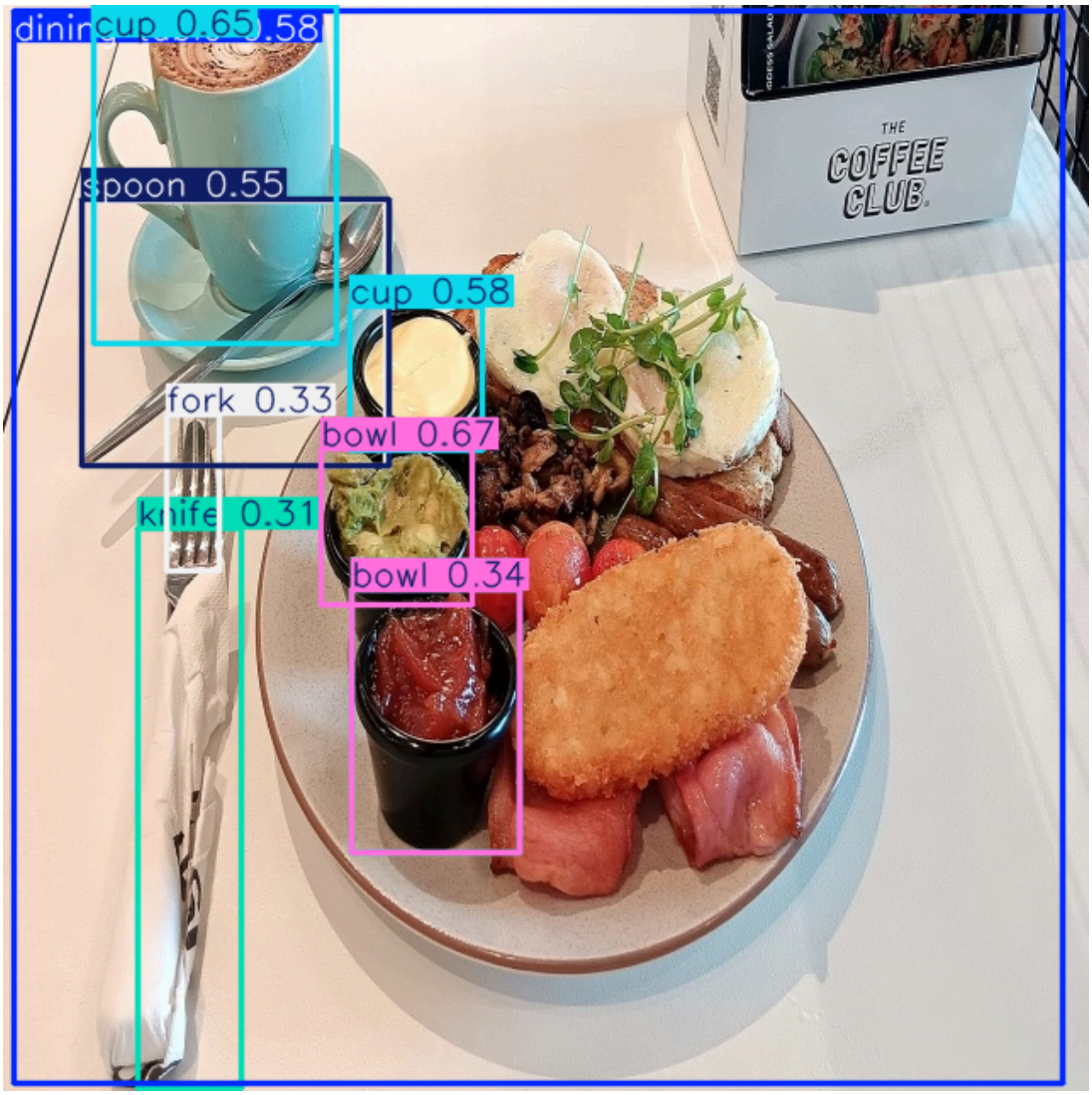
```
model   # display value
```

```
YOLO(
  (model): DetectionModel(
    (model): Sequential(
      (0): Conv(
        (conv): Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (act): SiLU(inplace=True)
      )
      (1): Conv(
        (conv): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1))
        (act): SiLU(inplace=True)
      )
      (2): C3k2(
```

```
        (cv1): Conv(
          (conv): Conv2d(32, 32, kernel_size=(1, 1), stride=(1, 1))
          (act): SiLU(inplace=True)
        )
        (cv2): Conv(
          (conv): Conv2d(48, 64, kernel_size=(1, 1), stride=(1, 1))
          (act): SiLU(inplace=True)
        )
[... 424 lines omitted ...]
            (2): Conv2d(80, 80, kernel_size=(1, 1), stride=(1, 1))
          )
        )
        (dfl): DFL(
          (conv): Conv2d(16, 1, kernel_size=(1, 1), stride=(1, 1), bias=False)
        )
      )
    )
  )
)
```

Going trough the output for all images and displaying them.

```
for i, result in enumerate(results):
    result.show()  # display to screen
    result.save(filename=f"/content/runs/detect/result_{i}.jpg")  # save to disk
```


dinin
cup 0.65
.58
spoon 0.55
cup 0.58
fork 0.33
bowl 0.67
knife 0.31
bowl 0.34
THE
COFFEE
CLUB




```
for i, result in enumerate(results):
    boxes = result.boxes  # Boxes object for bounding box outputs
    print(f'Bounding boxes shape: {boxes.shape} - The first dimension represents the number of objects in the
  image, with each box consisting of 6 values: x_center, y_center, width, height, confidence, class.')
```

```
Bounding boxes shape: torch.Size([10, 6]) - The first dimension represents the
number of objects in the image, with each box consisting of 6 values: x_center,
y_center, width, height, confidence, class.
Bounding boxes shape: torch.Size([2, 6]) - The first dimension represents the
number of objects in the image, with each box consisting of 6 values: x_center,
y_center, width, height, confidence, class.
```

The `xy` tensor stores the pixel coordinates for all the locations containing an object.

Class labels are not stored in `result.boxes`. **YOLO** first detects objects within an image using bounding boxes and classifies them into specific categories (classes) during object detection.

Additionally, the `xyn` tensor contains the normalized `x` and `y` values for each masked pixel, ensuring that the positions are scaled appropriately.

```
print(boxes)
```

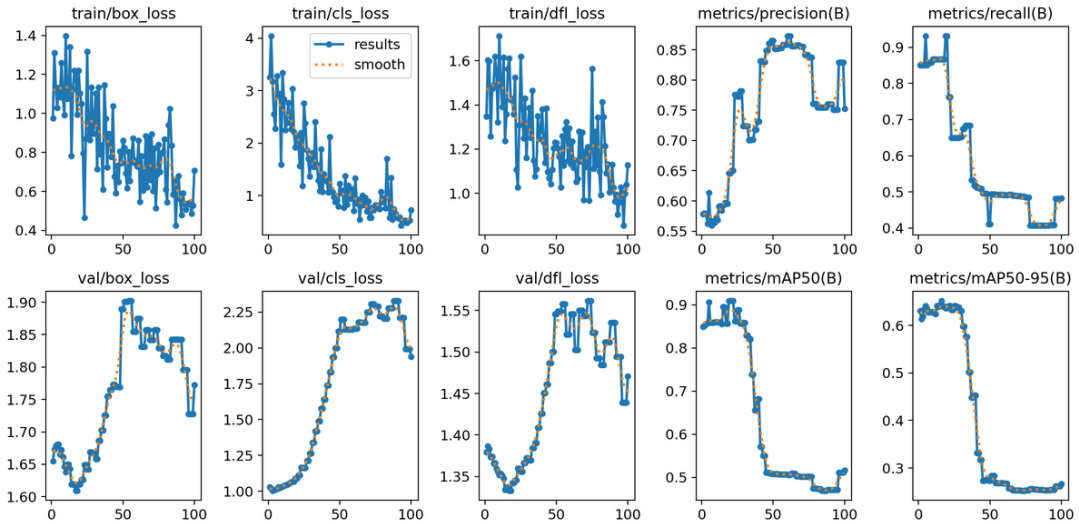
```
ultralytics.engine.results.Boxes object with attributes:

cls: tensor([58., 75.], device='cuda:0')
conf: tensor([0.3638, 0.3581], device='cuda:0')
data: tensor([[1.9280e+01, 3.3410e+02, 1.0272e+02, 5.4859e+02, 3.6383e-01,
5.8000e+01],
        [1.7718e+01, 4.1124e+02, 7.6252e+01, 5.4886e+02, 3.5815e-01,
7.5000e+01]], device='cuda:0')
id: None
is_track: False
orig_shape: (640, 640)
shape: torch.Size([2, 6])
xywh: tensor([[ 60.9997, 441.3423,  83.4398, 214.4943],
        [ 46.9847, 480.0526,  58.5336, 137.6237]], device='cuda:0')
xywhn: tensor([[0.0953, 0.6896, 0.1304, 0.3351],
        [0.0734, 0.7501, 0.0915, 0.2150]], device='cuda:0')
xyxy: tensor([[ 19.2798, 334.0952, 102.7196, 548.5894],
        [ 17.7179, 411.2408,  76.2516, 548.8645]], device='cuda:0')
xyxyn: tensor([[0.0301, 0.5220, 0.1605, 0.8572],
        [0.0277, 0.6426, 0.1191, 0.8576]], device='cuda:0')
```

12.13. Train Results

Each training run stores its own set of results, including detailed metrics for both training and validation. These results record losses for bounding box regression (`box loss`), classification (`cls loss`), and distribution focal loss (`dfl loss`), along with precision and recall values. Additionally, the runs report performance metrics such as `mAP` at an `IoU` threshold of 0.50 (`mAP50`) and the average mAP across IoU thresholds ranging from 0.50 to 0.95 (`mAP50-95`). This comprehensive set of metrics provides a clear picture of the model's detection accuracy and helps in evaluating its performance across different aspects of object detection.

```
Image.open('/content/runs/detect/train2/results.png')
```



12.14. Homework (Optional)

1. **Size vs. accuracy.** Run inference on the same sample video with `yolo11n.pt`, `yolo11s.pt`, and `yolo11m.pt`. Report mAP on the COCO val subset and latency per image on CPU. Plot accuracy vs. latency and choose an operating point.
2. **Fine-tune on a custom class.** Pick any single-class dataset you can find or collect (50-200 labelled images is enough), format the labels in YOLO txt format, and fine-tune `yolo11n.pt` for 30 epochs. Report training curves and test mAP.
3. **Confidence threshold sweep.** Run inference on a fixed test set with confidence thresholds `0.1, 0.25, 0.5, 0.75`. Plot precision and recall vs. threshold and identify the knee of the curve.
4. **Export to ONNX.** Use the Ultralytics `model.export(format='onnx')` API to convert a trained model to ONNX. Load the ONNX model with `onnxruntime` and verify that inference produces the same boxes as the PyTorch model.
5. **Per-class analysis.** Compute per-class AP on COCO val using the pretrained `yolo11n.pt`. Identify the three best-performing and three worst-performing classes. Speculate on why the worst ones are harder.
6. **Augmentation audit.** The Ultralytics training config applies `mosaic`, `mixup`, and `random_perspective` by default. Turn each off individually and retrain for 10 epochs. Report which augmentation contributes the most to final accuracy.

Chapter 13

Faster R-CNN for Object Detection

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1ClHbP4lBDy4IrPXEnRIpjNTxBtAajBEE?usp=sharing

Object detection tasks require identifying **what** objects are present in an image and **where** they are located. Unlike **single-shot** detectors such as YOLO or SSD, **Faster R-CNN** follows a **two-stage** approach: 1. **Region Proposal Network (RPN)**: Generates a set of object proposal bounding boxes. 2. **Fast R-CNN head**: Classifies each region and refines its bounding box coordinates.

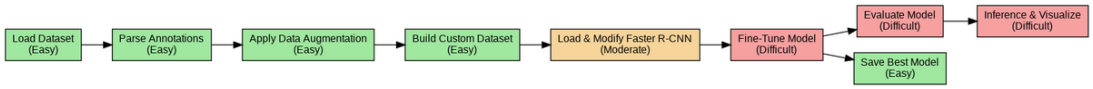
In this notebook, we'll: - **Load** a benchmark object detection dataset from TorchVision (Penn-Fudan [7] Pedestrian Detection, for demonstration). - Apply **data augmentations** using PyTorch transforms. - **Fine-tune** a **Faster R-CNN [16]** model pre-trained on COCO [8] for multi-class object detection. - Evaluate the model using **metrics** such as **mAP** and **visualize** detection results.

13.1. Necessary Imports

```
!pip install torchinfo --quiet
```

```
import os
import glob
import re
import random
import math
import numpy as np
import torch
import torchvision
import time
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import matplotlib.patches as patches
import torchvision
import cv2
import torchvision.transforms.functional as TF
import torchvision.transforms as T
from PIL import Image
from torchinfo import summary
from torchvision.datasets.utils import download_url
from IPython.display import Image as IMG
from torchvision import transforms
from torch.utils.data import DataLoader, random_split, Subset, Dataset
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
import matplotlib.pyplot as plt
from torchvision.io import read_image
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```



13.2. The Penn-Fudan Pedestrian Detection Dataset

For demonstration, we'll use the Penn-Fudan Pedestrian Detection Dataset. This dataset contains images of pedestrians with segmentation masks. However, **we only need bounding box annotations** for Faster R-CNN. This dataset includes both.

The **Penn-Fudan Dataset** is a lightweight dataset containing only **170 images**, with object classes limited to **`person`** and **`background`**. While it is possible to **fine-tune a Faster R-CNN model** on a larger dataset like **MS COCO** (which it was originally trained on), increasing the number of classes requires a **significantly larger dataset** to ensure proper representation.

An alternative is the **COCO128** dataset, a smaller subset of **MS COCO**, which can be used for **fine-tuning Faster R-CNN**. However, since **MS COCO** contains **80 object classes**, class representation in such a subset can be **highly imbalanced and sparse**.

For this tutorial, we will **use the Penn-Fudan dataset** because it is: - **Simple** and easy to work with. - **Computationally efficient** for training and evaluation. - **Well-suited for demonstrating** the process of training a **Faster R-CNN model** without requiring excessive resources.

Let's load in our data **using Python**.

```
root = './data'
url = 'https://www.cis.upenn.edu/~jshi/ped_html/PennFudanPed.zip'
zip_name = 'PennFudanPed.zip'

if not os.path.exists(os.path.join(root, "PennFudanPed")):
    download_url(url, root)
    import zipfile
    with zipfile.ZipFile(os.path.join(root, zip_name), 'r') as zip_ref:
        zip_ref.extractall(root)
    print("Dataset extracted.")
else:
    print("Dataset already downloaded and extracted.")
```

```
Downloading https://www.cis.upenn.edu/~jshi/ped_html/PennFudanPed.zip to
./data/PennFudanPed.zip
```

```
100%|##########| 53.7M/53.7M [00:01<00:00, 29.2MB/s]
```

```
Dataset extracted.
```

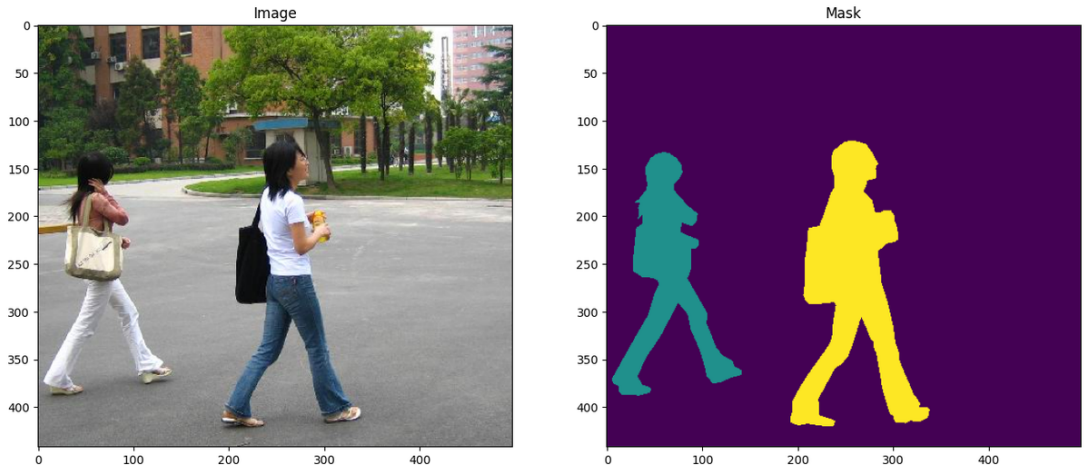
13.3. Showcasing our Dataset

Since the **Penn-Fudan dataset** supports **instance segmentation**, each image is accompanied by its corresponding **segmentation mask**. However, since we are **not utilizing the mask head** of **Faster**

R-CNN, these masks can be **omitted** for our task. Instead, we will focus solely on **bounding box detection**.

```
image = read_image("/content/data/PennFudanPed/PNGImages/FudanPed00019.png")
mask = read_image("/content/data/PennFudanPed/PedMasks/FudanPed00019_mask.png")

plt.figure(figsize=(16, 8))
plt.subplot(121)
plt.title("Image")
plt.imshow(image.permute(1, 2, 0))
plt.subplot(122)
plt.title("Mask")
plt.imshow(mask.permute(1, 2, 0))
plt.show()
```



The annotation files for the **Penn-Fudan Pedestrian dataset** are **Pascal VOC [6] -compatible** and follow a structured format. Each image located in `data/PennFudanPed/PNGImages/.png` has a corresponding annotation file in `data/PennFudanPed/Annotation/.txt`.

Each annotation file contains **bounding box coordinates** and **class labels** for the objects in that specific image. The content of a typical annotation file is as follows:

```
# Compatible with PASCAL Annotation Version 1.00
Image filename : "PennFudanPed/PNGImages/FudanPed00001.png"
Image size (X x Y x C) : 559 x 536 x 3
Database : "The Penn-Fudan-Pedestrian Database"
Objects with ground truth : 2 { "PASpersonWalking" "PASpersonWalking" }
# Note there may be some objects not included in the ground truth list for they are severe-occluded
# or have very small size.
# Top left pixel coordinates : (1, 1)
# Details for pedestrian 1 ("PASpersonWalking")
Original label for object 1 "PASpersonWalking" : "PennFudanPed"
Bounding box for object 1 "PASpersonWalking" (Xmin, Ymin) - (Xmax, Ymax) : (160, 182) - (302, 431)
Pixel mask for object 1 "PASpersonWalking" : "PennFudanPed/PedMasks/FudanPed00001_mask.png"

# Details for pedestrian 2 ("PASpersonWalking")
Original label for object 2 "PASpersonWalking" : "PennFudanPed"
Bounding box for object 2 "PASpersonWalking" (Xmin, Ymin) - (Xmax, Ymax) : (420, 171) - (535, 486)
Pixel mask for object 2 "PASpersonWalking" : "PennFudanPed/PedMasks/FudanPed00001_mask.png"
```

Each annotation file provides detailed **bounding box** information for detected objects:

- Class label: *PASpersonWalking*
- Bounding box coordinates: (`xmin, ymin, xmax, ymax`)
- Segmentation mask file (not needed for Faster R-CNN without a mask head)

```
def parse_annotation_file(annotation_path):
    """
    Parses a PennFudanPed annotation text file to extract bounding boxes and their labels.

    Args:
        annotation_path (str): Path to the annotation text file.

    Returns:
        List[Tuple[str, int, int, int, int]]: A list of tuples for each object in the format
            (object_label, xmin, ymin, xmax, ymax).
    """
    with open(annotation_path, 'r') as f:
        content = f.read()

    pattern = r'Bounding box for object \d+ "([^"]+)" \(Xmin, Ymin\) - \(Xmax, Ymax\) :
  \((\d+),\s*(\d+)\)\s*-\s*\((\d+),\s*(\d+)\)'
    matches = re.findall(pattern, content)
    boxes = [(label, int(xmin), int(ymin), int(xmax), int(ymax)) for label, xmin, ymin, xmax, ymax in matches]
    return boxes

def plot_image_with_annotations(image_path, annotation_path, ax):
    """
    Loads an image and its annotation, then plots the image on the given axis with bounding boxes and labels.

    Args:
        image_path (str): Path to the image file.
        annotation_path (str): Path to the corresponding annotation file.
        ax (matplotlib.axes.Axes): The axes on which to plot the image and annotations.
    """
    image = read_image(image_path)
    image_np = image.permute(1, 2, 0).cpu().numpy()
    boxes = parse_annotation_file(annotation_path)

    ax.imshow(image_np)
    ax.set_title(os.path.basename(image_path))

    for (label, xmin, ymin, xmax, ymax) in boxes:
        width = xmax - xmin
        height = ymax - ymin
        rect = patches.Rectangle((xmin, ymin), width, height, linewidth=2, edgecolor='cyan', facecolor='none')
        ax.add_patch(rect)
        ax.text(xmin + 4, max(0, ymin - 6), label, fontsize=9, color='cyan',
                verticalalignment='bottom', bbox=dict(facecolor='black', alpha=0.7, edgecolor='none'))
    ax.axis('off')
```

```
annotations_dir = '/content/data/PennFudanPed/Annotation'
images_dir = '/content/data/PennFudanPed/PNGImages'

annotation_files = sorted(glob.glob(os.path.join(annotations_dir, '*.txt')))
image_files = sorted(glob.glob(os.path.join(images_dir, '*.png')))

print("Number of annotation files: ", len(annotation_files))
print("Number of image files: ", len(image_files))
```

```
Number of annotation files:  170
Number of image files:  170
```

```
classes = ['Background', 'PASpersonWalking']
label_mapping = {cls: i for i, cls in enumerate(classes)}

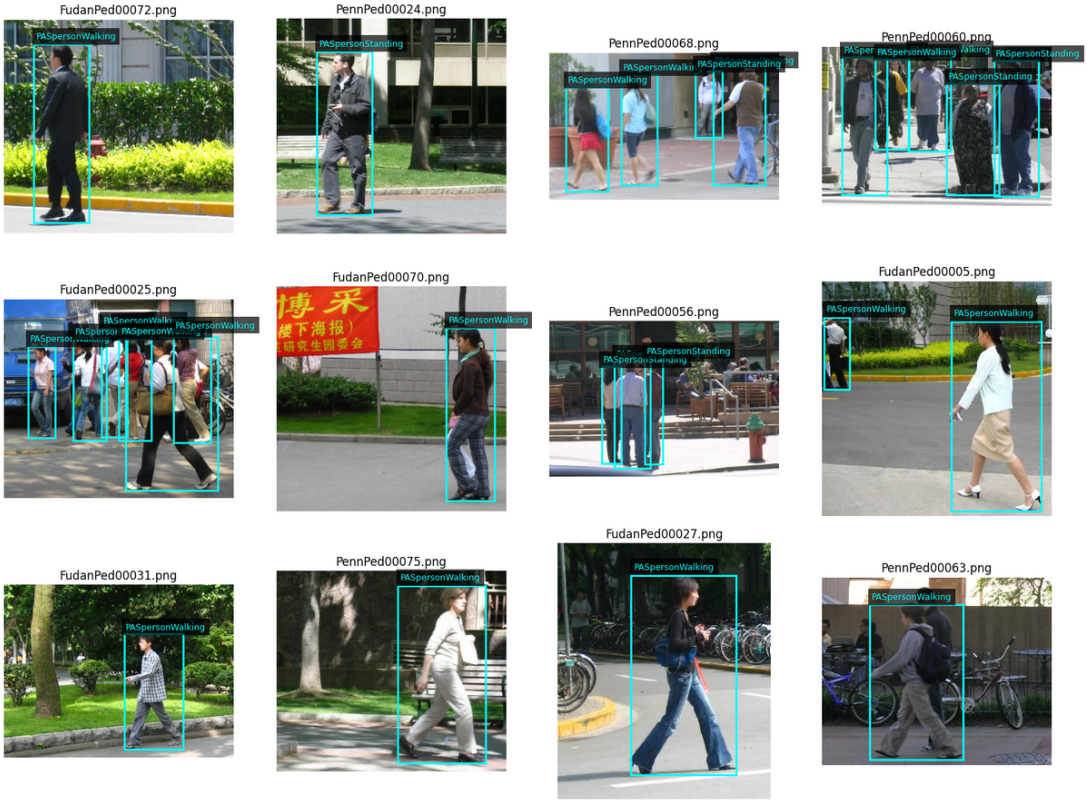
num_examples = 12
random_indices = random.sample(range(len(image_files)), 12)

selected_annotation_files = [annotation_files[i] for i in random_indices]
selected_image_files = [image_files[i] for i in random_indices]

fig, axes = plt.subplots(3, 4, figsize=(16, 12))
axes = axes.flatten()

for i in range(num_examples):
    plot_image_with_annotations(selected_image_files[i], selected_annotation_files[i], axes[i])

plt.tight_layout()
plt.show()
```



13.4. Creating a Dataset and Transformations

We'll create a custom dataset class (`PennFudanDatase`) that: - Loads an image and corresponding **bounding boxes** (and labels). - Applies transformations/augmentations (e.g., **random horizontal flips, resizing,** etc.).

We'll define a few **transformations** for images and bounding boxes. For instance:

- **Convert** PIL images to tensors
- Random **horizontal flip** with some probability

```
class PennFudanDataset(Dataset):
    def __init__(self, image_files, annotation_files, transforms=None):
        """
        Custom dataset that uses pre-loaded lists of image and annotation files.

        Args:
            image_files (list): Sorted list of image file paths.
            annotation_files (list): Sorted list of annotation file paths.
            transforms (callable, optional): A function/transform that takes in a PIL image and
                                             a target dict, and returns transformed versions.
        """
        assert len(image_files) == len(annotation_files), "Images and annotations lists must have the same
  length."
        self.image_files = image_files
        self.annotation_files = annotation_files
        self.transforms = transforms

    def __getitem__(self, idx):
        image_path = self.image_files[idx]
        annot_path = self.annotation_files[idx]

        img = Image.open(image_path).convert("RGB")
        boxes_and_labels = parse_annotation_file(annot_path)
        boxes = []
        labels = []
        for label_str, xmin, ymin, xmax, ymax in boxes_and_labels:
            boxes.append([xmin, ymin, xmax, ymax])
            labels.append(label_mapping.get(label_str, 0))

        boxes = torch.as_tensor(boxes, dtype=torch.float32)
        labels = torch.as_tensor(labels, dtype=torch.int64)
        target = {"boxes": boxes, "labels": labels}

        if self.transforms is not None:
            img, target = self.transforms(img, target)
        return img, target

    def __len__(self):
        return len(self.image_files)
```

```
class ComposeTransforms:
    """
    Composes a list of transformations to be applied sequentially to an image and its target.
    """
    def __init__(self, transforms_list):
        self.transforms_list = transforms_list

    def __call__(self, img, target):
        for t in self.transforms_list:
            img, target = t(img, target)
        return img, target

class ToTensor:
    """
    Converts a PIL image to a PyTorch tensor.
```

```
    """
    def __call__(self, img, target):
        img = TF.to_tensor(img)
        return img, target

class RandomHorizontalFlip:
    """
    Randomly horizontally flips an image and its bounding boxes with a given probability.
    """
    def __init__(self, flip_prob=0.5):
        self.flip_prob = flip_prob

    def __call__(self, img, target):
        if np.random.rand() < self.flip_prob:
            img = TF.hflip(img)
            boxes = target["boxes"]
            w = img.shape[2]
            boxes[:, [0, 2]] = w - boxes[:, [2, 0]]
            target["boxes"] = boxes
        return img, target

class ColorJitterTransform:
    """
    Randomly changes the brightness, contrast, saturation, and hue of an image.

    This transformation only modifies the image pixels and leaves the target (bounding boxes, etc.)
    unchanged.
    """
    def __init__(self, brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1):
        self.jitter = T.ColorJitter(brightness=brightness, contrast=contrast,
                                    saturation=saturation, hue=hue)

    def __call__(self, img, target):
        img = self.jitter(img)
        return img, target

class GaussianBlurTransform:
    """
    Applies a Gaussian blur to the image.

    This transformation only affects the image pixels. The kernel_size and sigma
    can be adjusted to control the amount of blurring.
    """
    def __init__(self, kernel_size=3, sigma=(0.1, 2.0)):
        self.kernel_size = kernel_size
        self.sigma = sigma

    def __call__(self, img, target):
        img = TF.gaussian_blur(img, kernel_size=self.kernel_size, sigma=self.sigma)
        return img, target

class NormalizeTransform:
    def __init__(self, mean, std):
        self.normalize = T.Normalize(mean, std)

    def __call__(self, img, target):
        img = self.normalize(img)
        return img, target
```

13.5. Building the Dataset

We will allocate the **Penn-Fudan dataset** into the following splits: - **70%** for the **training set**
- **15%** for the **validation set**
- **15%** for the **test set**

We **will not apply any transformations** to the test set. However, all images will be **normalized** to maintain consistency in input distributions across the training process.

```
mean = [0.485, 0.456, 0.406]
std = [0.229, 0.224, 0.225]

train_transforms = ComposeTransforms([
    ToTensor(),
    RandomHorizontalFlip(0.5),
    ColorJitterTransform(brightness=0.3, contrast=0.3, saturation=0.3, hue=0.05),
    GaussianBlurTransform(kernel_size=5, sigma=(0.5, 1.5)),
    NormalizeTransform(mean, std)
])

test_transforms = ComposeTransforms([
    ToTensor(),
    NormalizeTransform(mean, std)
])

dataset_train_full = PennFudanDataset(image_files, annotation_files, transforms=train_transforms)
dataset_test_full  = PennFudanDataset(image_files, annotation_files, transforms=test_transforms)

num_total = len(dataset_train_full)

torch.manual_seed(42)
indices = torch.randperm(len(dataset_train_full)).tolist()

# There are 170 images in total only
train_end = int(0.7 * num_total)
valid_end = train_end + int(0.15 * num_total)

train_idx = indices[:train_end]
valid_idx = indices[train_end:valid_end]
test_idx  = indices[valid_end:]

dataset_train = Subset(dataset_train_full, train_idx)
dataset_valid = Subset(dataset_test_full, valid_idx)
dataset_test  = Subset(dataset_test_full, test_idx)

train_loader = DataLoader(dataset_train, batch_size=8, shuffle=True, num_workers=2, collate_fn=lambda x:
  tuple(zip(*x)))
valid_loader = DataLoader(dataset_valid, batch_size=8, shuffle=True, num_workers=2, collate_fn=lambda x:
  tuple(zip(*x)))
test_loader  = DataLoader(dataset_test, batch_size=8, shuffle=False, num_workers=2, collate_fn=lambda x:
  tuple(zip(*x)))

print(f"Number of training samples: {len(dataset_train)}")
print(f"Number of validation samples: {len(dataset_valid)}")
print(f"Number of testing samples: {len(dataset_test)}")
```

```
Number of training samples: 118
Number of validation samples: 25
Number of testing samples: 27
```

13.6. Visualizing our Transformed Samples

To visualize our transformed samples using **`matplotlib`**, we need to **denormalize** the images so that they appear in their original form.

We will implement a **dynamically scaling grid** that can adapt to **any number of training samples**, allowing for flexible visualization of multiple transformed images at once.

```python
def denormalize_image(img_tensor, mean, std):
    """
    Denormalizes a normalized image tensor.

    Args:
        img_tensor (torch.Tensor): Normalized image tensor of shape (C, H, W).
        mean (list or tuple): Mean values used for normalization.
        std (list or tuple): Std values used for normalization.

    Returns:
        torch.Tensor: Denormalized image tensor in range [0, 1].
    """
    mean = torch.tensor(mean).view(-1, 1, 1)
    std = torch.tensor(std).view(-1, 1, 1)
    denorm = img_tensor * std + mean
    denorm = torch.clamp(denorm, 0, 1)
    return denorm

def visualize_transformed_samples(dataset, num_samples, class_names, mean, std):
    """
    Visualizes a number of transformed samples from the dataset with bounding boxes and labels.
    Denormalizes the images before plotting.

    Args:
        dataset (Dataset): The dataset (or subset) from which to get samples.
        num_samples (int): The number of random samples to visualize.
        class_names (list): List of class names corresponding to label indices.
        mean (list): Mean values used for normalization.
        std (list): Std values used for normalization.
    """
    indices = random.sample(range(len(dataset)), num_samples)

    # try to make the grid roughly square
    n_cols = int(math.sqrt(num_samples))
    n_rows = math.ceil(num_samples / n_cols)

    fig, axes = plt.subplots(n_rows, n_cols, figsize=(8 * n_cols, 8 * n_rows))

    # Flatten axes array for easier iteration (if there's only one row/col, make it a list)
    if n_rows * n_cols == 1:
        axes = [axes]
    else:
        axes = axes.flatten()

    for i, idx in enumerate(indices):
        img, target = dataset[idx]
        img_denorm = denormalize_image(img, mean, std)
        image_np = img_denorm.permute(1, 2, 0).cpu().numpy()

        axes[i].imshow(image_np)
        axes[i].set_title(f"Sample index: {idx}")
```

```
        boxes = target["boxes"]
        labels = target["labels"]

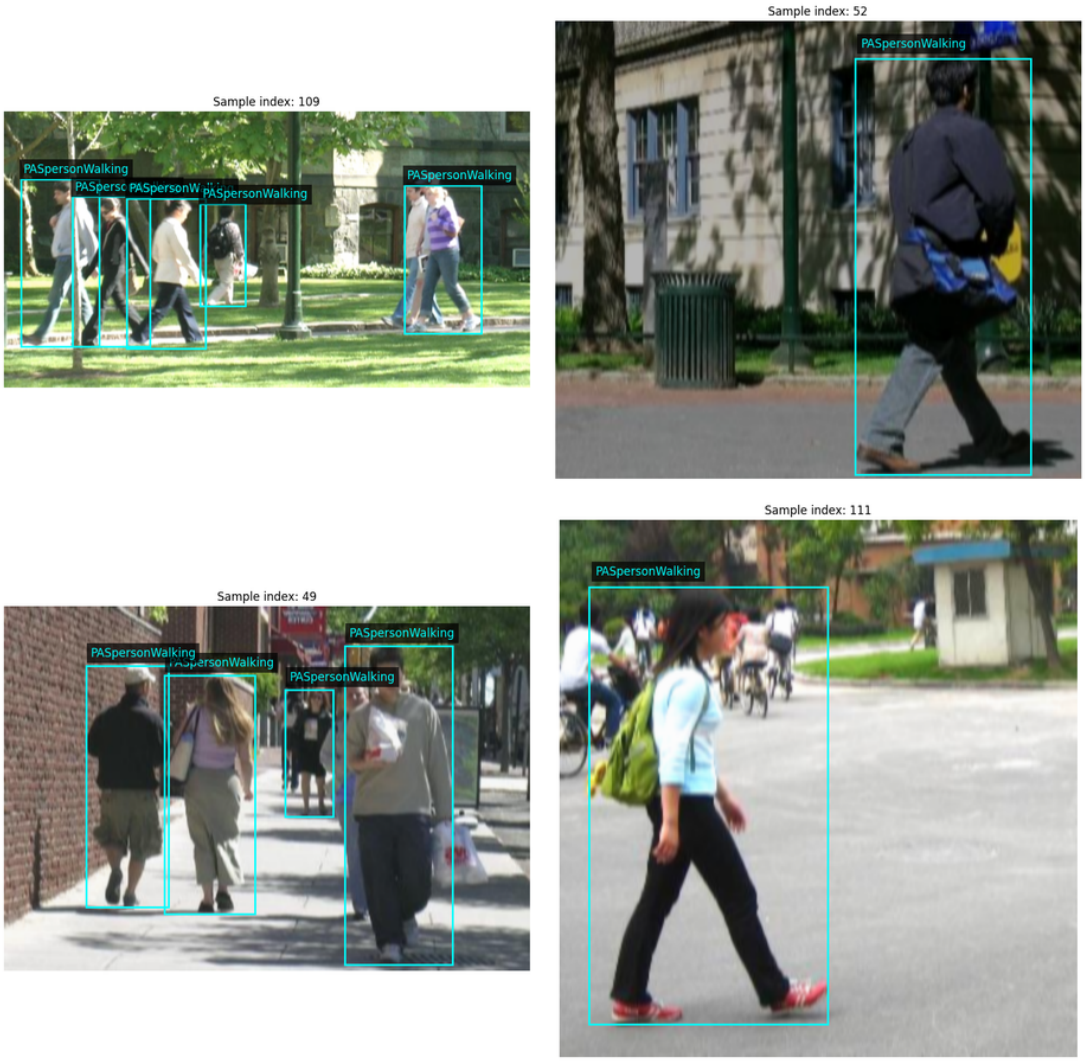
        for j, box in enumerate(boxes):
            xmin, ymin, xmax, ymax = box.tolist()
            width, height = xmax - xmin, ymax - ymin
            label_text = class_names[labels[j]]

            rect = patches.Rectangle((xmin, ymin), width, height, linewidth=2, edgecolor='cyan',
    facecolor='none')
            axes[i].add_patch(rect)
            axes[i].text(xmin + 4, max(0, ymin - 6), label_text, fontsize=12, color='cyan',
                         backgroundcolor='red', verticalalignment='bottom',
                         bbox=dict(facecolor='black', alpha=0.7, edgecolor='none'))
        axes[i].axis("off")

    # Turn off any extra subplots if num_samples doesn't fill the grid
    for j in range(i + 1, len(axes)):
        axes[j].axis("off")

    plt.tight_layout()
    plt.show()
```

```
visualize_transformed_samples(dataset_train, num_samples=4, class_names=classes, mean=mean, std=std)
```



13.7. Two-stage vs. single-stage detectors

Faster R-CNN is the canonical *two-stage* detector. In the first stage, a Region Proposal Network (RPN) scans the feature map and produces a few thousand candidate boxes likely to contain an object. In the second stage, each proposal is cropped (via RoI Align), passed through a small classification head, and refined.

Single-stage detectors (YOLO, SSD, RetinaNet) skip the RPN: the detection head predicts boxes and classes directly on a dense feature grid. Fewer stages means lower latency, a well-tuned YOLO runs several times faster than Faster R-CNN at equivalent input resolution.

When to pick which. Two-stage detectors tend to be more accurate on small and overlapping objects because the RPN funnels computation toward plausible regions, and the second-stage head can afford to be heavier per proposal. Single-stage detectors win on inference speed and ease of deployment. For research and benchmarks where a few extra percent of mAP matters, Faster R-CNN remains a strong baseline. For real-time applications, single-stage is the default.

One practical consequence: Faster R-CNN produces post-NMS outputs that are directly `(box, score, class)` triples, while YOLO decoding requires an extra step to unpack grid predictions. The training pipelines also differ in their loss formulations (classification + regression on proposals vs. per-cell multi-task loss), which matters when you want to customise a detector for a non-standard problem.

13.8. Building the Fast R-CNN Model

A **pre-trained Faster R-CNN model with a ResNet-50 [13] backbone and Feature Pyramid Network (FPN)** is loaded from `torchvision`, initialized with default weights. The **classification head is replaced** to match the number of target classes by adjusting the fully connected layer in the `box_predictor`.

The model summary reveals that it includes a **built-in normalization layer** tailored for the **COCO dataset**. While our prior normalization step is not strictly necessary, it does not cause any issues, rather, it serves as a useful **exercise in preprocessing** that can be applied to other tasks where built-in normalization is not available.

```
model = torchvision.models.detection.fasterrcnn_resnet50_fpn(weights="DEFAULT")
model
```

```
Downloading:
"https://download.pytorch.org/models/fasterrcnn_resnet50_fpn_coco-258fb6c6.pth"
to /root/.cache/torch/hub/checkpoints/fasterrcnn_resnet50_fpn_coco-258fb6c6.pth
100%|##########| 160M/160M [00:00<00:00, 170MB/s]
```

```
FasterRCNN(
  (transform): GeneralizedRCNNTransform(
      Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
      Resize(min_size=(800,), max_size=1333, mode='bilinear')
  )
```

```
  (backbone): BackboneWithFPN(
    (body): IntermediateLayerGetter(
      (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
      (bn1): FrozenBatchNorm2d(64, eps=0.0)
      (relu): ReLU(inplace=True)
      (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
      (layer1): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(64, eps=0.0)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
          (bn2): FrozenBatchNorm2d(64, eps=0.0)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(256, eps=0.0)
          (relu): ReLU(inplace=True)
[... 198 lines omitted ...]
    (box_head): TwoMLPHead(
      (fc6): Linear(in_features=12544, out_features=1024, bias=True)
      (fc7): Linear(in_features=1024, out_features=1024, bias=True)
    )
    (box_predictor): FastRCNNPredictor(
      (cls_score): Linear(in_features=1024, out_features=91, bias=True)
      (bbox_pred): Linear(in_features=1024, out_features=364, bias=True)
    )
  )
)
```

```
# Replace the classifier head
in_features = model.roi_heads.box_predictor.cls_score.in_features
model.roi_heads.box_predictor = FastRCNNPredictor(in_features, len(classes))
model = model.to(device)
model
```

```
FasterRCNN(
  (transform): GeneralizedRCNNTransform(
      Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
      Resize(min_size=(800,), max_size=1333, mode='bilinear')
  )
  (backbone): BackboneWithFPN(
    (body): IntermediateLayerGetter(
      (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
      (bn1): FrozenBatchNorm2d(64, eps=0.0)
      (relu): ReLU(inplace=True)
      (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
      (layer1): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
          (bn1): FrozenBatchNorm2d(64, eps=0.0)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
          (bn2): FrozenBatchNorm2d(64, eps=0.0)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(256, eps=0.0)
          (relu): ReLU(inplace=True)
[... 198 lines omitted ...]
    (box_head): TwoMLPHead(
      (fc6): Linear(in_features=12544, out_features=1024, bias=True)
      (fc7): Linear(in_features=1024, out_features=1024, bias=True)
    )
    (box_predictor): FastRCNNPredictor(
      (cls_score): Linear(in_features=1024, out_features=2, bias=True)
      (bbox_pred): Linear(in_features=1024, out_features=8, bias=True)
    )
  )
)
```

```
model.eval()
dummy_image = torch.randn(3, 300, 300).unsqueeze(0).to(device)

with torch.no_grad():
    output = model(dummy_image)
    print("Model output:", output)

summary(model, input_data=dummy_image)
```

```
Model output: [{'boxes': tensor([[2.8041e+00, 2.0822e+02, 6.4758e+00,
2.1313e+02],
        [2.0548e+02, 0.0000e+00, 2.2659e+02, 2.0755e+01],
        [2.7085e+00, 5.1888e+01, 6.1349e+00, 5.5093e+01],
        [6.8275e+01, 1.7613e+02, 7.0763e+01, 1.7971e+02],
        [2.6474e+02, 8.8431e-02, 2.8990e+02, 2.1902e+01],
        [2.0412e+00, 5.0108e+01, 5.8884e+00, 5.4168e+01],
        [2.9318e+02, 1.6351e+02, 2.9576e+02, 1.6717e+02],
        [6.9321e+01, 1.7614e+02, 7.2219e+01, 1.7983e+02],
        [2.3286e+02, 7.4405e+01, 2.3677e+02, 7.7722e+01],
        [1.6058e+02, 1.4452e+01, 1.6444e+02, 1.8079e+01],
        [2.7661e+02, 1.6514e+02, 2.8025e+02, 1.6891e+02],
        [6.4962e+01, 1.7185e+02, 6.7984e+01, 1.7589e+02],
        [1.6175e+02, 1.5570e+01, 1.6580e+02, 1.9257e+01],
        [6.3076e+01, 1.7309e+02, 6.6764e+01, 1.7625e+02],
        [2.8055e+02, 9.2166e+01, 2.8438e+02, 9.6353e+01],
        [2.8750e+02, 1.5853e+01, 2.9008e+02, 1.9472e+01],
        [5.5493e+01, 2.1037e+02, 7.8340e+01, 2.3558e+02],
        [2.7615e+00, 1.6362e+02, 5.5564e+00, 1.6822e+02],
        [3.9235e+01, 1.8517e+02, 4.2641e+01, 1.8946e+02],
        [2.7482e+02, 6.0753e-01, 2.9706e+02, 2.1583e+01],
[... 85 lines omitted ...]
        0.5073, 0.5070, 0.5060, 0.5046, 0.5045, 0.5020, 0.5020, 0.5019, 0.5002,
        0.5000, 0.4997, 0.4990, 0.4982, 0.4973, 0.4951, 0.4944, 0.4943, 0.4940,
```

```
        0.4937, 0.4934, 0.4931, 0.4928, 0.4927, 0.4925, 0.4924, 0.4921, 0.4910,
        0.4910, 0.4909, 0.4906, 0.4902, 0.4899, 0.4892, 0.4892, 0.4891, 0.4886,
        0.4884, 0.4883, 0.4882, 0.4881, 0.4877, 0.4875, 0.4873, 0.4872, 0.4866,
        0.4866, 0.4856, 0.4850, 0.4848, 0.4845, 0.4838, 0.4838, 0.4836, 0.4833,
        0.4832, 0.4825, 0.4823, 0.4819, 0.4819, 0.4815, 0.4809, 0.4792, 0.4790,
        0.4788, 0.4787, 0.4787, 0.4783, 0.4783, 0.4781, 0.4780, 0.4779, 0.4774,
        0.4773, 0.4766, 0.4764, 0.4761, 0.4761, 0.4753, 0.4752, 0.4751, 0.4748,
        0.4746], device='cuda:0')}]
```

```
==========================================================================
========================
Layer (type:depth-idx)                                   Output Shape
Param #
==========================================================================
========================
FasterRCNN                                               [100, 4]
--
+-GeneralizedRCNNTransform: 1-1                          [1, 3, 800, 800]
--
+-BackboneWithFPN: 1-2                                   [1, 256, 13, 13]
--
|    +-IntermediateLayerGetter: 2-1                      [1, 2048, 25, 25]
--
|    |    +-Conv2d: 3-1                                  [1, 64, 400, 400]
(9,408)
|    |    +-FrozenBatchNorm2d: 3-2                       [1, 64, 400, 400]
--
|    |    +-ReLU: 3-3                                   [1, 64, 400, 400]
--
|    |    +-MaxPool2d: 3-4                              [1, 64, 200, 200]
--
|    |    +-Sequential: 3-5                             [1, 256, 200, 200]
(212,992)
|    |    +-Sequential: 3-6                             [1, 512, 100, 100]
1,212,416
|    |    +-Sequential: 3-7                             [1, 1024, 50, 50]
7,077,888
|    |    +-Sequential: 3-8                             [1, 2048, 25, 25]
14,942,208
|    +-FeaturePyramidNetwork: 2-2                       [1, 256, 13, 13]
--
|    |    +-ModuleList: 3-15                            --
(recursive)
|    |    +-ModuleList: 3-16                            --
(recursive)
|    |    +-ModuleList: 3-15                            --
(recursive)
|    |    +-ModuleList: 3-16                            --
(recursive)
[... 32 lines omitted ...]
Total params: 41,299,161
Trainable params: 41,076,761
```

```
Non-trainable params: 222,400
Total mult-adds (Units.GIGABYTES): 133.97
==========================================================================================
==========================
Input size (MB): 1.08
Forward/backward pass size (MB): 1483.69
Params size (MB): 165.20
Estimated Total Size (MB): 1649.96
==========================================================================================
==========================
```

13.9. Training

For fine-tuning, we use a **small batch size of 4** to accommodate the **memory constraints of a T4 GPU**. **SGD (Stochastic Gradient Descent)** is chosen as the optimizer, as it is well-suited for **small-scale fine-tuning tasks**, providing stable updates and better generalization. All **trainable parameters** (`requires_grad=True`) are optimized to allow effective adaptation of the pre-trained model.

```python
for name, param in model.named_parameters():
    print(f"{name}: requires_grad = {param.requires_grad}")

for param in model.backbone.body.layer2.parameters():
    param.requires_grad = False

params = [p for p in model.parameters() if p.requires_grad]
optimizer = optim.SGD(params, lr=0.005, momentum=0.9, weight_decay=0.0005)
lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=3, gamma=0.1)
```

```
backbone.body.conv1.weight: requires_grad = False
backbone.body.layer1.0.conv1.weight: requires_grad = False
backbone.body.layer1.0.conv2.weight: requires_grad = False
backbone.body.layer1.0.conv3.weight: requires_grad = False
backbone.body.layer1.0.downsample.0.weight: requires_grad = False
backbone.body.layer1.1.conv1.weight: requires_grad = False
backbone.body.layer1.1.conv2.weight: requires_grad = False
backbone.body.layer1.1.conv3.weight: requires_grad = False
backbone.body.layer1.2.conv1.weight: requires_grad = False
backbone.body.layer1.2.conv2.weight: requires_grad = False
backbone.body.layer1.2.conv3.weight: requires_grad = False
backbone.body.layer2.0.conv1.weight: requires_grad = True
backbone.body.layer2.0.conv2.weight: requires_grad = True
backbone.body.layer2.0.conv3.weight: requires_grad = True
backbone.body.layer2.0.downsample.0.weight: requires_grad = True
backbone.body.layer2.1.conv1.weight: requires_grad = True
backbone.body.layer2.1.conv2.weight: requires_grad = True
backbone.body.layer2.1.conv3.weight: requires_grad = True
backbone.body.layer2.2.conv1.weight: requires_grad = True
backbone.body.layer2.2.conv2.weight: requires_grad = True
[... 53 lines omitted ...]
rpn.head.bbox_pred.weight: requires_grad = True
rpn.head.bbox_pred.bias: requires_grad = True
roi_heads.box_head.fc6.weight: requires_grad = True
roi_heads.box_head.fc6.bias: requires_grad = True
roi_heads.box_head.fc7.weight: requires_grad = True
roi_heads.box_head.fc7.bias: requires_grad = True
roi_heads.box_predictor.cls_score.weight: requires_grad = True
```

```
roi_heads.box_predictor.cls_score.bias: requires_grad = True
roi_heads.box_predictor.bbox_pred.weight: requires_grad = True
roi_heads.box_predictor.bbox_pred.bias: requires_grad = True
```

```python
def print_epoch_summary(epoch, train_loss, train_loss_classifier, train_loss_box_reg,
                        train_loss_objectness, train_loss_rpn_box_reg,
                        val_loss, val_loss_classifier, val_loss_box_reg,
                        val_loss_objectness, val_loss_rpn_box_reg, epoch_time):
    print(f"\nEpoch [{epoch+1}] Summary:")
    print("-----------------------------------------------------------")
    print(f"Training Loss: {train_loss:.4f}")
    print(f"   Classifier Loss: {train_loss_classifier:.4f}")
    print(f"   Box Regression Loss: {train_loss_box_reg:.4f}")
    print(f"   Objectness Loss: {train_loss_objectness:.4f}")
    print(f"   RPN Box Regression Loss: {train_loss_rpn_box_reg:.4f}")
    print("-----------------------------------------------------------")
    print(f"Validation Loss: {val_loss:.4f}")
    print(f"   Classifier Loss: {val_loss_classifier:.4f}")
    print(f"   Box Regression Loss: {val_loss_box_reg:.4f}")
    print(f"   Objectness Loss: {val_loss_objectness:.4f}")
    print(f"   RPN Box Regression Loss: {val_loss_rpn_box_reg:.4f}")
    print("===========================================================")
    print(f"Epoch Time: {epoch_time:.2f} sec\n")
```

The built-in `fasterrcnn_resnet50_fpn` model includes an **integrated loss function** that computes a **composite loss** based on four key components:

- **Classifier Loss**: Measures classification errors for detected objects.
- **Box Regression Loss**: Evaluates how well the predicted bounding boxes align with ground truth.
- **Objectness Loss**: Determines whether a region proposal contains an object.
- **RPN Box Regression Loss**: Assesses the accuracy of region proposal bounding boxes.

When using this model, we do not need to manually define a loss function. In **training mode**, the model automatically returns a **loss dictionary** instead of standard outputs, allowing us to track running losses. In **evaluation mode**, the model outputs **predictions** rather than loss values.

```python
num_epochs = 10
model.train()

best_val_loss = float('inf')

# Histories
train_loss_history = []
train_loss_classifier_history = []
train_loss_box_reg_history = []
train_loss_objectness_history = []
train_loss_rpn_box_reg_history = []
val_loss_history = []
val_loss_classifier_history = []
val_loss_box_reg_history = []
val_loss_objectness_history = []
val_loss_rpn_box_reg_history = []
```

```
for epoch in range(num_epochs):
    start_time = time.time()
    model.train()

    train_loss = 0.0
    train_loss_classifier = 0.0
    train_loss_box_reg = 0.0
    train_loss_objectness = 0.0
    train_loss_rpn_box_reg = 0.0
    train_samples = 0

    # ---------------------- Training Loop ---------------------- #
    for images, targets in train_loader:
        images = [img.to(device) for img in images]
        targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

        loss_dict = model(images, targets)

        loss_classifier = loss_dict['loss_classifier']
        loss_box_reg = loss_dict['loss_box_reg']
        loss_objectness = loss_dict['loss_objectness']
        loss_rpn_box_reg = loss_dict['loss_rpn_box_reg']

        losses = loss_classifier + loss_box_reg + loss_objectness + loss_rpn_box_reg

        optimizer.zero_grad()
        losses.backward()
        optimizer.step()

        batch_size = len(images)
        train_loss += losses.item() * batch_size
        train_loss_classifier += loss_classifier.item() * batch_size
        train_loss_box_reg += loss_box_reg.item() * batch_size
        train_loss_objectness += loss_objectness.item() * batch_size
        train_loss_rpn_box_reg += loss_rpn_box_reg.item() * batch_size
        train_samples += batch_size

    # Average training losses
    train_loss /= train_samples
    train_loss_classifier /= train_samples
    train_loss_box_reg /= train_samples
    train_loss_objectness /= train_samples
    train_loss_rpn_box_reg /= train_samples

    train_loss_history.append(train_loss)
    train_loss_classifier_history.append(train_loss_classifier)
    train_loss_box_reg_history.append(train_loss_box_reg)
    train_loss_objectness_history.append(train_loss_objectness)
    train_loss_rpn_box_reg_history.append(train_loss_rpn_box_reg)

    # ---------------------- Validation Loop ---------------------- #
    # Use torch.no_grad() during validation to save memory.
    model.train()  # Still need to be in train mode for loss computation with targets.
    val_loss = 0.0
    val_loss_classifier = 0.0
    val_loss_box_reg = 0.0
    val_loss_objectness = 0.0
    val_loss_rpn_box_reg = 0.0
    val_samples = 0
```

```
    with torch.no_grad():
        for images, targets in valid_loader:
            images = [img.to(device) for img in images]
            targets = [{k: v.to(device) for k, v in t.items()} for t in targets]

            loss_dict = model(images, targets)
            losses = sum(loss for loss in loss_dict.values())

            loss_classifier = loss_dict['loss_classifier']
            loss_box_reg = loss_dict['loss_box_reg']
            loss_objectness = loss_dict['loss_objectness']
            loss_rpn_box_reg = loss_dict['loss_rpn_box_reg']

            batch_size = len(images)
            val_loss += losses.item() * batch_size
            val_loss_classifier += loss_classifier.item() * batch_size
            val_loss_box_reg += loss_box_reg.item() * batch_size
            val_loss_objectness += loss_objectness.item() * batch_size
            val_loss_rpn_box_reg += loss_rpn_box_reg.item() * batch_size
            val_samples += batch_size

    val_loss /= val_samples
    val_loss_classifier /= val_samples
    val_loss_box_reg /= val_samples
    val_loss_objectness /= val_samples
    val_loss_rpn_box_reg /= val_samples

    val_loss_history.append(val_loss)
    val_loss_classifier_history.append(val_loss_classifier)
    val_loss_box_reg_history.append(val_loss_box_reg)
    val_loss_objectness_history.append(val_loss_objectness)
    val_loss_rpn_box_reg_history.append(val_loss_rpn_box_reg)

    lr_scheduler.step()

    epoch_time = time.time() - start_time
    print_epoch_summary(epoch, train_loss, train_loss_classifier, train_loss_box_reg,
                        train_loss_objectness, train_loss_rpn_box_reg,
                        val_loss, val_loss_classifier, val_loss_box_reg,
                        val_loss_objectness, val_loss_rpn_box_reg, epoch_time)

    torch.cuda.empty_cache()

    # Save best model
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        torch.save(model.state_dict(), 'best_model_frcnn.pth')
        print("Saved best model.")
```

```
Epoch [1] Summary:
------------------------------------------------------------
Training Loss: 0.4623
   Classifier Loss: 0.2002
   Box Regression Loss: 0.1897
   Objectness Loss: 0.0541
   RPN Box Regression Loss: 0.0182
------------------------------------------------------------
Validation Loss: 0.2964
   Classifier Loss: 0.1049
   Box Regression Loss: 0.1591
```

```
   Objectness Loss: 0.0193
   RPN Box Regression Loss: 0.0131
============================================================
Epoch Time: 36.92 sec

Saved best model.

Epoch [2] Summary:
[... 145 lines omitted ...]
   Objectness Loss: 0.0020
   RPN Box Regression Loss: 0.0075
------------------------------------------------------------
Validation Loss: 0.1696
   Classifier Loss: 0.0559
   Box Regression Loss: 0.1054
   Objectness Loss: 0.0012
   RPN Box Regression Loss: 0.0070
============================================================
Epoch Time: 39.39 sec
```

Saving our Model

```
model_save_path = 'best_model_frcnn.pth'

# Save the model's state dictionary
torch.save(model.state_dict(), model_save_path)
```

Plotting Losses

```
def plot_losses(train_loss_history, train_loss_classifier_history, train_loss_box_reg_history,
                train_loss_objectness_history, train_loss_rpn_box_reg_history,
                val_loss_history, val_loss_classifier_history, val_loss_box_reg_history,
                val_loss_objectness_history, val_loss_rpn_box_reg_history):
    """
    Plots training and validation loss histories for overall loss and each individual loss component.

    Args:
        train_loss_history (list): Overall training loss per epoch.
        train_loss_classifier_history (list): Training classifier loss per epoch.
        train_loss_box_reg_history (list): Training box regression loss per epoch.
        train_loss_objectness_history (list): Training objectness loss per epoch.
        train_loss_rpn_box_reg_history (list): Training RPN box regression loss per epoch.
        val_loss_history (list): Overall validation loss per epoch.
        val_loss_classifier_history (list): Validation classifier loss per epoch.
        val_loss_box_reg_history (list): Validation box regression loss per epoch.
        val_loss_objectness_history (list): Validation objectness loss per epoch.
        val_loss_rpn_box_reg_history (list): Validation RPN box regression loss per epoch.
    """
    epochs = range(1, len(train_loss_history) + 1)
    plt.figure(figsize=(12, 8))

    # Plot overall loss
    plt.plot(epochs, train_loss_history, 'b-', label='Train Overall Loss')
    plt.plot(epochs, val_loss_history, 'b--', label='Val Overall Loss')

    # Plot classifier loss
    plt.plot(epochs, train_loss_classifier_history, 'r-', label='Train Classifier Loss')
    plt.plot(epochs, val_loss_classifier_history, 'r--', label='Val Classifier Loss')

    # Plot box regression loss
```

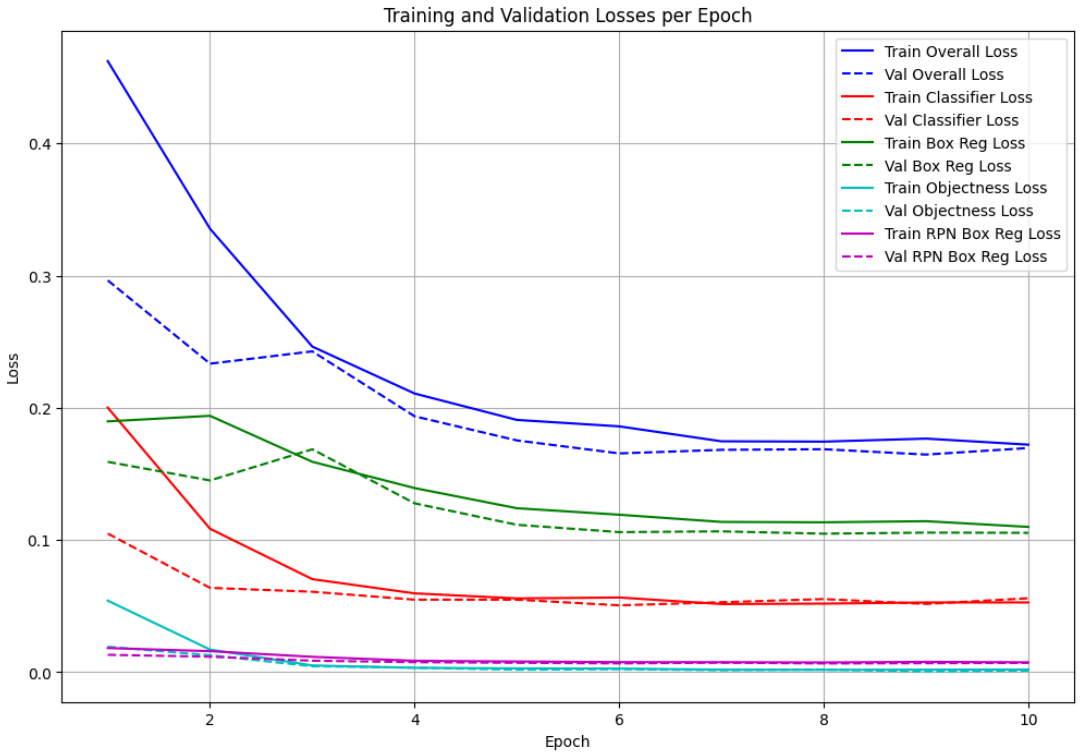
```
    plt.plot(epochs, train_loss_box_reg_history, 'g-', label='Train Box Reg Loss')
    plt.plot(epochs, val_loss_box_reg_history, 'g--', label='Val Box Reg Loss')

    # Plot objectness loss
    plt.plot(epochs, train_loss_objectness_history, 'c-', label='Train Objectness Loss')
    plt.plot(epochs, val_loss_objectness_history, 'c--', label='Val Objectness Loss')

    # Plot RPN box regression loss
    plt.plot(epochs, train_loss_rpn_box_reg_history, 'm-', label='Train RPN Box Reg Loss')
    plt.plot(epochs, val_loss_rpn_box_reg_history, 'm--', label='Val RPN Box Reg Loss')

    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Training and Validation Losses per Epoch')
    plt.legend()
    plt.grid(True)
    plt.show()
```

```
plot_losses(train_loss_history, train_loss_classifier_history, train_loss_box_reg_history,
            train_loss_objectness_history, train_loss_rpn_box_reg_history,
            val_loss_history, val_loss_classifier_history, val_loss_box_reg_history,
            val_loss_objectness_history, val_loss_rpn_box_reg_history)
```



The plot demonstrates that the **pre-trained Faster R-CNN model** was already highly effective at detecting **person bounding boxes** in images. This is because the **pre-trained weights** were trained on a **large-scale dataset (e.g., COCO)** that includes similar images of people walking. As a result, our **fine-tuning did not yield significant improvements**, since the model was already well-suited for the task.

Loading in the Model

```
model.load_state_dict(torch.load('best_model_frcnn.pth'))
model.eval()
```

```
<ipython-input-22-1ddb19c98d52>:1: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. ... [725 chars truncated]
  model.load_state_dict(torch.load('best_model_frcnn.pth'))
```

```
FasterRCNN(
  (transform): GeneralizedRCNNTransform(
      Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
      Resize(min_size=(800,), max_size=1333, mode='bilinear')
  )
  (backbone): BackboneWithFPN(
    (body): IntermediateLayerGetter(
      (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
bias=False)
      (bn1): FrozenBatchNorm2d(64, eps=0.0)
      (relu): ReLU(inplace=True)
      (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
      (layer1): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(64, eps=0.0)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
          (bn2): FrozenBatchNorm2d(64, eps=0.0)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1),
bias=False)
          (bn3): FrozenBatchNorm2d(256, eps=0.0)
          (relu): ReLU(inplace=True)
[... 198 lines omitted ...]
    (box_head): TwoMLPHead(
      (fc6): Linear(in_features=12544, out_features=1024, bias=True)
      (fc7): Linear(in_features=1024, out_features=1024, bias=True)
    )
    (box_predictor): FastRCNNPredictor(
      (cls_score): Linear(in_features=1024, out_features=2, bias=True)
      (bbox_pred): Linear(in_features=1024, out_features=8, bias=True)
    )
  )
)
```

13.10. Testing our Model

For testing purposes, we will compute the **mean Average Precision (mAP)** at different **Intersection over Union (IoU) thresholds**. The function **`compute_iou`** calculates the **IoU values**, while **`evaluate_predictions`** computes the **mAP** at a specified **IoU threshold**.

The **IoU** between two bounding boxes is computed as:

$$IoU = \frac{|A \cap B|}{|A \cup B|}$$

where: - A and B are the predicted and ground truth bounding boxes, respectively. - $|A \cap B|$ represents the **area of intersection** between the two bounding boxes. - $|A \cup B|$ represents the **area of their union**.

The **Average Precision (AP)** at a given IoU threshold is computed as:

$$AP = \sum_{n} (R_n - R_{n-1}) P_n$$

where: - P_n and R_n are the **precision** and **recall** at the nth threshold, respectively.

The **mean Average Precision (mAP)** is then obtained by averaging the **AP values** across multiple **IoU thresholds**:

$$mAP = \frac{1}{T} \sum_{t=1}^{T} AP_{IoU_t}$$

where: - T is the total number of IoU thresholds. - AP_{IoU_t} is the **AP** at the tth **IoU threshold**.

This metric provides a comprehensive evaluation of the model's detection performance across varying degrees of localization accuracy.

A prediction looks like the following:

```
{'boxes': tensor([[119.7279,  45.0536, 244.8307, 332.0854],
        [115.9338,  62.2116, 179.0104, 319.6712],
        [144.6216,  50.3902, 217.2193, 209.9409],
        [  1.6695,  52.2323,  22.4929, 112.4107],
        [108.8810,  67.1334, 173.3511, 201.6209]], device='cuda:0'), 'labels': tensor([1, 1, 1, 1, 1], device='cuda:0'), 'scores': t
```

```python
def compute_iou(box1, box2):
    """
    Compute Intersection over Union (IoU) between two boxes.

    Args:
        box1 (array-like): [x1, y1, x2, y2] for the first box.
        box2 (array-like): [x1, y1, x2, y2] for the second box.

    Returns:
        float: IoU value.
    """
    xA = max(box1[0], box2[0])
    yA = max(box1[1], box2[1])
    xB = min(box1[2], box2[2])
    yB = min(box1[3], box2[3])

    interW = max(0, xB - xA)
    interH = max(0, yB - yA)
    interArea = interW * interH
    box1Area = (box1[2] - box1[0]) * (box1[3] - box1[1])
    box2Area = (box2[2] - box2[0]) * (box2[3] - box2[1])

    unionArea = box1Area + box2Area - interArea
    iou = interArea / unionArea if unionArea > 0 else 0
```

```
    return iou

def evaluate_detections(predictions, ground_truths, iou_threshold=0.5):
    """
    Evaluates detections for non-background classes and computes a simplified average precision (AP)
    at the given IoU threshold. The background class (label 0) is ignored.

    Args:
        predictions (list): List of dicts for each image with keys 'boxes', 'scores', and 'labels'.
        ground_truths (list): List of dicts for each image with keys 'boxes' and 'labels'.
        iou_threshold (float): IoU threshold for considering a prediction as a true positive.

    Returns:
        tuple: (ap_per_class, mAP) where ap_per_class is a dict mapping class labels (non-zero)
               to their average precision, and mAP is the mean average precision over those classes.
    """
    pred_by_class = {}
    gt_by_class = {}

    for preds, gts in zip(predictions, ground_truths):
        # Gets all the unique labels in an image (Here: 0 and 1)
        labels_in_image = np.unique(np.concatenate([preds['labels'].cpu().numpy(), gts['labels'].cpu().
  numpy()]))
        for label in labels_in_image:
            if label == 0:
                continue  # Skip background

            # If label is not already a key in pred_by_class or gt_by_class, it adds that key with an empty
  list as its value
            pred_by_class.setdefault(label, [])
            gt_by_class.setdefault(label, [])

            # For predictions: store each detection as (score, box, matched flag)
            # Get all label indices that match our pedestrian label
            inds = (preds['labels'] == label).nonzero().squeeze(1)
            # Store all predictions for that label in a dict
            for idx in inds:
                pred_by_class[label].append({
                    'score': preds['scores'][idx].item(),
                    'box': preds['boxes'][idx].cpu().numpy(),
                    'matched': False
                })
            # For ground truths: simply store each box
            inds = (gts['labels'] == label).nonzero().squeeze(1)
            for idx in inds:
                gt_by_class[label].append(gts['boxes'][idx].cpu().numpy())

    ap_per_class = {}
    for label in gt_by_class.keys():
        # Sort predictions by decreasing score for a specific label
        preds = sorted(pred_by_class[label], key=lambda x: x['score'], reverse=True)

        # Number of ground truths for a specific label
        num_gt = len(gt_by_class[label])

        # Set True Positives and False Positives to 0
        tp = np.zeros(len(preds))
        fp = np.zeros(len(preds))

        # Compute IoU for GT bbox and PRED bbox
```

```
        # We iterate through the ground truth bboxes for the label
        # We find the best match for it (based on Iou)
        # If the match has not been marked before, then we set the match to True
        # Then increase TP by 1
        for i, pred in enumerate(preds):
            best_iou = 0
            for gt_box in gt_by_class[label]:
                iou = compute_iou(pred['box'], gt_box)
                if iou > best_iou:
                    best_iou = iou
            if best_iou >= iou_threshold:
                if not pred['matched']:
                    tp[i] = 1
                    pred['matched'] = True
                else:
                    fp[i] = 1
            else:
                fp[i] = 1

        # Compute cumulative sums
        tp_cumsum = np.cumsum(tp)
        fp_cumsum = np.cumsum(fp)

        # Calculating Prec and Recall
        precisions = tp_cumsum / (tp_cumsum + fp_cumsum + 1e-10)
        recalls = tp_cumsum / (num_gt + 1e-10)

        # Simplified AP calculation: average precision at fixed recall thresholds
        # 20-point interpolation approximation of Average Precision (AP)
        # It generates 20 evenly spaced recall thresholds between 0 and 1
        # For each threshold, it finds the maximum precision value achieved at any recall level greater than
  or equal to that threshold
        # It sums these maximum precision values and divides by 20 to obtain the average
        ap = 0
        for r_thresh in np.linspace(0, 1, 20):
            precisions_at_recall = precisions[recalls >= r_thresh]
            if precisions_at_recall.size > 0:
                ap += np.max(precisions_at_recall)
        ap /= 20.0
        ap_per_class[label] = ap

    mAP = np.mean(list(ap_per_class.values()))
    return ap_per_class, mAP
```

Inferring our Whole Test Set

```
all_predictions = []
all_ground_truths = []
with torch.no_grad():
    for images, targets in test_loader:
        images = [img.to(device) for img in images]
        preds = model(images)
        all_predictions.extend(preds)
        all_ground_truths.extend(targets)
```

Computing IoU

```
ap_per_class, mAP = evaluate_detections(all_predictions, all_ground_truths, iou_threshold=0.5)
print("Average Precision per class:", ap_per_class)
print(f"Mean Average Precision (mAP) @ IoU 0.5: {mAP:.4f}\n")
```

```
ap_per_class, mAP = evaluate_detections(all_predictions, all_ground_truths, iou_threshold=0.9)
print("Average Precision per class:", ap_per_class)
print(f"Mean Average Precision (mAP) @ IoU 0.9: {mAP:.4f}")
```

```
Average Precision per class: {1: 0.9874125874095974}
Mean Average Precision (mAP) @ IoU 0.5: 0.9874

Average Precision per class: {1: 0.09788934897105692}
Mean Average Precision (mAP) @ IoU 0.9: 0.0979
```

Visualizing Results

```
def visualize_detections(image, predictions, ground_truth, class_names, threshold=0.5):
    """
    Visualizes the predicted and ground truth bounding boxes on the image.

    Args:
        image (np.ndarray): Image array in [H, W, 3] format.
        predictions (dict): Dictionary with keys 'boxes', 'labels', 'scores' from the model.
        ground_truth (dict): Dictionary with keys 'boxes', 'labels'.
        class_names (list): List of class names.
        threshold (float): Score threshold to display a prediction.
    """
    fig, ax = plt.subplots(1, figsize=(12, 8))
    ax.imshow(image)

    # Draw ground truth boxes in cyan
    for box, label in zip(ground_truth['boxes'], ground_truth['labels']):
        x1, y1, x2, y2 = box.cpu().numpy()
        rect = patches.Rectangle((x1, y1), x2 - x1, y2 - y1, linewidth=2, edgecolor='cyan', facecolor='none')
        ax.add_patch(rect)
        ax.text(x1 + 4, max(0, y1 - 4), f"GT: {class_names[label.item()]}", color='cyan', fontsize=9,
  verticalalignment='bottom', bbox=dict(facecolor='black', alpha=0.7, edgecolor='none'))

    # Draw predicted boxes in orange (if above score threshold)
    for box, label, score in zip(predictions['boxes'], predictions['labels'], predictions['scores']):
        if score.item() < threshold:
            continue
        x1, y1, x2, y2 = box.cpu().numpy()
        rect = patches.Rectangle((x1, y1), x2 - x1, y2 - y1, linewidth=2, edgecolor='orange', facecolor='none')
        ax.add_patch(rect)
        ax.text(x1 + 4, max(0, y2 - 4), f"Pred: {class_names[label.item()]} ({score.item():.2f})",
  color='orange', fontsize=9, verticalalignment="bottom", bbox=dict(facecolor='black', alpha=0.7,
  edgecolor='none'))

    plt.axis('off')
    plt.title("Detections: Green = Ground Truth, Red = Predictions")
    plt.show()
```

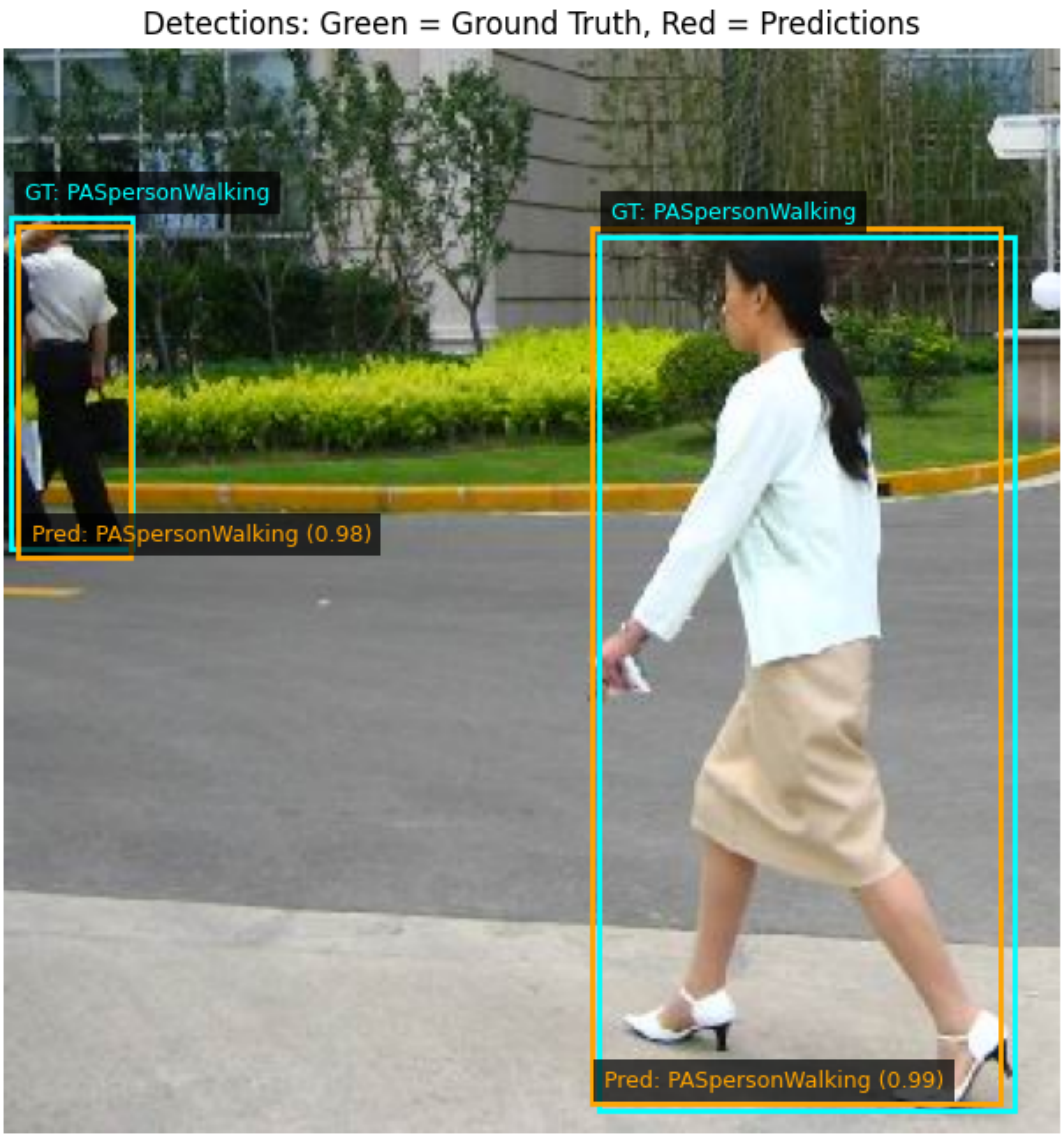
```
sample_idx = random.randint(0, len(dataset_test) - 1)
sample_image, sample_target = dataset_test[sample_idx]

denorm_sample_image = denormalize_image(sample_image, mean, std)
denorm_sample_image = denorm_sample_image.permute(1, 2, 0).cpu().numpy()
denorm_sample_image = (denorm_sample_image * 255).astype(np.uint8)

with torch.no_grad():
    prediction = model([sample_image.to(device)])
```

```
sample_pred = prediction[0]
sample_gt = sample_target

visualize_detections(denorm_sample_image, sample_pred, sample_gt, classes, threshold=0.9)
```



13.11. Homework (Optional)

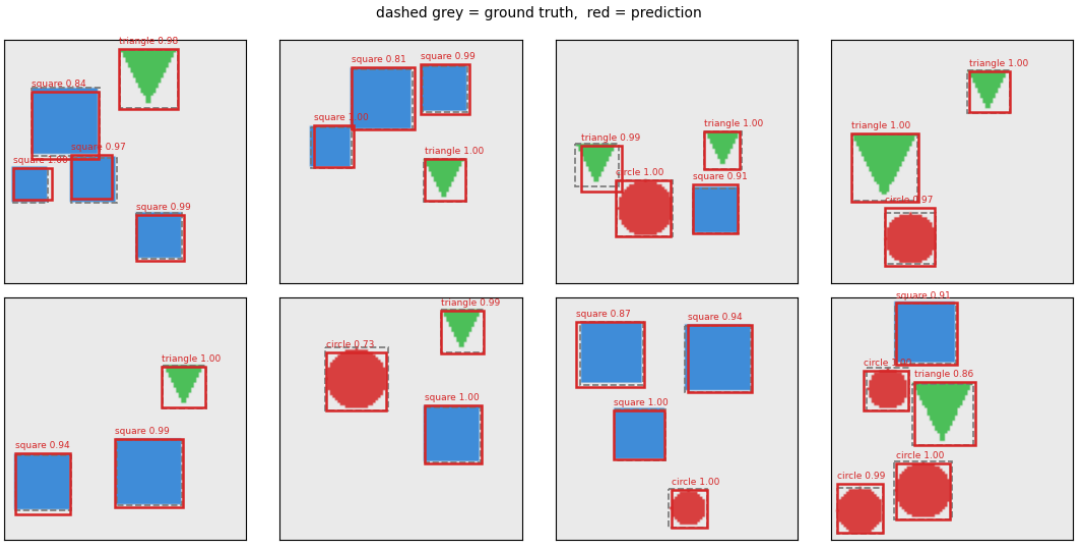
1. **Freeze vs. train the backbone.** Train two Faster R-CNN variants on Penn-Fudan: one with the ResNet-50 backbone frozen, one with it fine-tuned. Report mAP and training time for each and discuss the trade-off for this very small dataset.
2. **Anchor-box sensitivity.** The default RPN uses anchor scales and aspect ratios tuned for natural images. Experiment with narrower or wider aspect ratios (appropriate for pedestrians, which are tall and thin) and report whether mAP improves.

3. **Swap the backbone.** Replace the ResNet-50-FPN backbone with `torchvision.models.detection.backbone_utils.resnet_fpn_backbone('resnet18', weights=...)`. Compare parameter count, training time, and final mAP.
4. **IoU threshold analysis.** Evaluate the trained model's mAP at several IoU thresholds: `0.3, 0.5, 0.75, 0.9`. Plot mAP vs. IoU threshold and comment on how strictly the model localises boxes.
5. **Per-image inference time.** Measure the time taken for a single-image forward pass on GPU and on CPU. Contrast this with the equivalent YOLO11 [17] -nano inference time reported in the previous chapter.
6. **False-positive investigation.** Identify three high-confidence false-positive predictions on the test set. Display the images, the boxes, and the ground truth. Propose one change to the training pipeline that would likely reduce each type of error.

Chapter 14

Object Detection from Scratch

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/10sZXlJvRoagwiwSFH3zcMpGqaV12vucF?usp=sharing



Object detection is the task of both **localising** and **classifying** every object of interest in an image. Unlike classification, which predicts a single label per image, a detector must output a variable-length list of boxes together with their class labels and confidence scores.

This chapter builds a single-stage object detector from first principles, on a synthetic multi-object dataset. Each grid cell predicts several *anchor* boxes, which is what lets the same cell handle multiple objects of the same class. Non-Maximum Suppression cleans up overlapping predictions at inference time. The goal is not state-of-the-art accuracy but a transparent working understanding of each piece so that heavier frameworks (Faster R-CNN [16] , YOLO, DETR) read as natural extensions.

14.1. Necessary Imports

```
import random
import math
from typing import List, Tuple

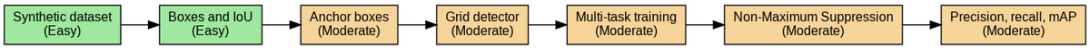
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
```

```
import matplotlib.pyplot as plt
import matplotlib.patches as patches

torch.manual_seed(0)
np.random.seed(0)
random.seed(0)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Using device:', device)
```

```
Using device: cuda
```



14.2. A synthetic multi-object dataset

Training data is generated on the fly: each image is a light background holding three to five coloured shapes (circle, square, triangle) at random non-overlapping positions. Shape classes are allowed to repeat within a single image, so the network must learn to produce multiple predictions of the same class side by side, not rely on class uniqueness. Bounding boxes are stored in the standard `(x, y, w, h)` normalised format where `(x, y)` is the box centre and the width and height are fractions of the image dimensions.

The synthetic setup keeps the chapter reproducible (no downloads) and lets us control the difficulty: more objects per image, smaller sizes, or overlapping placements can all be flipped on with one constant.

```
IMG_SIZE = 96
NUM_CLASSES = 3   # 0: circle, 1: square, 2: triangle
CLASS_NAMES = ['circle', 'square', 'triangle']
CLASS_COLORS = [(0.85, 0.25, 0.25), (0.25, 0.55, 0.85), (0.30, 0.75, 0.35)]

def _draw_shape(img, cls, cx, cy, size, color):
    half = size // 2
    y0, y1 = max(0, cy - half), min(IMG_SIZE, cy + half)
    x0, x1 = max(0, cx - half), min(IMG_SIZE, cx + half)
    if cls == 0:  # circle
        yy, xx = np.ogrid[y0:y1, x0:x1]
        mask = (yy - cy) ** 2 + (xx - cx) ** 2 <= half ** 2
        img[y0:y1, x0:x1][mask] = color
    elif cls == 1:  # square
        img[y0:y1, x0:x1] = color
    else:  # triangle (downward-pointing rasterisation)
        for yi in range(y0, y1):
            t = (yi - y0) / max(1, y1 - y0)
            row_half = int(half * (1 - t))
            img[yi, max(0, cx - row_half):min(IMG_SIZE, cx + row_half)] = color

def _px_overlap(a, b):
    return not (a[2] <= b[0] or b[2] <= a[0] or a[3] <= b[1] or b[3] <= a[1])

def generate_image(rng: np.random.Generator = None
                   ) -> Tuple[np.ndarray, List[Tuple[int, float, float, float, float]]]:
    '''Return one (image, boxes) pair with 3-5 non-overlapping shapes.

    Boxes are (class_id, cx, cy, w, h), all in normalised coordinates.
    '''
    rng = rng or np.random.default_rng()
```

```
    num_objects = int(rng.integers(3, 6))

    img = np.full((IMG_SIZE, IMG_SIZE, 3), 0.92, dtype=np.float32)
    boxes, placed = [], []

    attempts = 0
    while len(boxes) < num_objects and attempts < 80:
        attempts += 1
        cls = int(rng.integers(0, NUM_CLASSES))
        size = int(rng.integers(14, 28))
        cx = int(rng.integers(size // 2 + 2, IMG_SIZE - size // 2 - 2))
        cy = int(rng.integers(size // 2 + 2, IMG_SIZE - size // 2 - 2))
        new_box = (cx - size // 2, cy - size // 2, cx + size // 2, cy + size // 2)
        if any(_px_overlap(new_box, p) for p in placed):
            continue
        placed.append(new_box)
        _draw_shape(img, cls, cx, cy, size, CLASS_COLORS[cls])
        boxes.append((cls, cx / IMG_SIZE, cy / IMG_SIZE, size / IMG_SIZE, size / IMG_SIZE))

    img = (img * 255).astype(np.uint8)
    return img, boxes
```

```
class ShapesDetectionDataset(Dataset):
    '''Deterministic synthetic dataset wrapping the image generator.'''

    def __init__(self, n_samples: int = 1500, seed: int = 0):
        self.n_samples = n_samples
        self.seeds = np.random.default_rng(seed).integers(0, 2**31, size=n_samples)

    def __len__(self):
        return self.n_samples

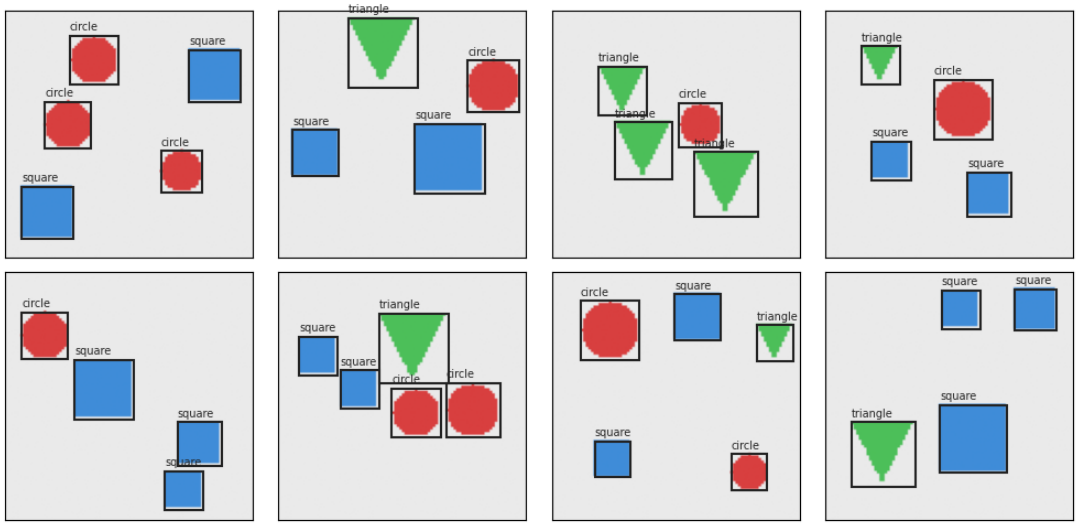
    def __getitem__(self, idx):
        rng = np.random.default_rng(int(self.seeds[idx]))
        img, boxes = generate_image(rng=rng)
        img_t = torch.from_numpy(img).permute(2, 0, 1).float() / 255.0
        boxes_t = torch.tensor(boxes, dtype=torch.float32) if boxes else torch.zeros(0, 5)
        return img_t, boxes_t

def collate_variable_length(batch):
    '''Stack images, keep per-image box lists.'''
    imgs = torch.stack([item[0] for item in batch], dim=0)
    boxes = [item[1] for item in batch]
    return imgs, boxes
```

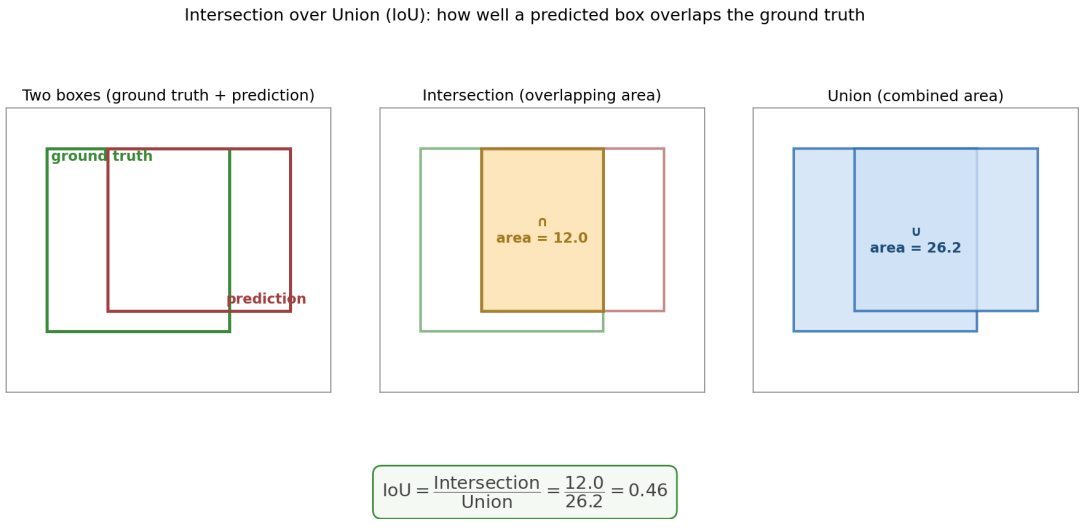
```
def draw_boxes(ax, img_np, boxes, color='#222', linewidth=1.5):
    '''Overlay (N, 5) boxes in (class, cx, cy, w, h) onto ax.'''
    ax.imshow(img_np)
    H, W = img_np.shape[:2]
    for b in boxes:
        cls = int(b[0].item())
        cx, cy, bw, bh = [v.item() for v in b[1:]]
        x = (cx - bw / 2) * W
        y = (cy - bh / 2) * H
        rect = patches.Rectangle((x, y), bw * W, bh * H,
                                 fill=False, edgecolor=color, linewidth=linewidth)
        ax.add_patch(rect)
        ax.text(x, max(0, y - 2), CLASS_NAMES[cls], fontsize=7, color=color)
    ax.set_xticks([]); ax.set_yticks([])
```

```
train_set = ShapesDetectionDataset(n_samples=1500, seed=42)
val_set = ShapesDetectionDataset(n_samples=300, seed=7)

fig, axes = plt.subplots(2, 4, figsize=(10, 5))
for ax, idx in zip(axes.flat, range(8)):
    img, boxes = train_set[idx]
    draw_boxes(ax, img.permute(1, 2, 0).numpy(), boxes)
plt.tight_layout()
plt.show()
```



14.2.1. Intersection over Union, visually



Intersection over Union is the workhorse metric for object detection: we use it to match predicted boxes to ground-truth boxes during training, to suppress duplicates at inference (NMS), and to compute precision/recall at evaluation. Values range from 0 (disjoint) to 1 (identical), with 0.5 commonly used as the matching threshold.

14.3. Bounding box representations and Intersection over Union

Three box conventions dominate detection code: `(cx, cy, w, h)` (centre and size, natural for regression), `(x1, y1, x2, y2)` (corners, natural for overlap computation and drawing), and `(x, y, w, h)` (top-left corner and size, the COCO [8] annotation format). The conversions between them are short and each code base tends to pick one representation and stick to it.

Intersection over Union (IoU) is the universal overlap measure used in detection: the area of the intersection divided by the area of the union of two boxes. It is zero for disjoint boxes and one for identical boxes. Every detection loss and post-processing step uses IoU at some point.

```
def cxcywh_to_xyxy(box: torch.Tensor) -> torch.Tensor:
    '''(..., 4) in (cx, cy, w, h) -> (..., 4) in (x1, y1, x2, y2).'''
    cx, cy, w, h = box.unbind(-1)
    return torch.stack([cx - w / 2, cy - h / 2, cx + w / 2, cy + h / 2], dim=-1)

def xyxy_to_cxcywh(box: torch.Tensor) -> torch.Tensor:
    x1, y1, x2, y2 = box.unbind(-1)
    return torch.stack([(x1 + x2) / 2, (y1 + y2) / 2, x2 - x1, y2 - y1], dim=-1)

def pairwise_iou(boxes_a: torch.Tensor, boxes_b: torch.Tensor) -> torch.Tensor:
    '''(M, N) IoU matrix for two sets of xyxy boxes.'''
    area_a = (boxes_a[:, 2] - boxes_a[:, 0]).clamp(min=0) * \
             (boxes_a[:, 3] - boxes_a[:, 1]).clamp(min=0)
    area_b = (boxes_b[:, 2] - boxes_b[:, 0]).clamp(min=0) * \
             (boxes_b[:, 3] - boxes_b[:, 1]).clamp(min=0)

    inter_lt = torch.max(boxes_a[:, None, :2], boxes_b[None, :, :2])
    inter_rb = torch.min(boxes_a[:, None, 2:], boxes_b[None, :, 2:])
    inter_wh = (inter_rb - inter_lt).clamp(min=0)
    inter_area = inter_wh[..., 0] * inter_wh[..., 1]

    union = area_a[:, None] + area_b[None, :] - inter_area
    return inter_area / union.clamp(min=1e-9)
```

```
# Sanity check: identical boxes give IoU 1, disjoint boxes give 0.
a = torch.tensor([[0.0, 0.0, 1.0, 1.0]])
b = torch.tensor([[0.0, 0.0, 1.0, 1.0],
                  [0.5, 0.5, 1.5, 1.5],
                  [2.0, 2.0, 3.0, 3.0]])
print('IoU matrix:')
print(pairwise_iou(a, b))
```

```
IoU matrix:
tensor([[1.0000, 0.1429, 0.0000]])
```

14.4. Anchor boxes

A grid cell with a single prediction slot cannot represent two objects of the same class in the same region: the target assignment would overwrite one with the other. The standard fix is *anchor boxes*. Each cell is given several predefined box shapes, and each anchor independently predicts its own refinement of that shape. Two objects can share a cell as long as they match different anchors.

For our simple shapes dataset, three square anchors at scales 0.15, 0.22 and 0.30 are enough to cover the range of shape sizes in the generator. Real detectors use anchors of varying aspect ratio too, but here the shapes are roughly square so the priors are square as well.

During training, each ground-truth box in a given cell is matched to the anchor whose *prior size* has the highest IoU with the box size. That anchor is then responsible for predicting the box.

```
# Anchor boxes are specified as normalised (w, h) priors.
ANCHORS = torch.tensor([
    [0.15, 0.15],
    [0.22, 0.22],
    [0.30, 0.30],
], dtype=torch.float32)
NUM_ANCHORS = ANCHORS.shape[0]

def anchor_iou_1d(box_wh: torch.Tensor, anchors_wh: torch.Tensor) -> torch.Tensor:
    '''IoU between a single (w, h) and (K, 2) anchors, assuming same centre.

    This simplification is standard for anchor matching: we only care which
    anchor shape fits the target best, independent of its spatial position.
    '''
    min_w = torch.minimum(box_wh[0], anchors_wh[:, 0])
    min_h = torch.minimum(box_wh[1], anchors_wh[:, 1])
    inter = min_w * min_h
    union = box_wh[0] * box_wh[1] + anchors_wh[:, 0] * anchors_wh[:, 1] - inter
    return inter / union.clamp(min=1e-9)

# Pick the best anchor for a few example sizes
for w, h in [(0.13, 0.13), (0.20, 0.20), (0.28, 0.28)]:
    ious = anchor_iou_1d(torch.tensor([w, h]), ANCHORS)
    print(f'box ({w:.2f},{h:.2f}) -> best anchor idx {ious.argmax().item()} '
          f'(IoU={ious.max().item():.2f})')
```

```
box (0.13,0.13) -> best anchor idx 0 (IoU=0.75)
box (0.20,0.20) -> best anchor idx 1 (IoU=0.83)
box (0.28,0.28) -> best anchor idx 2 (IoU=0.87)
```

14.5. Model architecture

A small CNN backbone downsamples the input to an `S x S` feature map. At each of the `S * S` spatial positions the detection head produces `B = NUM_ANCHORS` independent predictions, each consisting of four bounding-box values, one objectness score and `NUM_CLASSES` class logits, `B * (5 + NUM_CLASSES)` output channels per cell in total.

Using `S = 6` cells across a 96-pixel image means each cell covers a 16×16 patch, comfortably larger than the shapes. `B = 3` anchors means the grid can represent up to `S * S * B = 108` simultaneous objects, far more than the 3-5 actually present, which leaves plenty of room for the network to be confident only in the right places.

```
S = 6   # grid size

class GridDetector(nn.Module):
    '''CNN backbone + anchor-based grid detection head.'''

    def __init__(self, num_classes: int = NUM_CLASSES,
```

```
                 num_anchors: int = NUM_ANCHORS, grid: int = S):
        super().__init__()
        self.grid = grid
        self.num_classes = num_classes
        self.num_anchors = num_anchors

        def conv_block(in_c, out_c):
            return nn.Sequential(
                nn.Conv2d(in_c, out_c, 3, padding=1),
                nn.BatchNorm2d(out_c),
                nn.ReLU(inplace=True),
                nn.MaxPool2d(2),
            )

        # 96 -> 48 -> 24 -> 12 -> 6
        self.backbone = nn.Sequential(
            conv_block(3, 32),
            conv_block(32, 64),
            conv_block(64, 128),
            conv_block(128, 256),
        )

        # Per-cell prediction: B * (objectness + 4 bbox + num_classes)
        pred_depth = num_anchors * (5 + num_classes)
        self.head = nn.Conv2d(256, pred_depth, kernel_size=1)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        feat = self.backbone(x)                  # (B, 256, S, S)
        pred = self.head(feat)                   # (B, A*(5+C), S, S)
        # Reshape to (B, S, S, A, 5 + C) for clarity downstream
        B, _, H, W = pred.shape
        pred = pred.view(B, self.num_anchors, 5 + self.num_classes, H, W)
        return pred.permute(0, 3, 4, 1, 2).contiguous()   # (B, S, S, A, 5+C)

# Sanity check the shapes before training
_m = GridDetector().to(device)
with torch.no_grad():
    out = _m(torch.zeros(1, 3, IMG_SIZE, IMG_SIZE, device=device))
print('Output shape:', tuple(out.shape),
      '= (batch, S, S, anchors, 5 + classes)')
```

```
Output shape: (1, 6, 6, 3, 8) = (batch, S, S, anchors, 5 + classes)
```

14.6. Target assignment

For each ground-truth box we locate the grid cell containing its centre, then match the box to the anchor whose prior size has the highest IoU with the box's `(w, h)`. That `(cell, anchor)` slot is marked responsible. Its objectness target becomes 1, its class target becomes the box class, and its bounding-box target is the GT box. All other slots have objectness 0 and do not contribute to the class or box losses.

The bounding-box regression target is the box itself, re-expressed as cell-relative centre offsets (`tx`, `ty` in `[0, 1]`) and log-width / log-height relative to the matched anchor (`log(w_gt / w_anchor)`). Offsets are kept in the model output space via a sigmoid (for the centre offsets) and a plain exponential (for the size). This is exactly the YOLOv2/v3 parameterisation.

```python
def build_targets(boxes_list: List[torch.Tensor], grid: int = S,
                  anchors: torch.Tensor = ANCHORS
                  ) -> Tuple[torch.Tensor, torch.Tensor]:
    '''Construct (B, S, S, A, 5 + C) and (B, S, S, A) mask tensors.

    target channels: [tx, ty, tw, th, class_id]  for responsible anchors only
    has_obj mask    : 1 where an anchor is responsible, 0 otherwise
    '''
    B = len(boxes_list)
    A = anchors.shape[0]
    target = torch.zeros(B, grid, grid, A, 5)   # tx, ty, tw, th, class_id
    has_obj = torch.zeros(B, grid, grid, A)

    for b, boxes in enumerate(boxes_list):
        for cls, cx, cy, w, h in boxes:
            gx = int(min(grid - 1, int(cx * grid)))
            gy = int(min(grid - 1, int(cy * grid)))

            # Choose the anchor whose prior size best matches this box.
            ious = anchor_iou_1d(torch.tensor([w, h]), anchors)
            a_idx = int(ious.argmax().item())

            if has_obj[b, gy, gx, a_idx]:
                # Rare: two boxes want the same (cell, anchor). Skip the
                # second to keep the signal clean; exercise 4 of the
                # homework asks you to handle this case better.
                continue

            tx = cx * grid - gx                  # in [0, 1]
            ty = cy * grid - gy                  # in [0, 1]
            tw = math.log(max(w, 1e-6) / anchors[a_idx, 0].item())
            th = math.log(max(h, 1e-6) / anchors[a_idx, 1].item())

            target[b, gy, gx, a_idx] = torch.tensor([tx, ty, tw, th, cls])
            has_obj[b, gy, gx, a_idx] = 1.0

    return target, has_obj

# Show how many anchors are responsible for a typical batch
sample_boxes = [train_set[i][1] for i in range(8)]
_, mask = build_targets(sample_boxes)
print(f'Responsible anchors in batch of 8: {int(mask.sum().item())} '
      f'out of {mask.numel()} slots')
```

```
Responsible anchors in batch of 8: 35 out of 864 slots
```

14.7. Multi-task loss

The total loss sums three terms, each operating on the correct subset of slots.

1. *Objectness loss*, binary cross-entropy over every `(cell, anchor)`. The background term is down-weighted by `LAMBDA_NOOBJ` because there are far more empty slots than responsible ones.
2. *Classification loss*, cross-entropy, only on responsible slots.
3. *Box regression loss*, mean squared error on `(tx, ty, tw, th)`, only on responsible slots.

The `LAMBDA_COORD` and `LAMBDA_NOOBJ` weights come from the YOLOv1/v2 papers and keep the three losses on comparable scales during training.

```
LAMBDA_COORD = 5.0
LAMBDA_NOOBJ = 0.5

def detector_loss(preds: torch.Tensor, target: torch.Tensor, has_obj: torch.Tensor
                  ) -> Tuple[torch.Tensor, dict]:
    '''Return the scalar loss plus a dict of per-component values.

    preds   : (B, S, S, A, 5 + C) raw logits
    target  : (B, S, S, A, 5)  -- tx, ty, tw, th, class (only valid where has_obj)
    has_obj : (B, S, S, A)
    '''
    B, Gy, Gx, A, depth = preds.shape
    C = depth - 5

    bbox_pred = preds[..., :4]                  # (..., 4)
    obj_logit = preds[..., 4]                   # (...)
    class_logits = preds[..., 5:]               # (..., C)

    # Objectness, BCE, with noobj down-weighted
    obj_bce = F.binary_cross_entropy_with_logits(
        obj_logit, has_obj, reduction='none')
    obj_loss = (has_obj * obj_bce).sum() + LAMBDA_NOOBJ * ((1 - has_obj) * obj_bce).sum()

    pos_mask = has_obj.bool()
    if pos_mask.any():
        # Classification, cross-entropy on responsible anchors only
        cls_logits_flat = class_logits[pos_mask]            # (N_pos, C)
        cls_target_flat = target[pos_mask][:, 4].long()     # (N_pos,)
        cls_loss = F.cross_entropy(cls_logits_flat, cls_target_flat, reduction='sum')

        # Box regression: sigmoid on tx/ty (centre in cell), raw for tw/th (log ratio)
        pred_txy = torch.sigmoid(bbox_pred[..., :2])
        pred_twh = bbox_pred[..., 2:]
        tgt_txy = target[..., :2]
        tgt_twh = target[..., 2:4]
        box_loss = ((pred_txy - tgt_txy) ** 2).sum(dim=-1) + \
                   ((pred_twh - tgt_twh) ** 2).sum(dim=-1)
        box_loss = box_loss[pos_mask].sum()
    else:
        cls_loss = torch.zeros((), device=preds.device)
        box_loss = torch.zeros((), device=preds.device)

    total = (obj_loss + cls_loss + LAMBDA_COORD * box_loss) / B
    return total, {
        'obj': obj_loss.item() / B,
        'cls': cls_loss.item() / B,
        'box': box_loss.item() / B,
    }
```

14.8. Training

Thirty epochs on 1500 synthetic images with a cosine learning-rate schedule. The batch size is 32 and the optimiser is Adam with an initial learning rate of 2e-3. Training is fast: each epoch takes a fraction of a second on a T4 and should converge within the first 10-15 epochs. The remaining epochs polish localisation quality.

```
EPOCHS = 30
BATCH_SIZE = 32

model = GridDetector().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=2e-3, weight_decay=1e-4)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=EPOCHS)

train_loader = DataLoader(train_set, batch_size=BATCH_SIZE, shuffle=True,
                          collate_fn=collate_variable_length, num_workers=0)
val_loader   = DataLoader(val_set, batch_size=BATCH_SIZE, shuffle=False,
                          collate_fn=collate_variable_length, num_workers=0)

def run_epoch(loader, training: bool):
    model.train(training)
    tot = {'loss': 0.0, 'obj': 0.0, 'cls': 0.0, 'box': 0.0}
    n = 0
    for imgs, boxes in loader:
        imgs = imgs.to(device)
        target, has_obj = build_targets(boxes)
        target = target.to(device); has_obj = has_obj.to(device)

        preds = model(imgs)
        loss, parts = detector_loss(preds, target, has_obj)

        if training:
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        bs = imgs.size(0)
        tot['loss'] += loss.item() * bs
        for k, v in parts.items():
            tot[k] += v * bs
        n += bs

    return {k: v / n for k, v in tot.items()}

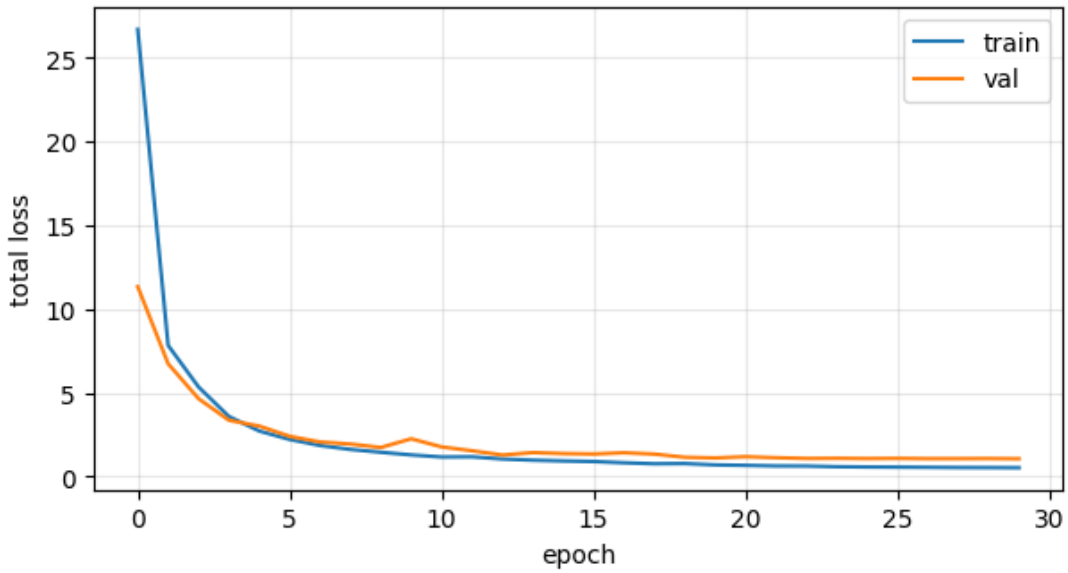
history = {'train': [], 'val': []}
for epoch in range(EPOCHS):
    tr = run_epoch(train_loader, training=True)
    with torch.no_grad():
        vl = run_epoch(val_loader, training=False)
    scheduler.step()
    history['train'].append(tr)
    history['val'].append(vl)
    if (epoch + 1) % 5 == 0 or epoch == 0:
        print(f'epoch {epoch+1:2d}/{EPOCHS}  '
              f'train={tr["loss"]:.3f}  val={vl["loss"]:.3f}  '
              f'(obj {tr["obj"]:.3f}  cls {tr["cls"]:.3f}  box {tr["box"]:.3f})')
```

```
epoch  1/30  train=26.696  val=11.336  (obj 11.454  cls 1.791  box 2.690)
epoch  5/30  train=2.712  val=3.008  (obj 2.209  cls 0.029  box 0.095)
epoch 10/30  train=1.285  val=2.250  (obj 1.022  cls 0.009  box 0.051)
epoch 15/30  train=0.927  val=1.368  (obj 0.748  cls 0.005  box 0.035)
epoch 20/30  train=0.691  val=1.106  (obj 0.587  cls 0.003  box 0.020)
```

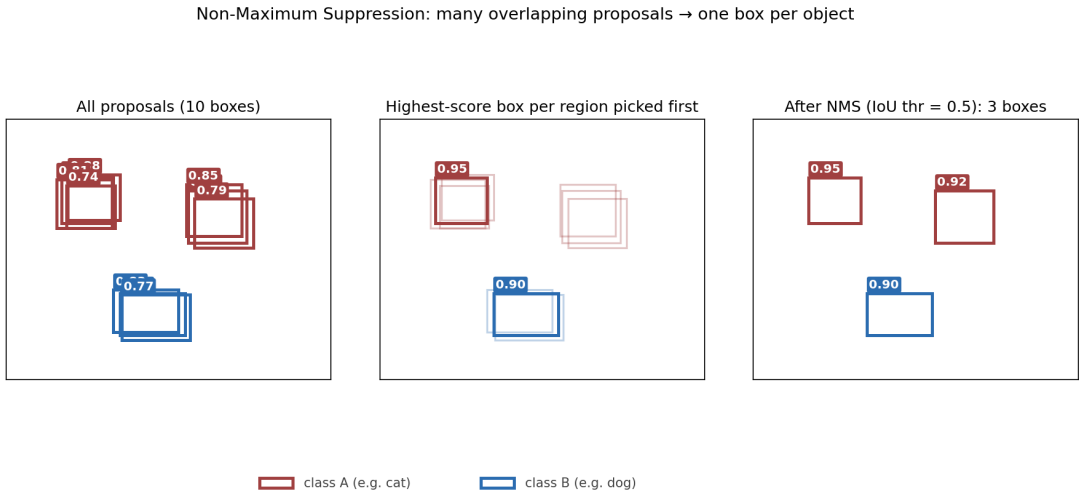
```
epoch 25/30  train=0.565  val=1.070  (obj 0.493  cls 0.003  box 0.014)
```

```
epoch 30/30  train=0.520  val=1.057  (obj 0.458  cls 0.003  box 0.012)
```

```
fig, ax = plt.subplots(figsize=(7, 3.5))
ax.plot([h['loss'] for h in history['train']], label='train')
ax.plot([h['loss'] for h in history['val']],   label='val')
ax.set_xlabel('epoch'); ax.set_ylabel('total loss'); ax.legend(); ax.grid(alpha=0.3)
plt.show()
```



14.8.1. NMS at a glance



An object detector typically returns many overlapping proposals around each true object. **Non-Maximum Suppression** keeps the highest-scoring box and discards any other box that has IoU above a threshold (typically 0.5) with one of the kept boxes, repeating per class. The result is roughly one box per object.

14.9. Non-Maximum Suppression

Anchor-based decoding yields many overlapping predictions for the same object. Non-Maximum Suppression keeps only the strongest prediction from each cluster of overlapping boxes: sort by confidence, keep the highest, discard everything with IoU above a threshold, repeat. NMS is greedy. Variants like Soft-NMS and DIoU-NMS exist but plain NMS remains the default in most detectors and is what we implement here.

```
def nms_from_scratch(boxes_xyxy: torch.Tensor,
                     scores: torch.Tensor,
                     iou_threshold: float = 0.5) -> torch.Tensor:
    '''Return indices of kept boxes (descending score order).'''
    order = scores.argsort(descending=True)
    kept = []
    while order.numel() > 0:
        top = order[0]
        kept.append(top.item())
        if order.numel() == 1:
            break
        rest = order[1:]
        ious = pairwise_iou(boxes_xyxy[top].unsqueeze(0),
                            boxes_xyxy[rest]).squeeze(0)
        order = rest[ious < iou_threshold]
    return torch.tensor(kept, dtype=torch.long)
```

```
# Verify our NMS against torchvision's official implementation.
from torchvision.ops import nms as torchvision_nms

gen = torch.Generator().manual_seed(0)
boxes = torch.rand(20, 4, generator=gen)
boxes = torch.stack([
    torch.minimum(boxes[:, 0], boxes[:, 2]),
    torch.minimum(boxes[:, 1], boxes[:, 3]),
    torch.maximum(boxes[:, 0], boxes[:, 2]),
    torch.maximum(boxes[:, 1], boxes[:, 3]),
], dim=1)
scores = torch.rand(20, generator=gen)

ours = nms_from_scratch(boxes, scores, iou_threshold=0.5)
theirs = torchvision_nms(boxes, scores, iou_threshold=0.5)
print('Our kept indices   :', sorted(ours.tolist()))
print('torchvision.ops.nms:', sorted(theirs.tolist()))
print('Match:', sorted(ours.tolist()) == sorted(theirs.tolist()))
```

```
Our kept indices   : [0, 1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18,
19]
torchvision.ops.nms: [0, 1, 2, 3, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18,
19]
Match: True
```

14.10. Inference pipeline

Decoding raw predictions takes three steps per image: (1) undo the sigmoid and exponential parameterisation to obtain normalised `(cx, cy, w, h)` predictions for every `(cell, anchor)` slot, (2) keep only slots whose combined `objectness * class_score` is above a confidence threshold, (3) run per-class

NMS to remove overlapping duplicates. The output is a clean list of `(class, score, x1, y1, x2, y2)` tuples suitable for evaluation or visualisation.

```
@torch.no_grad()
def decode_predictions(preds: torch.Tensor,
                       conf_threshold: float = 0.3,
                       iou_threshold: float = 0.4,
                       anchors: torch.Tensor = ANCHORS):
    '''Return a list (len = batch) of lists of (cls, score, x1, y1, x2, y2).'''
    B, Gy, Gx, A, depth = preds.shape
    C = depth - 5
    anchors_dev = anchors.to(preds.device)

    cell_y, cell_x = torch.meshgrid(
        torch.arange(Gy, device=preds.device),
        torch.arange(Gx, device=preds.device),
        indexing='ij',
    )
    # broadcast to (1, S, S, 1)
    cell_x = cell_x.unsqueeze(0).unsqueeze(-1)
    cell_y = cell_y.unsqueeze(0).unsqueeze(-1)

    tx = torch.sigmoid(preds[..., 0])
    ty = torch.sigmoid(preds[..., 1])
    tw = preds[..., 2]
    th = preds[..., 3]
    obj = torch.sigmoid(preds[..., 4])
    class_probs = F.softmax(preds[..., 5:], dim=-1)
    top_class_score, top_class = class_probs.max(dim=-1)
    confidence = obj * top_class_score          # (B, S, S, A)

    cx = (tx + cell_x) / Gx
    cy = (ty + cell_y) / Gy
    w = torch.exp(tw) * anchors_dev[:, 0]       # (B, S, S, A) via broadcasting
    h = torch.exp(th) * anchors_dev[:, 1]

    per_image = []
    for b in range(B):
        keep = confidence[b] > conf_threshold
        if keep.sum() == 0:
            per_image.append([])
            continue

        cxb = cx[b][keep]; cyb = cy[b][keep]
        wb = w[b][keep];   hb = h[b][keep]
        sb = confidence[b][keep]
        clsb = top_class[b][keep]

        boxes_xyxy = torch.stack([cxb - wb/2, cyb - hb/2,
                                  cxb + wb/2, cyb + hb/2], dim=-1)

        dets = []
        for cls_id in range(C):
            mask = clsb == cls_id
            if mask.sum() == 0:
                continue
            kept = nms_from_scratch(boxes_xyxy[mask], sb[mask], iou_threshold)
            for i in kept.tolist():
                bx = boxes_xyxy[mask][i].tolist()
                dets.append((cls_id, sb[mask][i].item(), *bx))
```

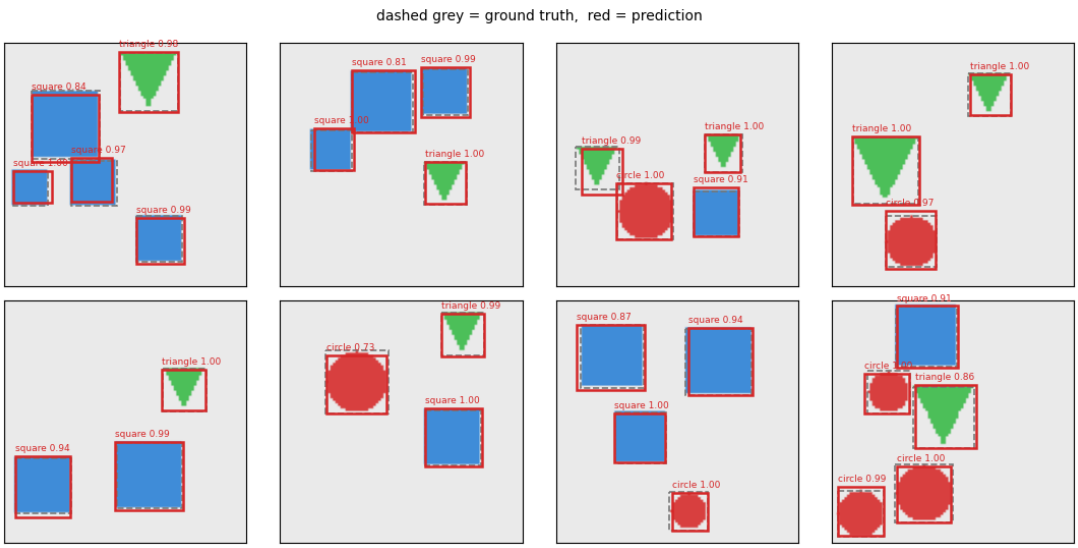
```
        per_image.append(dets)
    return per_image
```

```
def show_detections(ax, img_np, gt_boxes, predictions):
    ax.imshow(img_np)
    H, W = img_np.shape[:2]
    for b in gt_boxes:
        cls = int(b[0].item())
        cx, cy, bw, bh = [v.item() for v in b[1:]]
        x = (cx - bw / 2) * W
        y = (cy - bh / 2) * H
        ax.add_patch(patches.Rectangle((x, y), bw * W, bh * H,
                                       fill=False, edgecolor='#777',
                                       linewidth=1.2, linestyle='--'))

    for cls, score, x1, y1, x2, y2 in predictions:
        ax.add_patch(patches.Rectangle((x1 * W, y1 * H),
                                       (x2 - x1) * W, (y2 - y1) * H,
                                       fill=False, edgecolor='#d62728',
                                       linewidth=1.8))
        ax.text(x1 * W, max(0, y1 * H - 2),
                f'{CLASS_NAMES[cls]} {score:.2f}',
                fontsize=6.5, color='#d62728')
    ax.set_xticks([]); ax.set_yticks([])

model.eval()
sample_imgs, sample_boxes = next(iter(val_loader))
preds = model(sample_imgs[:8].to(device))
detections = decode_predictions(preds)

fig, axes = plt.subplots(2, 4, figsize=(11, 5.5))
for i, ax in enumerate(axes.flat):
    img = (sample_imgs[i].permute(1, 2, 0).numpy() * 255).astype(np.uint8)
    show_detections(ax, img, sample_boxes[i], detections[i])
fig.suptitle('dashed grey = ground truth,  red = prediction', fontsize=10)
plt.tight_layout()
plt.show()
```



14.11. Precision, recall, and mean Average Precision

Detection evaluation requires both class correctness and spatial correctness. A prediction counts as a true positive only if its class matches a ground-truth box's class and its IoU with that box exceeds a threshold, typically 0.5. All other predictions are false positives. Unmatched ground-truth boxes are false negatives.

Sweeping the confidence threshold traces out a precision-recall curve per class. Average Precision (AP) is the area under that curve. mAP is the mean over classes. The VOC 2007 convention uses 11-point interpolation, which we replicate below.

```
def compute_ap(predictions_by_image, gts_by_image, num_classes,
               iou_threshold: float = 0.5):
    '''Per-class AP and overall mAP, VOC 11-point style.'''
    aps = {}
    for c in range(num_classes):
        records = []
        n_gt = 0
        for preds, gts in zip(predictions_by_image, gts_by_image):
            gt_c = [b for b in gts if int(b[0].item()) == c]
            n_gt += len(gt_c)

            gt_xyxy = [cxcywh_to_xyxy(b[1:].clone().detach()) for b in gt_c]
            gt_matched = [False] * len(gt_c)

            pred_c = sorted([p for p in preds if p[0] == c],
                            key=lambda p: p[1], reverse=True)

            for _, score, x1, y1, x2, y2 in pred_c:
                pred_xyxy = torch.tensor([x1, y1, x2, y2])
                best_iou, best_j = 0.0, -1
                for j, gt in enumerate(gt_xyxy):
                    if gt_matched[j]:
                        continue
                    iou = pairwise_iou(pred_xyxy.unsqueeze(0), gt.unsqueeze(0)).item()
                    if iou > best_iou:
                        best_iou, best_j = iou, j
                if best_iou >= iou_threshold and best_j != -1:
                    gt_matched[best_j] = True
                    records.append((score, 1))
                else:
                    records.append((score, 0))

        records.sort(key=lambda r: r[0], reverse=True)
        tp = fp = 0
        precisions, recalls = [], []
        for _, is_tp in records:
            tp += is_tp
            fp += 1 - is_tp
            precisions.append(tp / (tp + fp))
            recalls.append(tp / max(1, n_gt))

        ap = 0.0
        for t in np.linspace(0, 1, 11):
            p_at_t = [p for p, r in zip(precisions, recalls) if r >= t]
            ap += max(p_at_t) if p_at_t else 0.0
        aps[c] = ap / 11
```

```
    aps['mAP'] = sum(aps[c] for c in range(num_classes)) / num_classes
    return aps
```

```
model.eval()
all_preds, all_gts = [], []
with torch.no_grad():
    for imgs, boxes in val_loader:
        preds = model(imgs.to(device))
        all_preds.extend(decode_predictions(preds))
        all_gts.extend(boxes)

aps = compute_ap(all_preds, all_gts, NUM_CLASSES, iou_threshold=0.5)
for c in range(NUM_CLASSES):
    print(f'AP@0.5 for {CLASS_NAMES[c]:<10s}  {aps[c]:.3f}')
print(f'mAP@0.5                {aps["mAP"]:.3f}')
```

```
AP@0.5 for circle      0.908
AP@0.5 for square      0.909
AP@0.5 for triangle    0.909
mAP@0.5                0.909
```

14.12. Homework (Optional)

1. **Rectangular anchors.** Extend the anchor set to three square plus two rectangular priors (one tall, one wide). Regenerate the dataset so rectangular boxes occur naturally. Retrain and compare per-class AP.
2. **GIoU loss.** Replace the mean-squared-error bounding-box loss with Generalised IoU loss (Rezatofighi et al., 2019). Report mAP@0.5 before and after.
3. **Soft-NMS.** Re-implement NMS so overlapping boxes have their scores *decayed* by a Gaussian function of IoU instead of being discarded. Compare mAP against plain NMS.
4. **Multiple boxes per cell/anchor slot.** In `build_targets`, when two GT boxes collide on the same `(cell, anchor)`, the second one is dropped. Extend the matching so collisions fall back to the next-best anchor instead. Measure the gain on dense scenes.
5. **Grid-size study.** Train detectors with `S = 4, 6, 12` and plot mAP against grid size. Explain both the lower bound (too coarse) and the upper bound (too many predictions per object to learn well).
6. **Class imbalance.** Bias the generator so `circle` appears in 60 percent of images but `triangle` in only 10 percent. Report per-class AP. Which class suffers most and what interventions (focal loss, class weights, oversampling) help?
7. **Transfer to VOC.** Swap the synthetic dataset for a small subset of Pascal VOC [6] 2007 (say 5 classes). Adjust `IMG_SIZE`, anchors, and `NUM_CLASSES` accordingly. Report VOC mAP@0.5 after 20 epochs.

Chapter 15

Image Segmentation with U-Net

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1txAqAFxTRLSPhrw7u0w9Vnf1t8_ohCmr?usp=sharing

U-Net [15] is a convolutional neural network architecture designed specifically for semantic segmentation, featuring an encoder-decoder structure with skip connections that helps preserve spatial information while learning abstract features. Its ability to produce precise pixel-level predictions has made it particularly effective in applications such as biomedical image segmentation and other tasks requiring detailed image analysis.

15.1. Necessary Imports

```
import copy
import os
import torch
import time
import torchvision
import numpy as np
import torch.nn.functional as F
import matplotlib.pyplot as plt
import torch.optim as optim
import torch.utils.data as data
from IPython.display import Image as IMG
from torch import nn, Tensor
from torch.optim import lr_scheduler
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, datasets, models
from torchsummary import summary
from torch import nn, Tensor
from zipfile import ZipFile
from PIL import Image

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```



15.1.1. Loading the data

In this class, we will employ the U-Net architecture to perform semantic segmentation on **brain MRI images**, using the **LGG MRI Segmentation** dataset from Kaggle. Specifically, we'll train the network to accurately delineate abnormal regions in the FLAIR images by learning from manually created segmentation masks, thereby demonstrating how **U-Net** can be applied to complex medical imaging tasks.

```
%%bash
if [ -d "data" ]; then
  echo "Data folder already exists. Skipping download and extraction."
else
  echo "Data folder does not exist. Downloading and extracting..."
  gdown 1yp6qcFC8fzjQP6Ri3CJNDn53bDr0wkQ7
  unzip -q archive.zip -d data
  rm archive.zip
fi
```

```
Data folder does not exist. Downloading and extracting...
```

```
Downloading...
From (original):
https://drive.google.com/uc?id=1yp6qcFC8fzjQP6Ri3CJNDn53bDr0wkQ7
From (redirected): https://drive.google.com/uc?id=1yp6qcFC8fzjQP6Ri3CJNDn53bDr0w
kQ7&confirm=t&uuid=db5b13aa-6f96-4033-a44c-55f20df9ac63
To: /content/archive.zip

[... 80 progress-bar updates ...]
100%|##########| 747M/747M [00:16<00:00, 46.0MB/s]
```

15.2. Looking at Our Dataset

```
mri_paths = [
    'data/archive/train/lgg-mri-segmentation/kaggle_3m/TCGA_DU_A5TP_19970614/TCGA_DU_A5TP_19970614_27.tif',
    'data/archive/train/lgg-mri-segmentation/kaggle_3m/TCGA_DU_A5TP_19970614/TCGA_DU_A5TP_19970614_28.tif',
    'data/archive/train/lgg-mri-segmentation/kaggle_3m/TCGA_DU_A5TP_19970614/TCGA_DU_A5TP_19970614_29.tif'
]

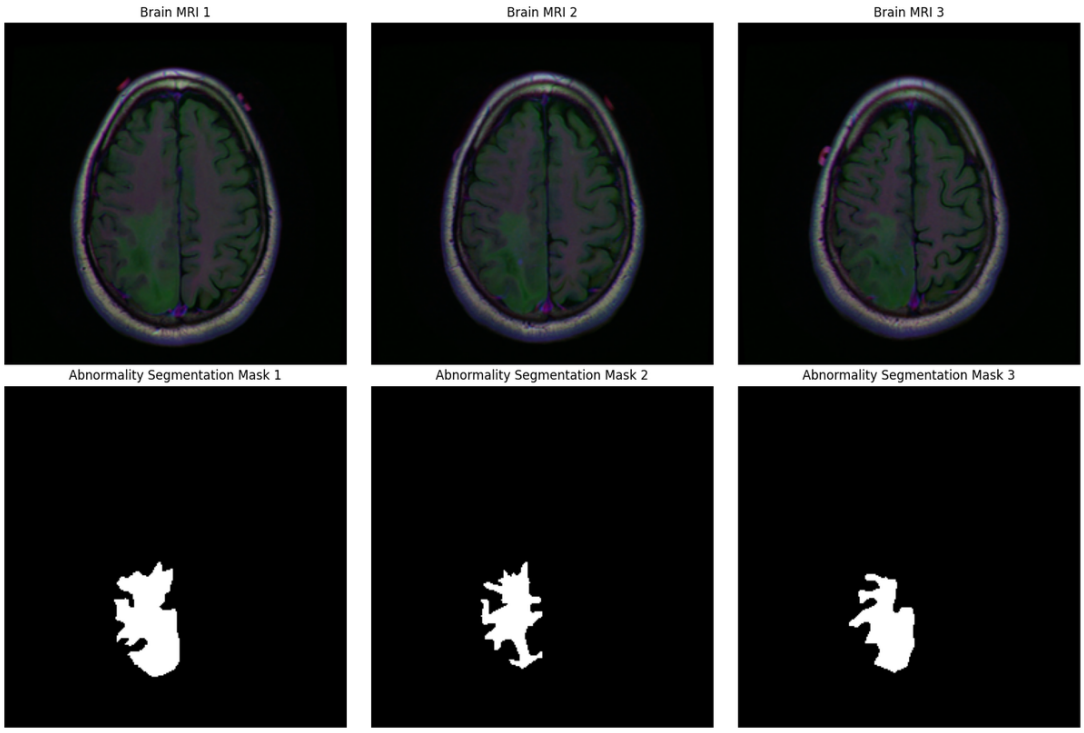
mask_paths = [
    'data/archive/train/lgg-mri-segmentation/kaggle_3m/TCGA_DU_A5TP_19970614/TCGA_DU_A5TP_19970614_27_mask.
   tif',
    'data/archive/train/lgg-mri-segmentation/kaggle_3m/TCGA_DU_A5TP_19970614/TCGA_DU_A5TP_19970614_28_mask.
   tif',
    'data/archive/train/lgg-mri-segmentation/kaggle_3m/TCGA_DU_A5TP_19970614/TCGA_DU_A5TP_19970614_29_mask.tif'
]

fig, axes = plt.subplots(2, 3, figsize=(15, 10))
for i in range(3):
    mri_img = Image.open(mri_paths[i])
    mask_img = Image.open(mask_paths[i])

    axes[0, i].imshow(mri_img, cmap='gray')
    axes[0, i].axis('off')
    axes[0, i].set_title(f'Brain MRI {i+1}')

    axes[1, i].imshow(mask_img, cmap='gray')
    axes[1, i].axis('off')
    axes[1, i].set_title(f'Abnormality Segmentation Mask {i+1}')

plt.tight_layout()
plt.show()
```



15.3. Loss functions for semantic segmentation

Pixel-level classification can be trained with the same Binary Cross-Entropy loss we would use for a standard classifier, applied per pixel. BCE converges quickly and is robust, but on datasets with heavy class imbalance (tumour pixels versus brain pixels, road pixels versus everything else) the network learns to predict the majority class and ignore the thin foreground.

Dice loss is the standard remedy. It is the complement of the Dice coefficient, `1 - 2|A ∩ B| / (|A| + |B|)`, computed over the predicted and ground-truth masks. Dice loss focuses the gradient on overlap with the foreground regardless of how many background pixels exist, so it performs much better on small-target segmentation.

In practice the two losses are **combined**: `L = L_BCE + L_Dice`. BCE provides stable per-pixel gradients that prevent degenerate solutions early in training, and Dice ensures the final masks are shape-correct once training stabilises. Most modern segmentation codebases default to this combination.

15.4. Preparing our Data for Training

We will create a `BrainSegmentationDataset` PyTorch dataset that scans our data directory and collects MRI images and their corresponding segmentation masks from the `train`, `val`, and `test` splits for later use. The dataset applies minimal transformations, such as resizing and normalization, to prepare the data for training and evaluation.

```
class BrainSegmentationDataset(Dataset):
    """
    A custom dataset for brain MRI segmentation.

    This dataset loads brain MRI images and their corresponding abnormality masks
    from a specified root directory. It expects that the mask files have filenames
    ending with '_mask.tif' and that the corresponding image files end with '.tif'.
    The dataset ensures that the images and masks are sorted so that they are aligned.

    Args:
        root_dir (str): Directory containing the images and masks.
        transforms (callable, optional): A function/transform that takes in an image and mask
                                         and returns transformed versions.
    """
    def __init__(self, root_dir, transforms=None):
        self.root_dir = root_dir
        self.transforms = transforms
        self.image_paths = []
        self.mask_paths = []

        # Traverse the directory tree and collect image and mask file paths.
        for root, _, files in os.walk(self.root_dir):
            for file in files:
                if file.endswith(".tif"):
                    full_path = os.path.join(root, file)
                    if file.endswith("_mask.tif"):
                        self.mask_paths.append(full_path)
                    else:
                        self.image_paths.append(full_path)

        # Sort the lists to ensure that images and masks are aligned.
        self.image_paths = sorted(self.image_paths)
        self.mask_paths = sorted(self.mask_paths)

        # Ensure that each image has a corresponding mask.
        assert len(self.image_paths) == len(self.mask_paths), (
            f"Number of images ({len(self.image_paths)}) does not match number of masks ({len(self.
  mask_paths)})."
        )

    def __len__(self):
        return len(self.image_paths)

    def __getitem__(self, idx):
        # Load the image and mask using PIL.
        img_path = self.image_paths[idx]
        mask_path = self.mask_paths[idx]
        image = Image.open(img_path).convert("RGB")  # Ensure 3-channel RGB for images.
        mask = Image.open(mask_path)  # You can convert mask mode if needed, e.g., mask.convert("L")

        # Apply transforms if provided.
        if self.transforms:
            image, mask = self.transforms(image, mask)

        return image, mask
```

We create a `DualCompose` class to enable separate transformation pipelines for the MRI images and their corresponding masks. MRI images typically have three channels and benefit from normalization (to standardize pixel values, often based on ImageNet statistics), whereas segmentation masks are single-

channel and usually represent `categorical labels` or grayscale intensities; **therefore, they do not require normalization**.

```
class DualCompose(object):
    def __init__(self, image_transforms, mask_transforms):
        self.image_transforms = image_transforms
        self.mask_transforms = mask_transforms

    def __call__(self, image, mask):
        if self.image_transforms:
            image = self.image_transforms(image)
        if self.mask_transforms:
            mask = self.mask_transforms(mask)
        return image, mask
```

```
mean = [0.485, 0.456, 0.406]
std  = [0.229, 0.224, 0.225]

image_transforms = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor(),
    transforms.Normalize(mean = mean,
                         std = std)
])

mask_transforms = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor()
])

dual_transform = DualCompose(image_transforms, mask_transforms)

root_dir = 'data/archive'
train_set = BrainSegmentationDataset(os.path.join(root_dir, 'train'), transforms=dual_transform)
valid_set = BrainSegmentationDataset(os.path.join(root_dir, 'valid'), transforms=dual_transform)
test_set  = BrainSegmentationDataset(os.path.join(root_dir, 'test'), transforms=dual_transform)

batch_size = 16
dataloaders = {
    'train': DataLoader(train_set, batch_size=batch_size, shuffle=True, num_workers=0),
    'valid': DataLoader(valid_set, batch_size=batch_size, shuffle=False, num_workers=0),
    'test':  DataLoader(test_set, batch_size=batch_size, shuffle=False, num_workers=0)
}
```

15.5. Visualizing our Dataset Again using DataLoaders

To visualize training images properly, we reverse the normalization process by denormalizing them. This ensures that the images are rendered in their original appearance, as they were before normalization was applied.

```
def denormalize_image(img_tensor, mean, std):
    """
    Denormalizes an image tensor that was normalized with the given mean and std.

    Args:
        img_tensor (numpy.ndarray): An image in CxHxW format with normalized values.
        mean (list or np.ndarray): The mean values used for normalization.
```

```
        std (list or np.ndarray): The std values used for normalization.

    Returns:
        numpy.ndarray: The denormalized image in HxWxC format with values clipped to [0, 1].
    """
    mean = np.array(mean).reshape(3, 1, 1)
    std = np.array(std).reshape(3, 1, 1)

    img_denorm = img_tensor * std + mean
    img_denorm = np.clip(img_denorm, 0, 1)

    return np.transpose(img_denorm, (1, 2, 0))

def plot_sample_pairs(dataloader, num_samples=3):
    """
    Plots a specified number of image and mask pairs from the given DataLoader.

    Args:
        dataloader (DataLoader): A PyTorch DataLoader that yields (image, mask) pairs.
        num_samples (int): The number of pairs to display. Defaults to 3.
    """
    images, masks = next(iter(dataloader))
    fig, axes = plt.subplots(2, num_samples, figsize=(num_samples * 4, 8))

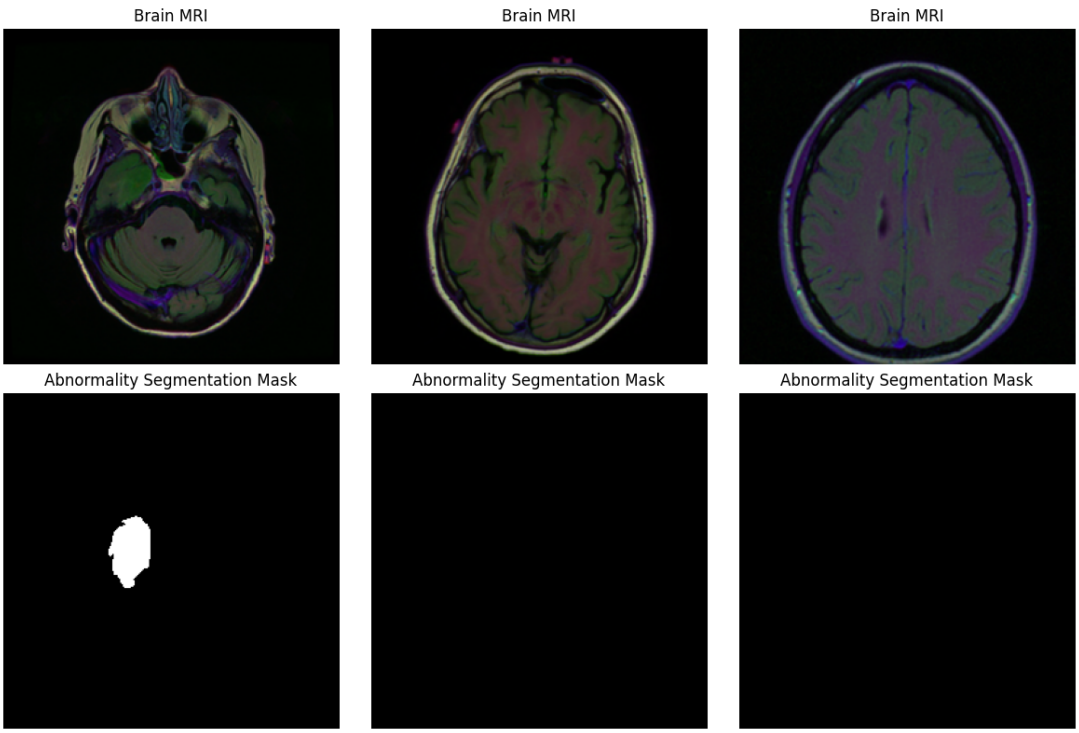
    for i in range(num_samples):
        img = images[i].cpu().numpy()
        img = denormalize_image(img, mean, std)

        axes[0, i].imshow(img)
        axes[0, i].axis('off')
        axes[0, i].set_title("Brain MRI")

        mask = masks[i].cpu().numpy()
        if mask.ndim == 3:
            mask = np.squeeze(mask)
        axes[1, i].imshow(mask, cmap='gray')
        axes[1, i].axis('off')
        axes[1, i].set_title("Abnormality Segmentation Mask")

    plt.tight_layout()
    plt.show()
```

```
plot_sample_pairs(dataloaders['train'])
```



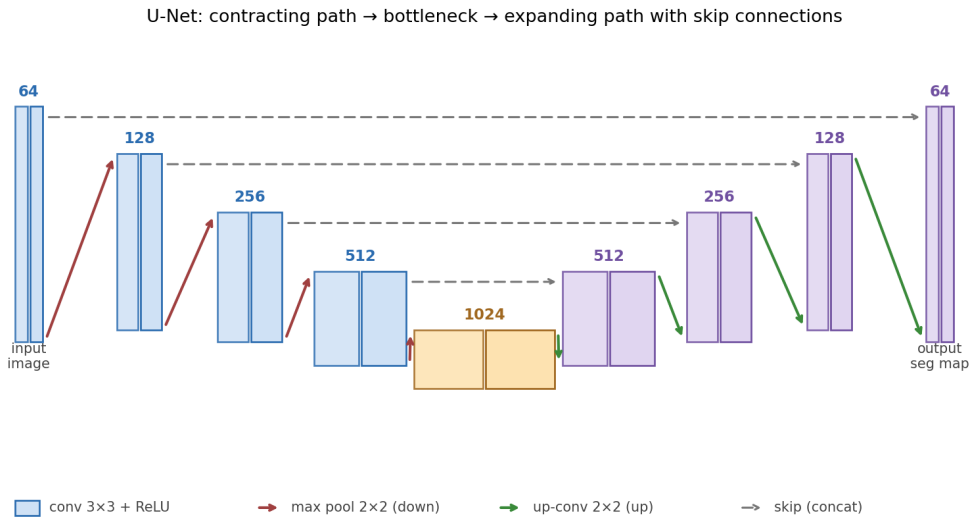
15.5.1. Createing a Neural Network with PyTorch

A Pytorch Neural Network model class is divided into two main components: - the initialization method, where all layers and submodules are defined - the forward method, which specifies how data flows through these layers.

In a simple **U-Net model**, the initialization phase sets up several key layers:

- `Conv2D layers` for extracting features (configured with parameters like the number of input and output channels, kernel size, and padding)
- `Activation layers` (typically using ReLU to introduce non-linearity)
- `MaxPool2d layers` for downsampling.

The forward method then arranges these layers into an **encoder-decoder** structure, with skip connections linking corresponding stages of the encoder and decoder to preserve fine spatial details essential for accurate segmentation.




```
class UNet(nn.Module):
    def __init__(self, n_class):
        super(UNet, self).__init__()
        # Encoder
        self.e11 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True)
        )
        self.e12 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True)
        )
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.e21 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True)
        )
        self.e22 = nn.Sequential(
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True)
        )
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.e31 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True)
        )
        self.e32 = nn.Sequential(
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True)
        )
        self.pool3 = nn.MaxPool2d(kernel_size=2, stride=2)

        self.e41 = nn.Sequential(
            nn.Conv2d(256, 512, kernel_size=3, padding=1),
```

```
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True)
        )
        self.e42 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(inplace=True)
        )
        # Decoder
        self.upconv3 = nn.ConvTranspose2d(512, 256, kernel_size=2, stride=2)
        self.d31 = nn.Sequential(
            nn.Conv2d(512, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True)
        )
        self.d32 = nn.Sequential(
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True)
        )

        self.upconv2 = nn.ConvTranspose2d(256, 128, kernel_size=2, stride=2)
        self.d21 = nn.Sequential(
            nn.Conv2d(256, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True)
        )
        self.d22 = nn.Sequential(
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True)
        )

        self.upconv1 = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
        self.d11 = nn.Sequential(
            nn.Conv2d(128, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True)
        )
        self.d12 = nn.Sequential(
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True)
        )

        # Output layer for binary segmentation: 1 channel with Sigmoid activation.
        self.outconv = nn.Conv2d(64, n_class, kernel_size=1)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # Encoder
        x1 = self.e11(x)
        x1 = self.e12(x1)
        p1 = self.pool1(x1)

        x2 = self.e21(p1)
        x2 = self.e22(x2)
        p2 = self.pool2(x2)

        x3 = self.e31(p2)
```

```
        x3 = self.e32(x3)
        p3 = self.pool3(x3)

        x4 = self.e41(p3)
        x4 = self.e42(x4)

        # Decoder
        up3 = self.upconv3(x4)
        merge3 = torch.cat([up3, x3], dim=1)
        d3 = self.d31(merge3)
        d3 = self.d32(d3)

        up2 = self.upconv2(d3)
        merge2 = torch.cat([up2, x2], dim=1)
        d2 = self.d21(merge2)
        d2 = self.d22(d2)

        up1 = self.upconv1(d2)
        merge1 = torch.cat([up1, x1], dim=1)
        d1 = self.d11(merge1)
        d1 = self.d12(d1)

        out = self.outconv(d1)
        out = self.sigmoid(out)
        return out
```

15.5.2. Setting Up the Model

```
model = UNet(n_class=1).to(device)
model
```

```
UNet(
  (e11): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (e12): Sequential(
    (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
  (e21): Sequential(
    (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (e22): Sequential(
    (0): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
[... 50 lines omitted ...]
    (2): ReLU(inplace=True)
  )
  (d12): Sequential(
    (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (outconv): Conv2d(64, 1, kernel_size=(1, 1), stride=(1, 1))
  (sigmoid): Sigmoid()
)
```

15.5.3. Model and its Parameters

The `torchsummary` package is an excellent tool for obtaining a concise overview of our model architecture. It provides detailed information on each layer, including output shapes, the number of parameters, and the overall computational complexity.

```
summary(model, (3, 256, 256))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 256, 256]           1,792
       BatchNorm2d-2         [-1, 64, 256, 256]             128
              ReLU-3         [-1, 64, 256, 256]               0
            Conv2d-4         [-1, 64, 256, 256]          36,928
       BatchNorm2d-5         [-1, 64, 256, 256]             128
              ReLU-6         [-1, 64, 256, 256]               0
         MaxPool2d-7         [-1, 64, 128, 128]               0
            Conv2d-8        [-1, 128, 128, 128]          73,856
       BatchNorm2d-9        [-1, 128, 128, 128]             256
             ReLU-10        [-1, 128, 128, 128]               0
           Conv2d-11        [-1, 128, 128, 128]         147,584
      BatchNorm2d-12        [-1, 128, 128, 128]             256
             ReLU-13        [-1, 128, 128, 128]               0
        MaxPool2d-14          [-1, 128, 64, 64]               0
           Conv2d-15          [-1, 256, 64, 64]         295,168
      BatchNorm2d-16          [-1, 256, 64, 64]             512
             ReLU-17          [-1, 256, 64, 64]               0
[... 33 lines omitted ...]
================================================================
Total params: 7,702,977
Trainable params: 7,702,977
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.75
Forward/backward pass size (MB): 767.00
Params size (MB): 29.38
Estimated Total Size (MB): 797.13
----------------------------------------------------------------
```

15.5.4. Optimizers and their options

There are several pre-built optimizers in PyTorch that work well in most scenarios when their parameters are appropriately tuned. Two of the most widely used optimizers are ADAM and SGD, both of which are based on gradient descent principles that we implemented earlier.

- **Stochastic Gradient Descent (SGD)**
- **Adaptive Moment Estimation (ADAM)**

For a comprehensive overview of optimization algorithms and best practices in gradient descent, check out this excellent resource.

```
optimizer = optim.Adam(model.parameters(), lr=1e-3)
# Learning rate scheduler: decay LR by 0.1 every 10 epochs.
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=10, gamma=0.1)
```

15.5.5. Loss Functions and Their Options

PyTorch provides a variety of built-in cost/loss functions that can be used for training deep learning models. These loss functions generally fall into two main categories: categorical losses and general losses.

One loss function particularly useful for highly unbalanced segmentation tasks is the Generalised Dice Loss, introduced by Sudre et al. (more details). It is defined as:

$$\text{Dice Loss} = 1 - \frac{2y\hat{y} + \epsilon}{y + \hat{y} + \epsilon}$$

where: - $\hat{y}$ represents the predicted segmentation, - y denotes the ground truth label, - ϵ is a small constant added for numerical stability and smoothness.

In our implementation, we combine a weighted cost function based on Binary Cross Entropy with the Dice Loss, allowing us to leverage both pixel-wise classification accuracy and the overlap-based metric to improve segmentation performance, particularly in scenarios with class imbalance.

```
# Loss Functions: Dice loss and a combined loss with Binary Cross-Entropy.
def dice_loss(pred, target, smooth=1.):
    pred = pred.contiguous()
    target = target.contiguous()
    intersection = (pred * target).sum(dim=2).sum(dim=2)
    loss = 1 - ((2. * intersection + smooth) / (pred.sum(dim=2).sum(dim=2) + target.sum(dim=2).sum(dim=2) +
   smooth))
    return loss.mean()

def loss_fn(pred, target, bce_weight=0.5):
    bce = F.binary_cross_entropy(pred, target)
    dice = dice_loss(pred, target)
    loss = bce * bce_weight + dice * (1 - bce_weight)
    return loss
```

15.6. Training

Training and evaluating a model typically involves three distinct phases:

- **Training:** The model is trained exclusively on the training dataset, during which its weights are updated based on the loss computed on this data.
- **Validation:** The validation dataset is used to monitor the training process. Although it does not directly influence the weight updates, it helps in tuning hyperparameters and detecting issues like overfitting.
- **Testing:** The test dataset is reserved for independent evaluation of the final model performance. It does not participate in any part of the training process, ensuring an unbiased assessment.

```
best_model_wts = copy.deepcopy(model.state_dict())
best_val_loss = float('inf')
history = {'train_loss': [], 'val_loss': []}
epochs_no_improve = 0
start_time = time.time()

num_epochs=10
early_stop_patience=3
bce_weight=0.5

for epoch in range(num_epochs):
    print(f"Epoch {epoch+1}/{num_epochs}")
    print("-" * 40)

    for phase in ['train', 'valid']:
        if phase == 'train':
            model.train()
        else:
            model.eval()

        running_loss = 0.0
        # Iterate over batches.
        for inputs, masks in dataloaders[phase]:
            inputs = inputs.to(device)
            masks = masks.to(device)
            optimizer.zero_grad()

            with torch.set_grad_enabled(phase == 'train'):
                outputs = model(inputs)
                loss = loss_fn(outputs, masks, bce_weight=bce_weight)

                if phase == 'train':
                    loss.backward()
                    optimizer.step()

            running_loss += loss.item() * inputs.size(0)

        epoch_loss = running_loss / len(dataloaders[phase].dataset)
        print(f"{phase.capitalize()} Loss: {epoch_loss:.4f}")

        if phase == 'train':
            history['train_loss'].append(epoch_loss)
        else:
            history['val_loss'].append(epoch_loss)
            if epoch_loss < best_val_loss:
                best_val_loss = epoch_loss
                best_model_wts = copy.deepcopy(model.state_dict())
                epochs_no_improve = 0
            else:
                epochs_no_improve += 1
```

```
    scheduler.step()
    print()

    if epochs_no_improve >= early_stop_patience:
        print("Early stopping triggered.")
        break

total_time = time.time() - start_time
print(f"Training complete in {total_time // 60:.0f}m {total_time % 60:.0f}s")
print(f"Best Validation Loss: {best_val_loss:.4f}")
```

```
Epoch 1/10
----------------------------------------
Train Loss: 0.5064
Valid Loss: 0.5066

Epoch 2/10
----------------------------------------
Train Loss: 0.4469
Valid Loss: 0.4367

Epoch 3/10
----------------------------------------
Train Loss: 0.3145
Valid Loss: 0.2470

Epoch 4/10
----------------------------------------
Train Loss: 0.2459
Valid Loss: 0.2932

[... 22 lines omitted ...]
Train Loss: 0.2064
Valid Loss: 0.1859

Epoch 10/10
----------------------------------------
Train Loss: 0.1877
Valid Loss: 0.1994

Training complete in 36m 51s
Best Validation Loss: 0.1859
```

15.6.1. Saving and Loading the Model

```
torch.save(best_model_wts, 'best_model.pt')

# Later, load the saved weights into your model
model.load_state_dict(torch.load('best_model.pt'))
```

```
<All keys matched successfully>
```

15.6.2. Testing our Model

```
def test_loop(model, dataloader, criterion, device):
    """
    Evaluates the model on the test dataset and computes the average test loss.
```

```
    Args:
        model (torch.nn.Module): The trained segmentation model.
        dataloader (DataLoader): DataLoader for the test dataset.
        criterion (callable): Loss function (e.g., your custom loss_fn).
        device (torch.device): Device to run evaluation on (CPU or GPU).

    Returns:
        float: Average test loss.
    """
    model.eval()  # Set model to evaluation mode
    running_loss = 0.0
    dataset_size = len(dataloader.dataset)

    with torch.no_grad():
        for inputs, masks in dataloader:
            inputs = inputs.to(device)
            masks = masks.to(device)

            outputs = model(inputs)
            loss = criterion(outputs, masks)
            running_loss += loss.item() * inputs.size(0)

    test_loss = running_loss / dataset_size
    print(f"Test Loss: {test_loss:.4f}")
    return test_loss
```

```
test_loss = test_loop(model, dataloaders['test'], loss_fn, device)
```

```
Test Loss: 0.1776
```

15.7. Visualizing our Predictions

```
def plot_predictions(model, dataloader, device, num_samples=3):
    """
    Plots test images along with their true and predicted segmentation masks.

    Args:
        model (torch.nn.Module): The trained segmentation model.
        dataloader (DataLoader): DataLoader for the test dataset.
        device (torch.device): Device to run inference on (CPU or GPU).
        num_samples (int): Number of test samples to display.
    """
    model.eval()
    # Get one batch of test data
    images, true_masks = next(iter(dataloader))
    images = images.to(device)

    with torch.no_grad():
        outputs = model(images)

    # For binary segmentation, we threshold the output at 0.5.
    preds = (outputs > 0.5).float()

    # Move tensors to CPU and convert to numpy arrays.
    images = images.cpu().numpy()
    true_masks = true_masks.cpu().numpy()
    preds = preds.cpu().numpy()

    # Define normalization parameters (if used during training)
```

```
    norm_mean = [0.485, 0.456, 0.406]
    norm_std  = [0.229, 0.224, 0.225]

    # Create a figure with 3 rows and num_samples columns:
    # Row 1: Original test images (denormalized)
    # Row 2: True masks
    # Row 3: Predicted masks
    fig, axes = plt.subplots(3, num_samples, figsize=(num_samples * 4, 12))

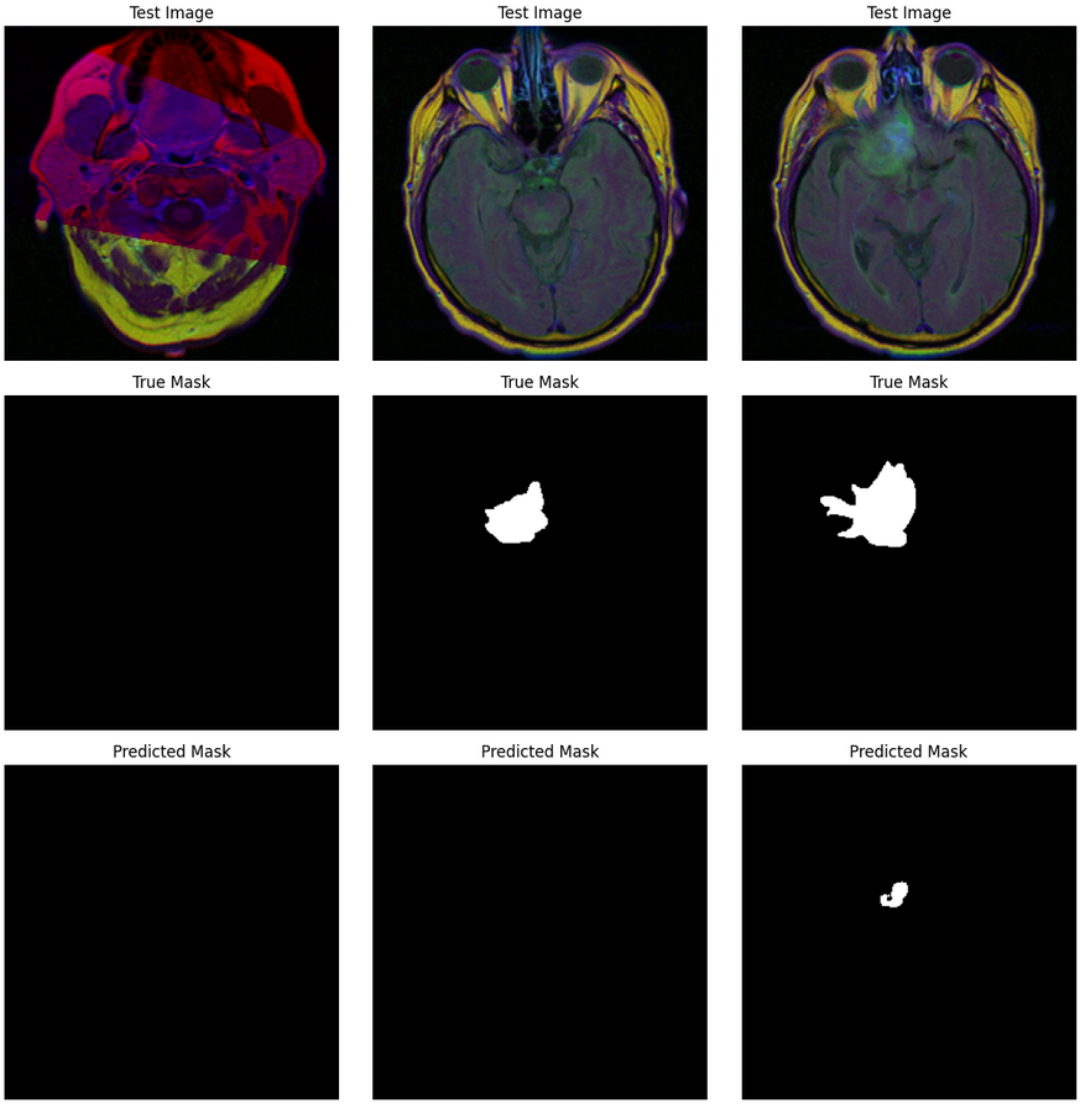
    for i in range(num_samples):
        # Process image: denormalize and convert from CxHxW to HxWxC.
        img = denormalize_image(images[i], norm_mean, norm_std)
        axes[0, i].imshow(img)
        axes[0, i].axis('off')
        axes[0, i].set_title("Test Image")

        # True mask: assume single channel
        true_mask = true_masks[i]
        if true_mask.shape[0] == 1:
            true_mask = true_mask[0]
        axes[1, i].imshow(true_mask, cmap="gray")
        axes[1, i].axis('off')
        axes[1, i].set_title("True Mask")

        # Predicted mask: assume single channel
        pred_mask = preds[i]
        if pred_mask.shape[0] == 1:
            pred_mask = pred_mask[0]
        axes[2, i].imshow(pred_mask, cmap="gray")
        axes[2, i].axis('off')
        axes[2, i].set_title("Predicted Mask")

    plt.tight_layout()
    plt.show()
```

```
plot_predictions(model, dataloaders['test'], device, num_samples=3)
```



15.8. Homework (Optional)

1. **BCE vs. Dice vs. combined.** Train three U-Net variants with pure BCE, pure Dice, and the combined loss. Report test Dice coefficient on brain-tumour segmentation for each.
2. **Encoder depth.** Reduce the U-Net to four encoder blocks instead of five, or extend to six. Report parameter count and test Dice for each depth and argue the sweet spot for 256×256 MRI slices.
3. **Data augmentation for medical imaging.** Add horizontal flip, small rotations, and elastic deformation to the training transforms. Retrain and report the change in Dice. Medical imaging augmentation must be careful: which transforms preserve anatomical validity?
4. **Boundary F1 score.** Implement a boundary-based metric that measures how well the predicted mask edges align with the ground-truth edges (rather than the full region overlap that Dice measures). Compare with per-pixel Dice on a handful of test samples.

5. **Post-processing.** Apply morphological opening then closing to the thresholded output mask. Show examples where this cleans up small islands and fills holes, and report the net Dice change on the test set.
6. **Transfer to a different modality.** Adapt the trained U-Net to a public CT-scan dataset (or any other single-channel medical dataset) by fine-tuning for 10 epochs. Compare from-scratch vs. fine-tuned Dice.

Chapter 16

Instance Segmentation with YOLO11

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1asBca8ROMn8swA4r11qMJoQ1oOiQzIfM?usp=sharing

Instance segmentation goes a step further than object detection and involves identifying individual objects in an image and segmenting them from the rest of the image.

The output of an instance segmentation model is a set of masks or contours that outline each object in the image, along with class labels and confidence scores for each object. Instance segmentation is useful when you need to know not only where objects are in an image, but also what their exact shape is.

16.1. Models

YOLO11 [17] pretrained Pose models are shown here. **Detect**, **Segment**, and **Pose** models are pretrained on the **COCO** dataset, while Classify models are pretrained on the **ImageNet** dataset.

Models download automatically from the latest Ultralytics release on first use.

Model	Size (pixels)	mAPbox 50-95	mAPmask 50-95	Speed CPU ONNX (ms)	Speed T4 TensorRT10 (ms)	Params (M)	FLOPs (B)
YOLO11n-seg	640	38.9	32.0	65.9 ± 1.1	1.8 ± 0.0	2.9	4.5
YOLO11s-seg	640	46.6	37.8	117.6 ± 4.9	2.9 ± 0.0	10.1	35.5
YOLO11m-seg	640	51.5	41.5	281.6 ± 1.2	6.3 ± 0.1	22.4	123.3
YOLO11l-seg	640	53.4	42.9	344.2 ± 3.2	7.8 ± 0.2	27.6	142.2
YOLO11x-seg	640	54.7	43.8	664.5 ± 3.2	15.8 ± 0.7	62.1	319.0

- **mAPval** values are for single-model single-scale on COCO val2017 dataset.
  - Reproduce by: `yolo val segment data=coco -seg.yaml device=0`
- **Speed** averaged over COCO [8] val images using an Amazon EC2 P4d instance.
  - Reproduce by: `yolo val segment data=coco-seg.yaml batch=1 device=0|cpu`

16.1.1. Key Points:

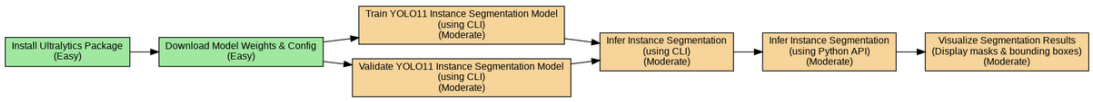
- **Different Models**:
  - The models range from **YOLO11n-seg** (smallest, fastest, least parameters) to **YOLO11x-seg** (largest, slowest, most parameters).
  - As you move from **n-seg** to **x-seg**, model size, accuracy (mAP), and computational requirements increase.
- **Why Choosing the Right Model Matters**:
  - **Speed vs. Accuracy**: Smaller models like **YOLO11n-seg** are faster and require fewer computational resources but have lower accuracy. On the other hand, **YOLO11x-seg** provides higher accuracy but is slower and more computationally expensive.
  - **Use Case Considerations**: Depending on your application (e.g., real-time scenarios, resource-limited devices, or cases needing high accuracy), you should choose a model that balances speed, accuracy, and hardware requirements.

YOLO11 Segment models use the -seg suffix, i.e. `yolo11n-seg.pt` and are pretrained on **COCO**.

16.2. Installing the Ultralytics Package

Ultralytics is the organization behind the development of YOLO (You Only Look Once) family of models for object detection, segmentation, and pose estimation, and provides tools and pretrained models for computer vision tasks.

```
!pip install -qq ultralytics
```



YOLO11 offers two distinct approaches for training, validating, and predicting with a model:

- **Command-Line Interface (CLI)**: A straightforward, non-Pythonic method that leverages the Ultralytics package to handle everything seamlessly in the background, without needing any code.
- **Pythonic Approach**: A more traditional method that allows for greater flexibility and customization, where you work directly with the Python API to manage the model's functionality programmatically.

16.3. Masks versus boxes: what instance segmentation adds

An object detector returns a bounding box per instance. An instance-segmentation model returns a bounding box *and* a pixel-precise binary mask per instance. The extra mask is what separates "there is a cat somewhere in this region" from "these exact pixels are the cat". For downstream tasks like medical image analysis, autonomous driving, and video editing, the per-pixel mask is the actionable output, not the box.

Architecturally, YOLO11-seg extends YOLO11-det with a lightweight *prototype* head that predicts a small number of shared mask prototypes per image (typically 32) plus per-instance mask coefficients. The final

mask of each detected object is a linear combination of the prototypes weighted by its coefficients. This is dramatically cheaper than predicting a full-resolution mask per instance, and it is why single-stage instance segmentation can match detection speed on modern hardware.

Evaluation parallels detection: mean Average Precision at fixed IoU thresholds, but computed over the *mask* IoU instead of the box IoU. A model can have high box mAP and low mask mAP if its boxes are tight but its masks are coarse.

16.4. Train

You have three options to choose from when training a model: - **Build a new model from YAML**: Start from scratch by specifying model architecture in a YAML file (includes a download script for the coco8-pose dataset as well). - **Load a pretrained model**: Use a pretrained model to fine-tune or continue training, recommended for most use cases. - **Build from YAML and transfer weights**: Define your model architecture in YAML and load pretrained weights for a head start on training.

Downloading the `yolo11-nano` model with the `coco8-seg` dataset.

```
!wget -qq https://github.com/ultralytics/assets/releases/download/v8.3.0/yolo11n-seg.pt
!wget -qq https://raw.githubusercontent.com/ultralytics/ultralytics/refs/heads/main/ultralytics/cfg/datasets/
  coco8-seg.yaml
```

The `coco8-seg.yaml` has the following format:

```
## **Ultralytics YOLO , AGPL-3.0 license**
## **COCO8-seg dataset (first 8 images from COCO train2017) by Ultralytics**
## **Documentation: <https://docs.ultralytics.com/datasets/segment/coco8-seg/**>
## **Example usage: yolo train data=coco8-seg.yaml**
## **parent**
## **+-- ultralytics**
## **+-- datasets**
## **+-- coco8-seg  <- downloads here (1 MB)**

## **Train/val/test sets as 1) dir: path/to/imgs, 2) file: path/to/imgs.txt, or 3) list: [path/to/imgs1, path/to/imgs2, ..]**
path: ../datasets/coco8-seg # dataset root dir
train: images/train # train images (relative to 'path') 4 images
val: images/val # val images (relative to 'path') 4 images
test: # test images (optional)

## **Classes**
names:
  0: person
  1: bicycle
  2: car
  3: motorcycle
  4: airplane
  5: bus
  6: train
  7: truck
  8: boat
  9: traffic light
  10: fire hydrant
  11: stop sign
  12: parking meter
  13: bench
  14: bird
```

```
15: cat
16: dog
17: horse
18: sheep
19: cow
20: elephant
21: bear
22: zebra
23: giraffe
24: backpack
25: umbrella
26: handbag
27: tie
28: suitcase
29: frisbee
30: skis
31: snowboard
32: sports ball
33: kite
34: baseball bat
35: baseball glove
36: skateboard
37: surfboard
38: tennis racket
39: bottle
40: wine glass
41: cup
42: fork
43: knife
44: spoon
45: bowl
46: banana
47: apple
48: sandwich
49: orange
50: broccoli
51: carrot
52: hot dog
53: pizza
54: donut
55: cake
56: chair
57: couch
58: potted plant
59: bed
60: dining table
61: toilet
62: tv
63: laptop
64: mouse
65: remote
66: keyboard
67: cell phone
68: microwave
69: oven
70: toaster
71: sink
72: refrigerator
73: book
74: clock
75: vase
```

```
  76: scissors
  77: teddy bear
  78: hair drier
  79: toothbrush

## **Download script/URL (optional)**
download: <https://github.com/ultralytics/assets/releases/download/v0.0.0/coco8-seg.zip>
```

16.5. Building a New Model (CLI)

The expected `mAP50` for validation is approximately 0.

All training runs, including metrics, plot images, CSV logs, and model weights, are saved in the directory: `/content/runs/segment/train`

```
!yolo segment train data=coco8-seg.yaml model=yolo11n-seg.yaml epochs=100 imgsz=640
```

```
Creating new Ultralytics Settings v0.0.6 file
View Ultralytics Settings with 'yolo settings' or at
'/root/.config/Ultralytics/settings.json'
Update Settings with 'yolo settings key=value', i.e. 'yolo settings
runs_dir=path/to/dir'. For help see
https://docs.ultralytics.com/quickstart/#ultralytics-settings.
Ultralytics 8.3.3  Python-3.10.12 torch-2.4.1+cu121 CPU (Intel Xeon 2.20GHz)
engine/trainer: task=segment, mode=train, model=yolo11n-seg.yaml,
data=coco8-seg.yaml, epochs=100, time=None, patience=100, batch=16, imgsz=640,
save=True, save_period ... [1405 chars truncated]

Dataset 'coco8-seg.yaml' images not found !, missing path
'/content/datasets/coco8-seg/images/val'
Downloading https://ultralytics.com/assets/coco8-seg.zip to
'/content/datasets/coco8-seg.zip'...
100% 439k/439k [00:00<00:00, 50.2MB/s]
Unzipping /content/datasets/coco8-seg.zip to /content/datasets/coco8-seg...:
100% 25/25 [00:00<00:00, 793.37file/s]
Dataset download success  (0.2s), saved to /content/datasets

Downloading https://ultralytics.com/assets/Arial.ttf to
'/root/.config/Ultralytics/Arial.ttf'...
100% 755k/755k [00:00<00:00, 68.4MB/s]

                   from  n    params  module
arguments
  0                  -1  1       464  ultralytics.nn.modules.conv.Conv
[3, 16, 3, 2]
  1                  -1  1      4672  ultralytics.nn.modules.conv.Conv
[16, 32, 3, 2]
  2                  -1  1      6640  ultralytics.nn.modules.block.C3k2
[32, 64, 1, False, 0.25]
  3                  -1  1     36992  ultralytics.nn.modules.conv.Conv
[64, 64, 3, 2]
[... 544 lines omitted ...]
Optimizer stripped from runs/segment/train/weights/best.pt, 6.1MB
```

```
Validating runs/segment/train/weights/best.pt...
Ultralytics 8.3.3  Python-3.10.12 torch-2.4.1+cu121 CPU (Intel Xeon 2.20GHz)
YOLO11n-seg summary (fused): 265 layers, 2,868,664 parameters, 0 gradients, 10.4
GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50
mAP50-95)     Mask(P          R      mAP50  mAP50-95): 100% 1/1 [00:01<00:00,
1.24s/it]
                   all          4         17          0          0          0
0          0          0          0          0          0
Speed: 2.4ms preprocess, 300.1ms inference, 0.0ms loss, 0.9ms postprocess per
image
Results saved to runs/segment/train
 Learn more at https://docs.ultralytics.com/modes/train
```

16.6. Loading a Pretrained Model (CLI)

The expected `mAP50` (class average precision) for validation is approximately 0.941. The mask average precision should be around `0.849`

All training runs, including metrics, plot images, CSV logs, and model weights, are saved in the directory: `/content/runs/segment/train2`

```
!yolo segment train data=coco8-seg.yaml model=yolo11n-seg.pt epochs=100 imgsz=640
```

```
Ultralytics 8.3.3  Python-3.10.12 torch-2.4.1+cu121 CPU (Intel Xeon 2.20GHz)
engine/trainer: task=segment, mode=train, model=yolo11n-seg.pt,
data=coco8-seg.yaml, epochs=100, time=None, patience=100, batch=16, imgsz=640,
save=True, save_period=- ... [1405 chars truncated]

                   from  n    params  module
arguments
  0                  -1  1       464  ultralytics.nn.modules.conv.Conv
[3, 16, 3, 2]
  1                  -1  1      4672  ultralytics.nn.modules.conv.Conv
[16, 32, 3, 2]
  2                  -1  1      6640  ultralytics.nn.modules.block.C3k2
[32, 64, 1, False, 0.25]
  3                  -1  1     36992  ultralytics.nn.modules.conv.Conv
[64, 64, 3, 2]
  4                  -1  1     26080  ultralytics.nn.modules.block.C3k2
[64, 128, 1, False, 0.25]
  5                  -1  1    147712  ultralytics.nn.modules.conv.Conv
[128, 128, 3, 2]
  6                  -1  1     87040  ultralytics.nn.modules.block.C3k2
[128, 128, 1, True]
  7                  -1  1    295424  ultralytics.nn.modules.conv.Conv
[128, 256, 3, 2]
  8                  -1  1    346112  ultralytics.nn.modules.block.C3k2
[256, 256, 1, True]
  9                  -1  1    164608  ultralytics.nn.modules.block.SPPF
[256, 256, 5]
```

```
 10                  -1  1    249728  ultralytics.nn.modules.block.C2PSA
[256, 256, 1]
 11                  -1  1         0  torch.nn.modules.upsampling.Upsample
[None, 2, 'nearest']
 12             [-1, 6]  1         0  ultralytics.nn.modules.conv.Concat
[1]
 13                  -1  1    111296  ultralytics.nn.modules.block.C3k2
[384, 128, 1, False]
 14                  -1  1         0  torch.nn.modules.upsampling.Upsample
[None, 2, 'nearest']
 15             [-1, 4]  1         0  ultralytics.nn.modules.conv.Concat
[1]
[... 537 lines omitted ...]
                   all          4         17      0.782      0.915      0.941
0.688      0.711      0.845      0.849       0.57
                person          3         10      0.738      0.567      0.671
0.324      0.738      0.567      0.616      0.259
                   dog          1          1      0.743          1      0.995
0.895      0.743          1      0.995      0.895
                 horse          1          2      0.624          1      0.995
0.673      0.624          1      0.995      0.224
              elephant          1          2          1      0.925      0.995
0.348      0.575        0.5        0.5       0.25
              umbrella          1          1      0.661          1      0.995
0.995      0.661          1      0.995      0.895
          potted plant          1          1      0.924          1      0.995
0.895      0.924          1      0.995      0.895
Speed: 2.3ms preprocess, 301.8ms inference, 0.0ms loss, 2.1ms postprocess per
image
Results saved to runs/segment/train2
 Learn more at https://docs.ultralytics.com/modes/train
```

16.7. Validation

You have two options to choose from when validating a model: - **Validate an official model**: `yolo11n-seg.pt`, `yolo11x-seg.pt` etc. - **Validate your trained model**: `best-seg.pt`

16.8. Validate an Official Model (CLI)

The expected `mAP50` (class average precision) for validation is approximately 0.917. Pose `mAP50` is approximately 0.772.

All training runs, including metrics, plot images, CSV logs, and model weights, are saved in the directory: `/content/runs/segment/val`

```
!yolo segment val model=yolo11n-seg.pt
```

```
WARNING ! 'data' argument is missing. Using default 'data=coco8-seg.yaml'.
Ultralytics 8.3.3  Python-3.10.12 torch-2.4.1+cu121 CPU (Intel Xeon 2.20GHz)
YOLO11n-seg summary (fused): 265 layers, 2,868,664 parameters, 0 gradients, 10.4
GFLOPs
```

```
val: Scanning /content/datasets/coco8-seg/labels/val.cache... 4 images, 0
backgrounds, 0 corrupt: 100% 4/4 [00:00<?, ?it/s]
                 Class     Images  Instances      Box(P          R      mAP50
mAP50-95)     Mask(P          R      mAP50  mAP50-95): 100% 1/1 [00:01<00:00,
1.71s/it]
                   all          4         17      0.762       0.85      0.886
0.675      0.762       0.85      0.853      0.568
                person          3         10      0.984        0.6      0.672
0.307      0.984        0.6      0.639      0.249
                   dog          1          1      0.738          1      0.995
0.895      0.738          1      0.995      0.895
                 horse          1          2      0.611          1      0.995
0.673      0.611          1      0.995      0.224
              elephant          1          2      0.639        0.5      0.662
0.285      0.639        0.5      0.501      0.251
              umbrella          1          1      0.672          1      0.995
0.995      0.672          1      0.995      0.895
          potted plant          1          1      0.928          1      0.995
0.895      0.928          1      0.995      0.895
Speed: 7.1ms preprocess, 333.4ms inference, 0.0ms loss, 3.4ms postprocess per
image
Results saved to runs/segment/val
 Learn more at https://docs.ultralytics.com/modes/val
```

16.9. Inference

You have two options to choose from when testing a model: - **Test an official model**: `yolo11n-seg.pt`, `yolo11x-seg.pt` etc. - **Test your trained model**: `best-seg.pt`

16.10. Infer an Official Model (CLI)

The output of YOLO's prediction, which includes an image with detected people and their corresponding keypoints, is saved in the `/content/runs/segment/predict` folder.

```
!yolo segment predict model=yolo11n-seg.pt source='https://ultralytics.com/images/bus.jpg'
```

```
Ultralytics 8.3.3  Python-3.10.12 torch-2.4.1+cu121 CPU (Intel Xeon 2.20GHz)
YOLO11n-seg summary (fused): 265 layers, 2,868,664 parameters, 0 gradients, 10.4
GFLOPs

Downloading https://ultralytics.com/images/bus.jpg to 'bus.jpg'...
100% 134k/134k [00:00<00:00, 28.2MB/s]
image 1/1 /content/bus.jpg: 640x480 4 persons, 1 bus, 1 stop sign, 408.9ms
Speed: 11.7ms preprocess, 408.9ms inference, 39.7ms postprocess per image at
shape (1, 3, 640, 480)
Results saved to runs/segment/predict
 Learn more at https://docs.ultralytics.com/modes/predict
```

16.11. Displaying the output

It's important to note that **YOLO** first performs object detection as a prerequisite for segmentation. This means the model must identify objects in the image before it can accurately generate and display their segmentation masks.



16.12. Infer an Official Model (Python)

Now, let's perform inference using a larger, more advanced model, `yolo11x-seg`, for improved results.

This step involves importing the necessary libraries and downloading the `yolo11x-seg` model.

```
!wget -qq https://github.com/ultralytics/assets/releases/download/v8.3.0/yolo11x-seg.pt

from PIL import Image
import ultralytics
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
import torch
import numpy as np
import random
import cv2
```

```
ultralytics.checks()
print(f"Setup complete. Using torch {torch.__version__} on device ({torch.cuda.get_device_properties(0).name
   if torch.cuda.is_available() else 'CPU'})")

# load an official model
model = ultralytics.YOLO("yolo11n-seg.pt")
# predict on multiple images
results = model(["/content/room.jpg", "/content/food.jpg"],
   conf=0.37)
```

```
Ultralytics 8.3.3  Python-3.10.12 torch-2.4.1+cu121 CPU (Intel Xeon 2.20GHz)
Setup complete  (2 CPUs, 12.7 GB RAM, 36.6/107.7 GB disk)
Setup complete. Using torch 2.4.1+cu121 on device (CPU)

0: 640x640 1 bed, 314.7ms
1: 640x640 3 cups, 1 fork, 1 spoon, 1 dining table, 314.7ms
Speed: 8.1ms preprocess, 314.7ms inference, 14.6ms postprocess per image at
shape (1, 3, 640, 640)
```

Visualizing the model architecture

```
model   # display value
```

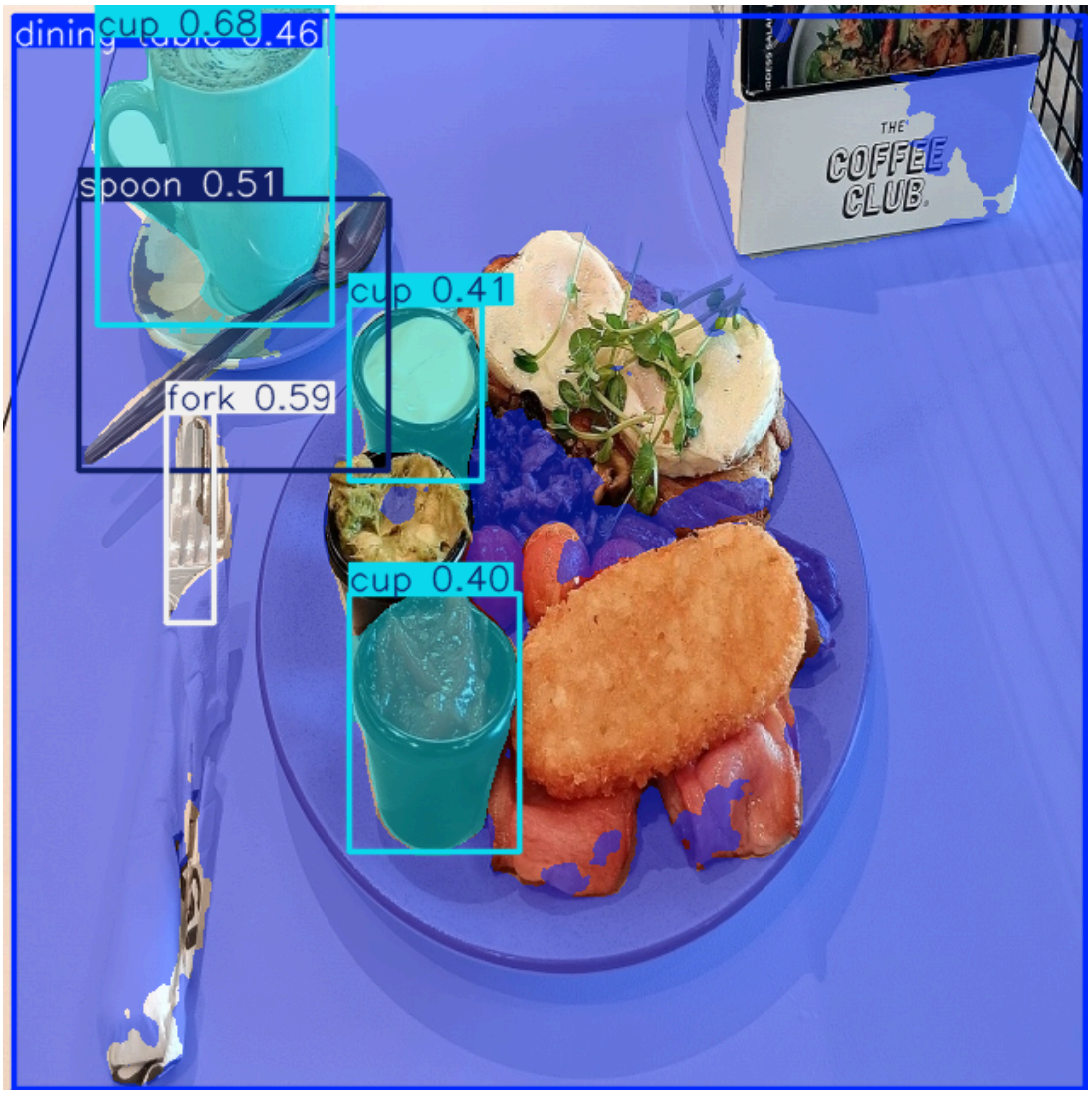
```
YOLO(
  (model): SegmentationModel(
    (model): Sequential(
      (0): Conv(
        (conv): Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (act): SiLU(inplace=True)
      )
      (1): Conv(
        (conv): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1))
        (act): SiLU(inplace=True)
      )
      (2): C3k2(
        (cv1): Conv(
          (conv): Conv2d(32, 32, kernel_size=(1, 1), stride=(1, 1))
          (act): SiLU(inplace=True)
        )
        (cv2): Conv(
          (conv): Conv2d(48, 64, kernel_size=(1, 1), stride=(1, 1))
          (act): SiLU(inplace=True)
        )
```

```
[... 474 lines omitted ...]
              (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
              (act): SiLU(inplace=True)
            )
            (2): Conv2d(32, 32, kernel_size=(1, 1), stride=(1, 1))
          )
        )
      )
    )
  )
)
```

Going trough the output for all images and displaying them.

```
for i, result in enumerate(results):
    result.show()  # display to screen
    result.save(filename=f"/content/runs/segment/result_{i}.jpg")  # save to disk
```






```
for i, result in enumerate(results):
    boxes = result.boxes  # Boxes object for bounding box outputs
    masks = result.masks  # Masks object for segmentation masks outputs
    print(f'Bounding boxes shape: {boxes.shape} - The first dimension represents the number of objects in the
  image, with each box consisting of 6 values: x_center, y_center, width, height, confidence, class.')
    print(f'Masks shape: {masks.shape} - The first dimension represents the number of objects in the image,
  while the rest of the dimensions match the original image shape (segmentation map).')
```

```
Bounding boxes shape: torch.Size([1, 6]) - The first dimension represents the
number of objects in the image, with each box consisting of 6 values: x_center,
y_center, width, height, confidence, class.
Masks shape: torch.Size([1, 640, 640]) - The first dimension represents the
number of objects in the image, while the rest of the dimensions match the
original image shape (segmentation map).
Bounding boxes shape: torch.Size([6, 6]) - The first dimension represents the
number of objects in the image, with each box consisting of 6 values: x_center,
y_center, width, height, confidence, class.
Masks shape: torch.Size([6, 640, 640]) - The first dimension represents the
```

```
number of objects in the image, while the rest of the dimensions match the
original image shape (segmentation map).
```

The data tensor contains the original image shape, with values of `0` and `1`. A value of `1` indicates where the mask is active (i.e., where an object is detected).

The `xy` tensor stores the pixel coordinates for all the locations containing an object.

Class labels are not stored in `result.masks`, but instead, they are located in `result.boxes`. **YOLO** first detects objects within an image using bounding boxes and classifies them into specific categories (classes) during object detection. After the object detection phase, it applies segmentation masks to the detected objects. Each segmentation mask highlights the pixels associated with a specific object.

Additionally, the `xyn` tensor contains the normalized `x` and `y` values for each masked pixel, ensuring that the positions are scaled appropriately.

```
print(masks[0])
```

```
ultralytics.engine.results.Masks object with attributes:

data: tensor([[[0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         ...,
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.],
         [0., 0., 0.,  ..., 0., 0., 0.]]])
orig_shape: (640, 640)
shape: torch.Size([1, 640, 640])
xy: [array([[        305,         526],
       [        305,         529],
       [        303,         531],
       [        301,         531],
       [        300,         532],
       [        298,         532],
       [        297,         533],
       [        295,         533],
       [        294,         534],
[... 575 lines omitted ...]
       [    0.58281,     0.83281],
       [    0.57969,     0.83281],
       [    0.57812,     0.83125],
       [    0.57344,     0.83125],
       [    0.57187,     0.82969],
       [    0.56875,     0.82969],
       [    0.56719,     0.82812],
       [    0.56563,     0.82812],
       [    0.56406,     0.82656],
       [    0.56406,     0.82187]], dtype=float32)]
```

16.13. Mask Plotting and Coverage Calculation

In segmentation tasks, each object instance is assigned a unique mask, which can be visualized by assigning different colors to each mask. The segmentation mask represents the object's pixels, allowing for visualization and analysis of the segmented regions.

To compute the **coverage percentage** of each object: 1. **Binary Mask**: A binary mask is created where each pixel is marked as `True` (non-zero) if it belongs to the object. 2. **Mask Area**: The total number of pixels in the binary mask represents the object's area. 3. **Coverage Percentage**: The object's coverage is calculated as the ratio of the mask area to the total image area:

$$\text{Coverage Percentage} = \left(\frac{\text{Object Area}}{\text{Total Image Area}} \right) \times 100$$

This allows us to visualize the segmented regions and quantify how much of the image each object covers.

16.14. IMPORTANT

The images used for testing were already sized to match the model's output (`640x640`), so no scaling was necessary. As a useful exercise, work with larger images (e.g., `1920x1080`) to generate a 640x640 mask from the model, and then practice rescaling the mask back to the original image dimensions.

```
for j, result in enumerate(results):
    masks = result.masks  # Segmentation mask outputs
    boxes = result.boxes  # Bounding box outputs (to get class IDs)

    # Initialize a blank segmentation map for plotting
    seg_map = np.zeros((result.orig_img.shape[0], result.orig_img.shape[1], 3), dtype=np.uint8)

    # Convert the mask tensor data to a numpy array
    masks_data = masks.data.numpy()
    mask_total_area = masks_data.shape[1] * masks_data.shape[2]

    # Prepare a list for the legend entries
    legend_entries = []

    # Predicted class IDs were Tensors. Needs detaching.
    class_ids = boxes.cls.detach().cpu().numpy()
    class_names = model.names

    for i, mask in enumerate(masks_data):
        # Generate a random color for this instance
        color = [random.randint(0, 255), random.randint(0, 255), random.randint(0, 255)]

        # Create a binary mask (True where mask is non-zero)
        binary_mask = mask > 0

        # Apply the color to the mask in the segmentation map
        seg_map[binary_mask] = color

        # Calculate the percentage of the image covered by this mask
        mask_area = np.sum(binary_mask)
        coverage_percentage = (mask_area / mask_total_area) * 100

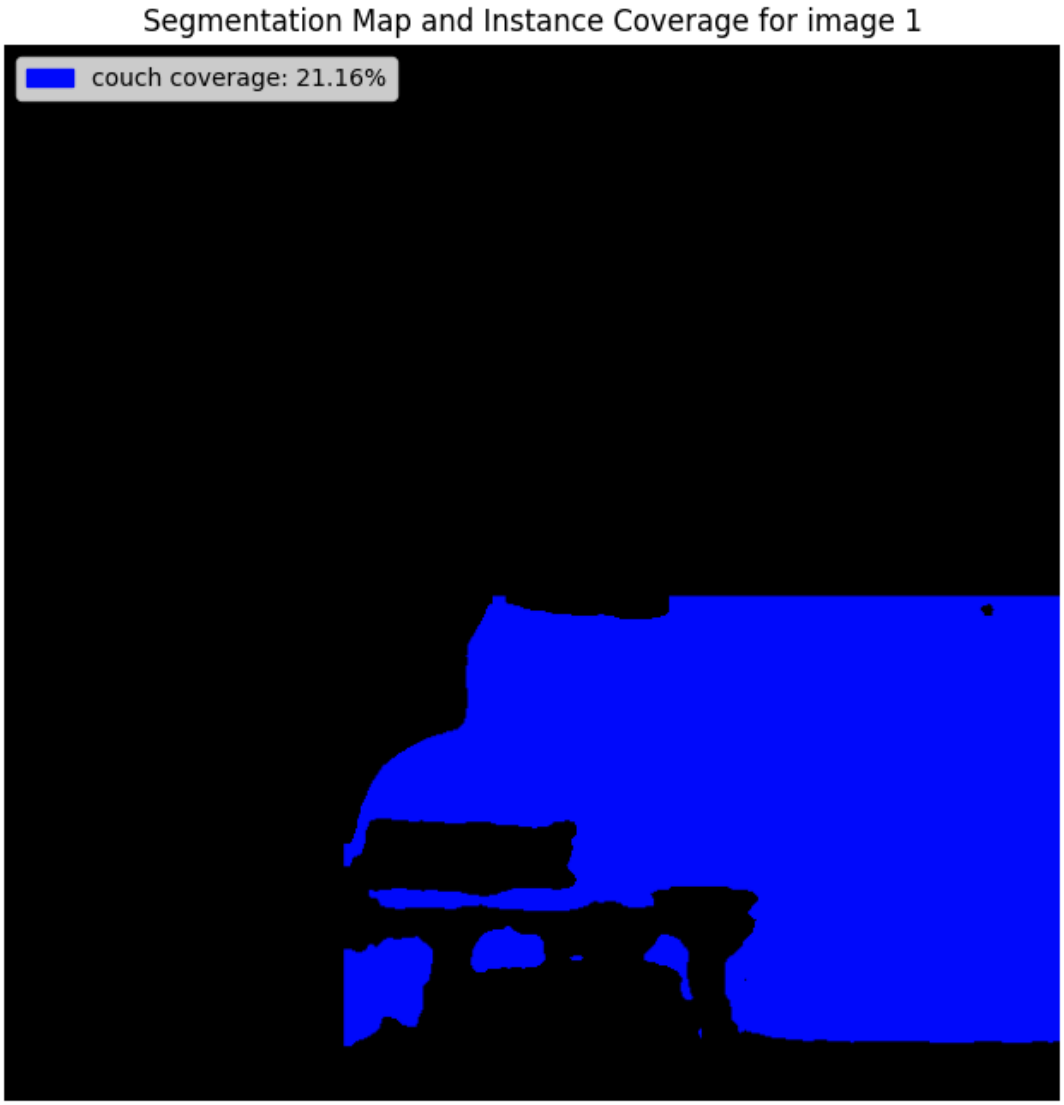
        class_label = class_names[class_ids[i]]
        legend_entries.append(mpatches.Patch(color=np.array(color) / 255, label=f"{class_label} coverage:
  {coverage_percentage:.2f}%"))

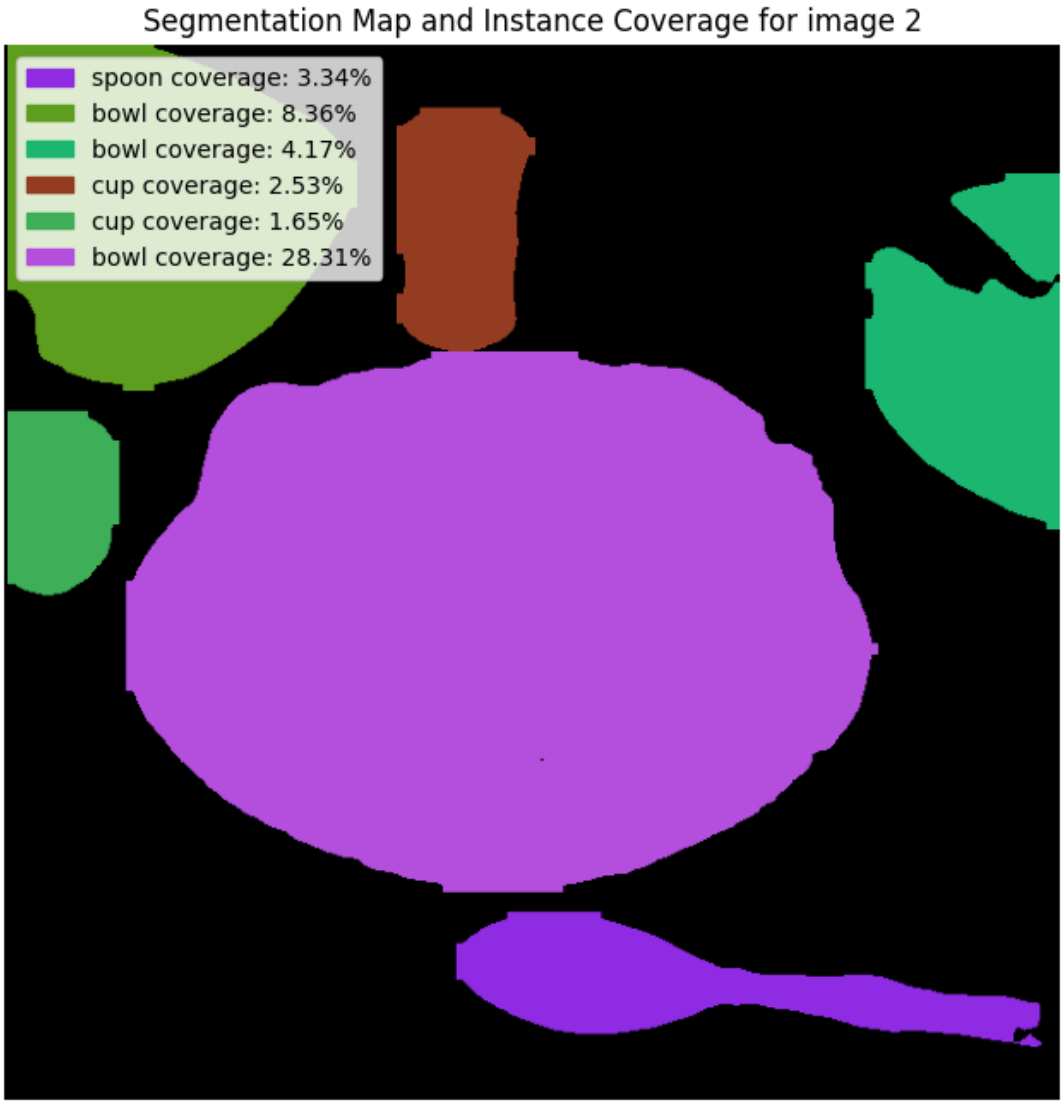
    plt.figure(figsize=(8, 8))
    plt.imshow(seg_map)
    plt.axis('off')
```

```
    plt.title(f"Segmentation Map and Instance Coverage for image {j+1}")

    # Create a legend in the top-left corner with colored squares
    plt.legend(handles=legend_entries, loc='upper left', bbox_to_anchor=(0, 1), fontsize=10, frameon=True)

    plt.show()
```





16.15. Homework (Optional)

1. **Mask mAP vs. box mAP.** Evaluate the trained model on the COCO val subset for both metrics. Report the gap between them and explain why mask mAP is almost always the lower number.
2. **Fine-tune on a custom instance.** Collect or download 50-100 images of one object class (e.g. a specific product, plant species, or tool). Annotate with polygons using `labelme` or Roboflow, train `yolo11n-seg` for 30 epochs, and evaluate mask mAP.
3. **Confidence threshold sweep.** Sweep the confidence threshold from `0.1` to `0.9` and plot precision and recall separately for boxes and masks. Identify the threshold that maximises mask F1.
4. **Mask post-processing.** Morphological opening/closing on the raw mask, plus contour smoothing. Demonstrate visually on three test images and discuss when this helps (and when it over-smooths).

5. **Mask coverage.** For a test image with multiple instances, compute the union-of-masks coverage fraction (pixels covered by at least one predicted mask divided by total image pixels). Use this to flag over- or under-segmented images.
6. **Prototype inspection.** Extract the 32 prototype masks from an intermediate layer. Plot a few of them. Do any of them correspond to interpretable patterns (e.g. "rounded object in centre", "vertical stripe")?

Chapter 17

Pose Estimation with YOLO11

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1hjzMAEJtTUNmuI3XhHm2nr99SiE29r3Q?usp=sharing

Pose estimation is a task that involves identifying the location of specific points in an image, usually referred to as keypoints. The keypoints can represent various parts of the object such as joints, landmarks, or other distinctive features. The locations of the keypoints are usually represented as a set of 2D `[x, y]` or 3D `[x, y, visible]` coordinates.

The output of a pose estimation model is a set of points that represent the keypoints on an object in the image, usually along with the confidence scores for each point. Pose estimation is a good choice when you need to identify specific parts of an object in a scene, and their location in relation to each other.

The key paper for YOLO-Pose can be found here.

17.1. Models

YOLO11 [17] pretrained Pose models are shown here. **Detect**, **Segment**, and **Pose** models are pretrained on the **COCO** dataset, while Classify models are pretrained on the **ImageNet** dataset.

Models download automatically from the latest Ultralytics release on first use.

Model	size (pixels)	mAPpose 50-95	mAPpose 50	Speed CPU ONNX (ms)	Speed T4 TensorRT10 (ms)	params (M)
YOLO11n-pose	640	50.0	81.0	52.4 ± 0.5	1.7 ± 0.0	2.9
YOLO11s-pose	640	58.9	86.3	90.5 ± 0.6	2.6 ± 0.0	9.9
YOLO11m-pose	640	64.9	89.4	187.3 ± 0.8	4.9 ± 0.1	20.9
YOLO11l-pose	640	66.1	89.9	247.7 ± 1.1	6.4 ± 0.1	26.2
YOLO11x-pose	640	69.5	91.1	488.0 ± 13.9	12.1 ± 0.2	58.8

- **mAPval** values are for single-model single-scale on COCO Keypoints val2017 dataset.
  - Reproduce by: `yolo val pose data=coco-pose.yaml device=0`

- **Speed** averaged over COCO [8] val images using an Amazon EC2 P4d instance.
  - Reproduce by: `yolo val pose data=coco-pose.yaml batch=1 device=0|cpu`

17.1.1. Key Points:

- **Different Models**:
  - The models range from **YOLO11n-pose** (smallest, fastest, least parameters) to **YOLO11x-pose** (largest, slowest, most parameters).
  - As you move from **n-pose** to **x-pose**, model size, accuracy (mAP), and computational requirements increase.
- **Why Choosing the Right Model Matters**:
  - **Speed vs. Accuracy**: Smaller models like **YOLO11n-pose** are faster and require fewer computational resources but have lower accuracy. On the other hand, **YOLO11x-pose** provides higher accuracy but is slower and more computationally expensive.
  - **Use Case Considerations**: Depending on your application (e.g., real-time scenarios, resource-limited devices, or cases needing high accuracy), you should choose a model that balances speed, accuracy, and hardware requirements.

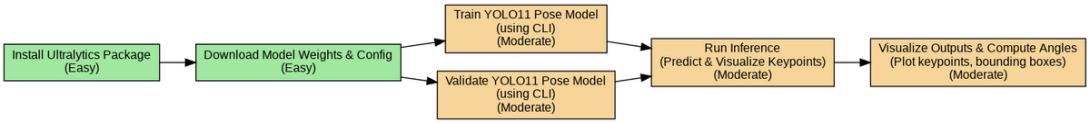
YOLO11 pose models use the `-pose` suffix, i.e. `yolo11n-pose.pt`. These models are trained on the **COCO keypoints** dataset and are suitable for a variety of pose estimation tasks. In the default **YOLO11 pose** model, there are **17** keypoints, each representing a different part of the human body. Here is the mapping of each index to its respective body joint:

`0`: Nose `1`: Left Eye `2`: Right Eye `3`: Left Ear `4`: Right Ear `5`: Left Shoulder `6`: Right Shoulder `7`: Left Elbow `8`: Right Elbow `9`: Left Wrist `10`: Right Wrist `11`: Left Hip `12`: Right Hip `13`: Left Knee `14`: Right Knee `15`: Left Ankle `16`: Right Ankle

17.2. Installing the Ultralytics Package

Ultralytics is the organization behind the development of YOLO (You Only Look Once) family of models for object detection, segmentation, and pose estimation, and provides tools and pretrained models for computer vision tasks.

```
!pip install -qq ultralytics
```



YOLO11 offers two distinct approaches for training, validating, and predicting with a model:

- **Command-Line Interface (CLI)**: A straightforward, non-Pythonic method that leverages the Ultralytics package to handle everything seamlessly in the background, without needing any code.
- **Pythonic Approach**: A more traditional method that allows for greater flexibility and customization, where you work directly with the Python API to manage the model's functionality programmatically.

17.3. Keypoints and visibility flags

Pose estimation predicts a *structured output* per detected person: a fixed ordered list of keypoints, each with an `(x, y)` pixel coordinate and a visibility flag. COCO's 17-keypoint skeleton (nose, eyes, ears, shoulders, elbows, wrists, hips, knees, ankles) is the de-facto standard and the format YOLO11-pose adopts.

The visibility flag takes three values: `0` (the keypoint is not annotated or not in the image), `1` (the keypoint is occluded but position is known) and `2` (the keypoint is visible). During training the loss is applied only to keypoints with `visibility > 0` so the network is not penalised for missing information.

At inference the model outputs a visibility probability per keypoint. Keypoints below a threshold are typically drawn faded or not at all. Most downstream uses (action recognition, gesture control, motion capture) rely on the per-keypoint confidence to ignore flaky detections rather than trust every predicted position.

17.4. Train

You have three options to choose from when training a model: - **Build a new model from YAML**: Start from scratch by specifying model architecture in a YAML file (includes a download script for the coco8-pose dataset as well). - **Load a pretrained model**: Use a pretrained model to fine-tune or continue training, recommended for most use cases. - **Build from YAML and transfer weights**: Define your model architecture in YAML and load pretrained weights for a head start on training.

Downloading the `yolo11-nano` model with the `coco8-pose` dataset.

```
!wget -qq https://github.com/ultralytics/assets/releases/download/v8.3.0/yolo11n-pose.pt
!wget -qq https://raw.githubusercontent.com/ultralytics/ultralytics/refs/heads/main/ultralytics/cfg/datasets/
  coco8-pose.yaml
```

The `coco8-pose.yaml` has the following format:

```
## **Ultralytics YOLO , AGPL-3.0 license**
## **COCO8-pose dataset (first 8 images from COCO train2017) by Ultralytics**
## **Documentation: <https://docs.ultralytics.com/datasets/pose/coco8-pose/**>
## **Example usage: yolo train data=coco8-pose.yaml**
## **parent**
## **+-- ultralytics**
## **+-- datasets**
## **+-- coco8-pose  <- downloads here (1 MB)**

## **Train/val/test sets as 1) dir: path/to/imgs, 2) file: path/to/imgs.txt, or 3) list: [path/to/imgs1, path/to/imgs2, ..]**
path: ../datasets/coco8-pose # dataset root dir
train: images/train # train images (relative to 'path') 4 images
val: images/val # val images (relative to 'path') 4 images
test: # test images (optional)

## **Keypoints**
kpt_shape: [17, 3] # number of keypoints, number of dims (2 for x,y or 3 for x,y,visible)
flip_idx: [0, 2, 1, 4, 3, 6, 5, 8, 7, 10, 9, 12, 11, 14, 13, 16, 15]

## **Classes**
names:
  0: person
```

```
## **Download script/URL (optional)**
download: <https://github.com/ultralytics/assets/releases/download/v0.0.0/coco8-pose.zip>
```

17.5. Building a New Model (CLI)

The expected `mAP50` (class average precision) for validation is approximately 0.0279.

All training runs, including metrics, plot images, CSV logs, and model weights, are saved in the directory: `/content/runs/pose/train`

```
!yolo pose train data=coco8-pose.yaml model=yolo11n-pose.yaml epochs=100 imgsz=640
```

```
Creating new Ultralytics Settings v0.0.6 file
View Ultralytics Settings with 'yolo settings' or at
'/root/.config/Ultralytics/settings.json'
Update Settings with 'yolo settings key=value', i.e. 'yolo settings
runs_dir=path/to/dir'. For help see
https://docs.ultralytics.com/quickstart/#ultralytics-settings.
Ultralytics 8.3.3  Python-3.10.12 torch-2.4.1+cu121 CPU (Intel Xeon 2.20GHz)
engine/trainer: task=pose, mode=train, model=yolo11n-pose.yaml,
data=coco8-pose.yaml, epochs=100, time=None, patience=100, batch=16, imgsz=640,
save=True, save_period= ... [1401 chars truncated]

Dataset 'coco8-pose.yaml' images not found !, missing path
'/content/datasets/coco8-pose/images/val'
Downloading https://ultralytics.com/assets/coco8-pose.zip to
'/content/datasets/coco8-pose.zip'...
100% 334k/334k [00:00<00:00, 2.78MB/s]
Unzipping /content/datasets/coco8-pose.zip to /content/datasets/coco8-pose...:
100% 27/27 [00:00<00:00, 676.21file/s]
Dataset download success  (1.4s), saved to /content/datasets

Downloading https://ultralytics.com/assets/Arial.ttf to
'/root/.config/Ultralytics/Arial.ttf'...
100% 755k/755k [00:00<00:00, 4.27MB/s]
Overriding model.yaml nc=80 with nc=1

                   from  n    params  module
arguments
  0                  -1  1       464  ultralytics.nn.modules.conv.Conv
[3, 16, 3, 2]
  1                  -1  1      4672  ultralytics.nn.modules.conv.Conv
[16, 32, 3, 2]
  2                  -1  1      6640  ultralytics.nn.modules.block.C3k2
[32, 64, 1, False, 0.25]
[... 61 lines omitted ...]
      Epoch    GPU_mem   box_loss  pose_loss  kobj_loss   cls_loss   dfl_loss
Instances       Size
      5/100         0G      3.442      8.535     0.7125      4.139      4.264
8        640: 100% 1/1 [00:03<00:00,  3.12s/it]
                 Class     Images  Instances      Box(P          R      mAP50
```

```
mAP50-95)     Pose(P          R      mAP50  mAP50-95): 100% 1/1 [00:01<00:00,
1.55s/it]
                   all          4         14    0.00583        0.5     0.0172
0.00547          0          0          0          0

      Epoch    GPU_mem   box_loss  pose_loss  kobj_loss   cls_loss   dfl_loss
Instances       Size
      6/100         0G      2.979      10.13     0.7338      3.723      4.272
17        640: 100% 1/1 [00:03<00:00,  3.57s/it]
                 Class     Images  Instances      Box(P          R      mAP50
mAP50-95)     Pose(P          R      mAP50  mAP50-95): 100% 1/1 [00:01<00:00,
1.09s/it]
                   all          4         14    0.00583        0.5     0.0153
0.00466          0          0          0          0
Traceback (most recent call last):
```

17.6. Loading a Pretrained Model (CLI)

The expected `mAP50` (class average precision) for validation is approximately 0.914.

All training runs, including metrics, plot images, CSV logs, and model weights, are saved in the directory: `/content/runs/pose/train2`

```
!yolo pose train data=coco8-pose.yaml model=yolo11n-pose.pt epochs=100 imgsz=640
```

```
Ultralytics 8.3.3  Python-3.10.12 torch-2.4.1+cu121 CPU (Intel Xeon 2.20GHz)
engine/trainer: task=pose, mode=train, model=yolo11n-pose.pt,
data=coco8-pose.yaml, epochs=100, time=None, patience=100, batch=16, imgsz=640,
save=True, save_period=-1 ... [1401 chars truncated]
Overriding model.yaml nc=80 with nc=1

                   from  n    params  module
arguments
  0                  -1  1       464  ultralytics.nn.modules.conv.Conv
[3, 16, 3, 2]
  1                  -1  1      4672  ultralytics.nn.modules.conv.Conv
[16, 32, 3, 2]
  2                  -1  1      6640  ultralytics.nn.modules.block.C3k2
[32, 64, 1, False, 0.25]
  3                  -1  1     36992  ultralytics.nn.modules.conv.Conv
[64, 64, 3, 2]
  4                  -1  1     26080  ultralytics.nn.modules.block.C3k2
[64, 128, 1, False, 0.25]
  5                  -1  1    147712  ultralytics.nn.modules.conv.Conv
[128, 128, 3, 2]
  6                  -1  1     87040  ultralytics.nn.modules.block.C3k2
[128, 128, 1, True]
  7                  -1  1    295424  ultralytics.nn.modules.conv.Conv
[128, 256, 3, 2]
  8                  -1  1    346112  ultralytics.nn.modules.block.C3k2
[256, 256, 1, True]
  9                  -1  1    164608  ultralytics.nn.modules.block.SPPF
```

```
[256, 256, 5]
 10                  -1  1    249728  ultralytics.nn.modules.block.C2PSA
[256, 256, 1]
 11                  -1  1         0  torch.nn.modules.upsampling.Upsample
[None, 2, 'nearest']
 12             [-1, 6]  1         0  ultralytics.nn.modules.conv.Concat
[1]
 13                  -1  1    111296  ultralytics.nn.modules.block.C3k2
[384, 128, 1, False]
 14                  -1  1         0  torch.nn.modules.upsampling.Upsample
[None, 2, 'nearest']
[... 532 lines omitted ...]
Optimizer stripped from runs/pose/train2/weights/best.pt, 6.1MB

Validating runs/pose/train2/weights/best.pt...
Ultralytics 8.3.3  Python-3.10.12 torch-2.4.1+cu121 CPU (Intel Xeon 2.20GHz)
YOLO11n-pose summary (fused): 257 layers, 2,866,468 parameters, 0 gradients, 7.4
GFLOPs
                 Class     Images  Instances      Box(P          R      mAP50
mAP50-95)     Pose(P          R      mAP50  mAP50-95): 100% 1/1 [00:00<00:00,
1.03it/s]
                   all          4         14      0.806       0.89      0.914
0.717          1      0.697      0.769      0.375
Speed: 2.5ms preprocess, 230.1ms inference, 0.0ms loss, 0.8ms postprocess per
image
Results saved to runs/pose/train2
 Learn more at https://docs.ultralytics.com/modes/train
```

17.7. Validation

You have two options to choose from when validating a model: - **Validate an official model**: `yolo11n-pose.pt`, `yolo11x-pose.pt` etc. - **Validate your trained model**: `best-pose.pt`

17.8. Validate an Official Model (CLI)

The expected `mAP50` (class average precision) for validation is approximately 0.917. Pose `mAP50` is approximately 0.772.

All training runs, including metrics, plot images, CSV logs, and model weights, are saved in the directory: `/content/runs/pose/val`

```
!yolo pose val model=yolo11n-pose.pt
```

```
WARNING ! 'data' argument is missing. Using default 'data=coco8-pose.yaml'.
Ultralytics 8.3.3  Python-3.10.12 torch-2.4.1+cu121 CPU (Intel Xeon 2.20GHz)
YOLO11n-pose summary (fused): 257 layers, 2,866,468 parameters, 0 gradients, 7.4
GFLOPs
val: Scanning /content/datasets/coco8-pose/labels/val.cache... 4 images, 0
backgrounds, 0 corrupt: 100% 4/4 [00:00<?, ?it/s]
                 Class     Images  Instances      Box(P          R      mAP50
mAP50-95)     Pose(P          R      mAP50  mAP50-95): 100% 1/1 [00:01<00:00,
```

```
1.35s/it]
                   all          4         14      0.811      0.929      0.917
0.728          1      0.697      0.772      0.355
Speed: 14.8ms preprocess, 307.9ms inference, 0.0ms loss, 0.8ms postprocess per
image
Results saved to runs/pose/val
 Learn more at https://docs.ultralytics.com/modes/val
```

17.9. Inference

You have two options to choose from when testing a model: - **Test an official model**: `yolo11n-pose.pt`, `yolo11x-pose.pt` etc. - **Test your trained model**: `best-pose.pt`

17.10. Infer an Official Model (CLI)

The output of YOLO's prediction, which includes an image with detected people and their corresponding keypoints, is saved in the `/content/runs/pose/predict` folder.

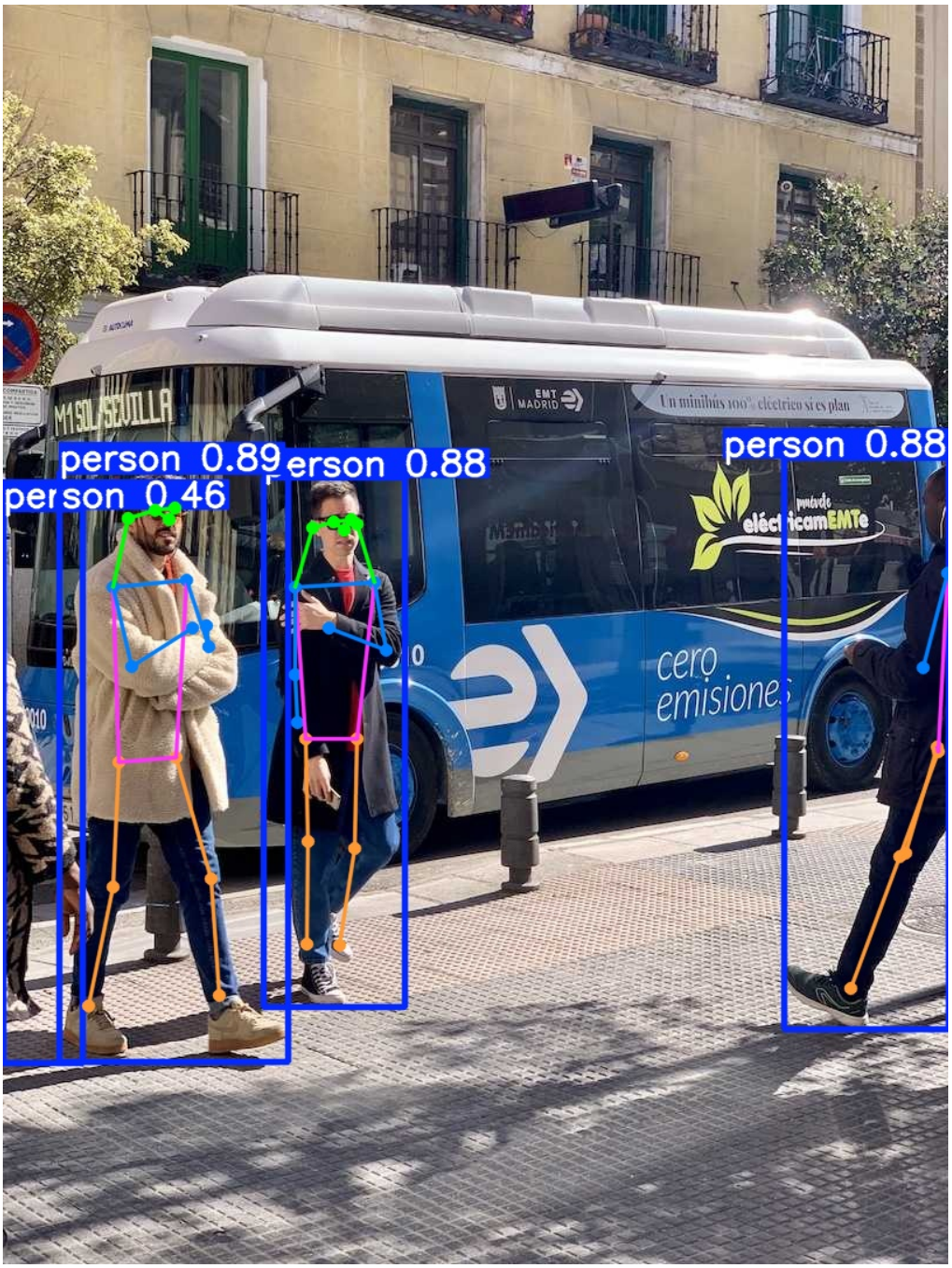
```
!yolo pose predict model=yolo11n-pose.pt source='https://ultralytics.com/images/bus.jpg'
```

```
Ultralytics 8.3.3  Python-3.10.12 torch-2.4.1+cu121 CPU (Intel Xeon 2.20GHz)
YOLO11n-pose summary (fused): 257 layers, 2,866,468 parameters, 0 gradients, 7.4
GFLOPs

Downloading https://ultralytics.com/images/bus.jpg to 'bus.jpg'...
100% 134k/134k [00:00<00:00, 1.47MB/s]
image 1/1 /content/bus.jpg: 640x480 4 persons, 205.1ms
Speed: 6.8ms preprocess, 205.1ms inference, 1.7ms postprocess per image at shape
(1, 3, 640, 480)
Results saved to runs/pose/predict
 Learn more at https://docs.ultralytics.com/modes/predict
```

17.11. Displaying the output

It's important to note that **YOLO** first performs object detection as a prerequisite for keypoint detection. This means the model must identify people in the image before it can accurately predict and display their keypoints.



17.12. Infer an Official Model (Python)

Now, let's perform inference using a larger, more advanced model, `yolo11x-pose`, for improved results.

This step involves importing the necessary libraries and downloading the `yolo11x-pose` model.

```
!wget -qq https://github.com/ultralytics/assets/releases/download/v8.3.0/yolo11x-pose.pt

from PIL import Image
import ultralytics
import matplotlib.pyplot as plt
import torch
import numpy as np
```

```
import random
import cv2
```

```
ultralytics.checks()
print(f"Setup complete. Using torch {torch.__version__} on device ({torch.cuda.get_device_properties(0).name
  if torch.cuda.is_available() else 'CPU'})")

# load an official model
model = ultralytics.YOLO("yolo11n-pose.pt")
# predict on multiple images
results = model(["https://ultralytics.com/images/zidane.jpg", "https://ultralytics.com/images/bus.jpg"])
```

```
Ultralytics 8.3.3  Python-3.10.12 torch-2.4.1+cu121 CPU (Intel Xeon 2.20GHz)
Setup complete  (2 CPUs, 12.7 GB RAM, 36.6/107.7 GB disk)
Setup complete. Using torch 2.4.1+cu121 on device (CPU)

0: 640x640 2 persons, 235.9ms
1: 640x640 4 persons, 235.9ms
Speed: 8.0ms preprocess, 235.9ms inference, 1.2ms postprocess per image at shape
(1, 3, 640, 640)
```

Visualizing the model architecture

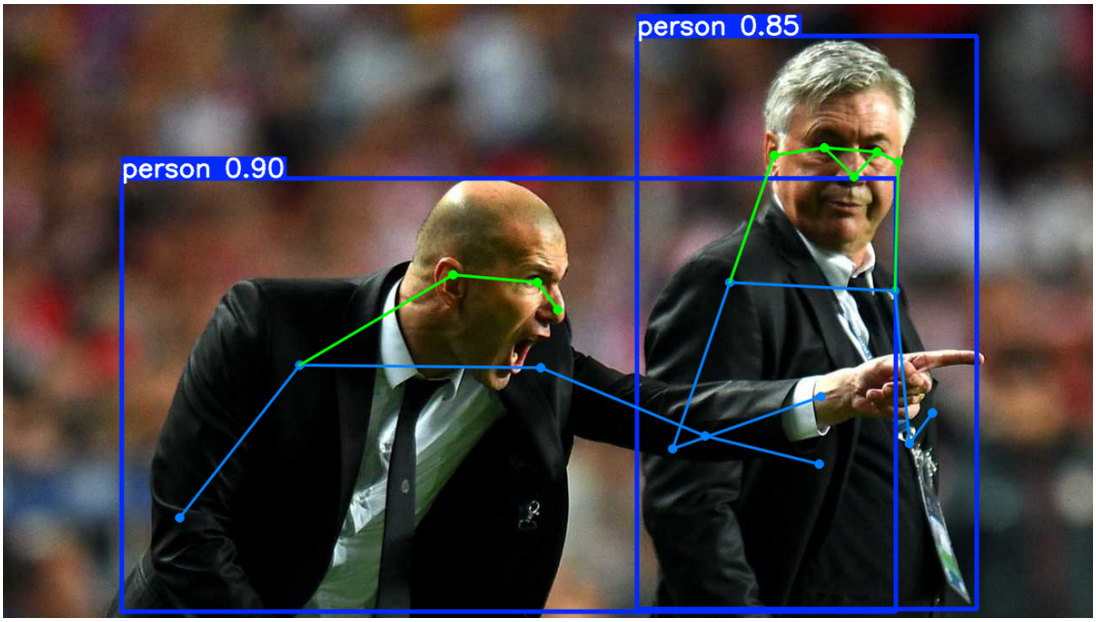
```
model   # display value
```

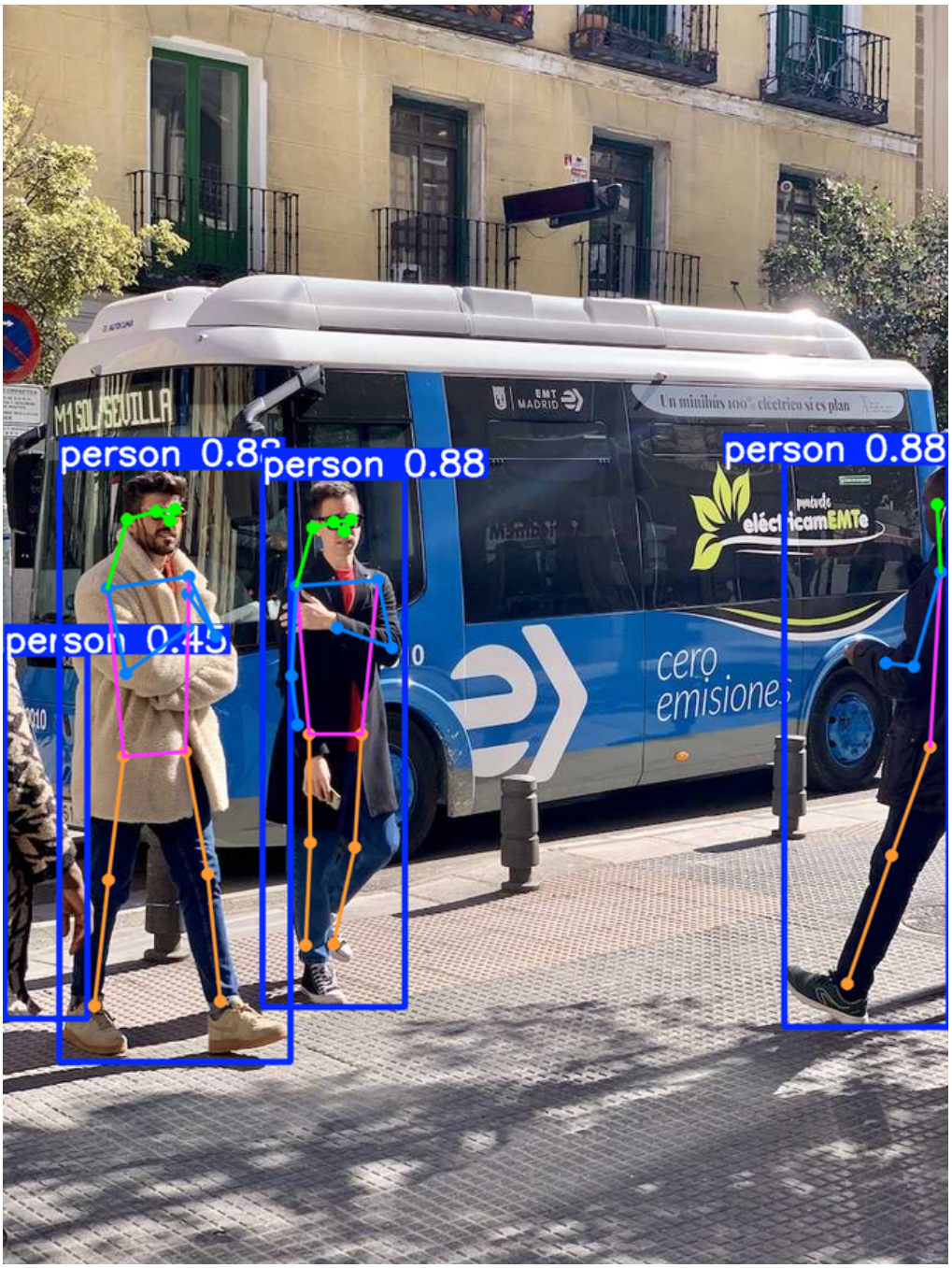
```
YOLO(
  (model): PoseModel(
    (model): Sequential(
      (0): Conv(
        (conv): Conv2d(3, 16, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (act): SiLU(inplace=True)
      )
      (1): Conv(
        (conv): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1))
        (act): SiLU(inplace=True)
      )
      (2): C3k2(
        (cv1): Conv(
          (conv): Conv2d(32, 32, kernel_size=(1, 1), stride=(1, 1))
          (act): SiLU(inplace=True)
        )
        (cv2): Conv(
          (conv): Conv2d(48, 64, kernel_size=(1, 1), stride=(1, 1))
          (act): SiLU(inplace=True)
        )
[... 459 lines omitted ...]
              (conv): Conv2d(51, 51, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
              (act): SiLU(inplace=True)
            )
            (2): Conv2d(51, 51, kernel_size=(1, 1), stride=(1, 1))
          )
        )
```

```
        )
    )
  )
)
```

Going through the output for all images and displaying them.

```
for i, result in enumerate(results):
    result.show()  # display to screen
    result.save(filename=f"/content/runs/pose/result_{i}.jpg")  # save to disk
```






```
for i, result in enumerate(results):
    boxes = result.boxes  # Bounding box outputs
    keypoints = result.keypoints  # Pose keypoint outputs
    print(f'Bounding boxes shape: {boxes.shape} - The first dimension represents the number of people in the
  image, with each box consisting of 6 values: x_center, y_center, width, height, confidence, class.')
    print(f'Keypoints shape: {keypoints.shape} - The first dimension represents the number of people in the
  image, with 17 keypoints per person. Each keypoint has 3 values.')
```

```
Bounding boxes shape: torch.Size([2, 6]) - The first dimension represents the
number of people in the image, with each box consisting of 6 values: x_center,
y_center, width, height, confidence, class.
Keypoints shape: torch.Size([2, 17, 3]) - The first dimension represents the
```

```
number of people in the image, with 17 keypoints per person. Each keypoint has 3
values.
Bounding boxes shape: torch.Size([4, 6]) - The first dimension represents the
number of people in the image, with each box consisting of 6 values: x_center,
y_center, width, height, confidence, class.
Keypoints shape: torch.Size([4, 17, 3]) - The first dimension represents the
number of people in the image, with 17 keypoints per person. Each keypoint has 3
values.
```

Each keypoint for a person contains a **confidence tensor with 17** values. These values represent how confident the model is for each corresponding keypoint.

The data tensor stores the `xy` coordinates for the keypoints, along with the confidence values, but with higher precision.

Additionally, the xyn tensor contains the normalized `x` and `y` values for each keypoint, ensuring that the positions are scaled appropriately.

```
print(keypoints[0])
```

```
ultralytics.engine.results.Keypoints object with attributes:

conf: tensor([[0.3312, 0.4495, 0.0920, 0.6189, 0.0953, 0.9085, 0.2751, 0.7790,
0.0435, 0.5104, 0.0477, 0.7993, 0.4204, 0.8042, 0.4388, 0.6483, 0.3711]])
data: tensor([[[0.0000e+00, 0.0000e+00, 3.3124e-01],
         [0.0000e+00, 0.0000e+00, 4.4953e-01],
         [0.0000e+00, 0.0000e+00, 9.1977e-02],
         [8.0045e+02, 4.2855e+02, 6.1891e-01],
         [0.0000e+00, 0.0000e+00, 9.5312e-02],
         [8.0364e+02, 4.8610e+02, 9.0845e-01],
         [0.0000e+00, 0.0000e+00, 2.7513e-01],
         [7.8002e+02, 5.6787e+02, 7.7898e-01],
         [0.0000e+00, 0.0000e+00, 4.3511e-02],
         [7.5680e+02, 5.6422e+02, 5.1036e-01],
         [0.0000e+00, 0.0000e+00, 4.7692e-02],
         [7.9476e+02, 6.3612e+02, 7.9933e-01],
         [0.0000e+00, 0.0000e+00, 4.2035e-01],
         [7.6167e+02, 7.2912e+02, 8.0421e-01],
         [0.0000e+00, 0.0000e+00, 4.3875e-01],
         [7.2400e+02, 8.3918e+02, 6.4829e-01],
         [0.0000e+00, 0.0000e+00, 3.7111e-01]]])
[... 27 lines omitted ...]
         [0.9630, 0.5258],
         [0.0000, 0.0000],
         [0.9343, 0.5224],
         [0.0000, 0.0000],
         [0.9812, 0.5890],
         [0.0000, 0.0000],
         [0.9403, 0.6751],
         [0.0000, 0.0000],
         [0.8938, 0.7770],
         [0.0000, 0.0000]]])
```

17.13. Angle Calculation

Understanding how points in 2D images are represented and manipulated is fundamental in computer vision tasks. Below is an example of how to calculate the angle between three key points without using the Ultralytics package for visualization. This method calculates the angle between the following keypoints:

- **Left Hip** (point A)
- **Left Knee** (point B - reference point)
- **Left Ankle** (point C)

The angle θ at the knee can be computed using vector mathematics. Given three points in 2D space: - $ A(x_1, y_1) $ - Left Hip - $ B(x_2, y_2) $ - Left Knee (reference point) - $ C(x_3, y_3) $ - Left Ankle

The angle θ between the vectors $\overrightarrow{BA}$ and $\overrightarrow{BC}$ can be calculated using the **dot product** formula:

$$\cos(\theta) = \frac{\overrightarrow{BA} \cdot \overrightarrow{BC}}{|\overrightarrow{BA}||\overrightarrow{BC}|}$$

Where: - $\overrightarrow{BA} = (x_1 - x_2, y_1 - y_2)$ - $\overrightarrow{BC} = (x_3 - x_2, y_3 - y_2)$

Once the cosine of the angle is found, the angle θ can be determined by:

$$\theta = \arccos\left(\frac{(x_1 - x_2)(x_3 - x_2) + (y_1 - y_2)(y_3 - y_2)}{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \cdot \sqrt{(x_3 - x_2)^2 + (y_3 - y_2)^2}}\right)$$

```python
from math import degrees, acos

def check_keypoints(keypoints, tolerance=1e-6):
    for point in keypoints:
        # Check if both x and y values are near zero
        if np.abs(point[0]) < tolerance and np.abs(point[1]) < tolerance:
            return False
    return True

def calculate_angle(pointA, pointB, pointC):
    # Create vectors
    BA = pointA - pointB
    BC = pointC - pointB

    # Dot product
    dot_product = np.dot(BA, BC)

    # Magnitudes
    magnitude_BA = np.linalg.norm(BA)
    magnitude_BC = np.linalg.norm(BC)

    # Calculate the cosine of the angle
    cos_theta = dot_product / (magnitude_BA * magnitude_BC)

    # Return the angle in degrees
    angle = degrees(acos(cos_theta))
    return angle
```

```
for result in results:
    # Contains keypoints for all people detected
    keypoints = result.keypoints.xy
    # Check if any keypoints were detected
    if keypoints.numel() > 0:
        rgb_image = cv2.cvtColor(result.orig_img, cv2.COLOR_BGR2RGB)
        plt.figure(figsize=(10, 6))
        plt.imshow(rgb_image)
        plt.axis('off')

        # Iterate through detected persons and plot their keypoints
        for i, person_keypoints in enumerate(keypoints):

            # Example: Calculate angle between three keypoints (hip, knee, ankle)
            pointA = person_keypoints[11]  # Example: left hip
            pointB = person_keypoints[13]  # Example: left knee
            pointC = person_keypoints[15]  # Example: left ankle

            if not check_keypoints([pointA, pointB, pointC]):
                print(f"No valid keypoints for the left leg (hip, knee, or ankle missing) for person {i}")
            else:
                angle = calculate_angle(pointA, pointB, pointC)
                print(f'Angle at the knee for person {i}: {angle:.2f} degrees')

            for x, y in person_keypoints:
                plt.scatter(x, y, color='red', s=10)

        plt.show()
```

```
No valid keypoints for the left leg (hip, knee, or ankle missing) for person 0
No valid keypoints for the left leg (hip, knee, or ankle missing) for person 1
```



```
Angle at the knee for person 0: 179.31 degrees
Angle at the knee for person 1: 171.32 degrees
```

```
Angle at the knee for person 2: 175.86 degrees
No valid keypoints for the left leg (hip, knee, or ankle missing) for person 3
```



17.14. Homework (Optional)

1. **Per-keypoint accuracy.** Evaluate the pretrained model on the COCO val subset and report mean OKS (Object Keypoint Similarity) plus per-keypoint mean Euclidean error. Which keypoints are hardest?
2. **Angle between joints.** Implement a function that computes the angle at the elbow from shoulder, elbow and wrist keypoint coordinates. Apply it to a test image and plot the angle as an annotation.
3. **Simple exercise classifier.** Given a short video of a person doing squats, track the knee and hip keypoints and use their vertical oscillation to count repetitions. Report the count and compare to the manual count.
4. **Visibility threshold sweep.** Vary the keypoint-visibility threshold from `0.1` to `0.9`. For each, compute OKS and a subjective assessment of how many correctly-visible keypoints are hidden vs. how many spurious ones are shown.

5. **Temporal smoothing.** On a video input, apply a moving-average filter over consecutive frames' keypoint positions. Demonstrate the reduction in jitter and discuss the trade-off with responsiveness to quick movements.
6. **Single-person vs. multi-person.** Run the model on a crowded scene with five or more people. Examine failure cases (crossed skeletons, missing persons) and propose one practical mitigation.

Chapter 18

Segment Anything Model 2

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1qr0PTdUWTUueNHwdW4UJ-Ua8-wBLmdLT?usp=sharing

SAM 2, the successor to Meta's Segment Anything Model (SAM), is a cutting-edge tool designed for comprehensive **object segmentation** in both images and videos. It excels in handling complex visual data through a unified, promptable model architecture that supports real-time processing and zero-shot generalization.

The key paper for SAM 2 can be found here.

SAM 2 predicts object masks given prompts that indicate the desired object. The model first converts the image into an image embedding that allows high quality masks to be efficiently produced from a prompt.

The `SAM2ImagePredictor` class provides an easy interface to the model for prompting the model. It allows the user to first set an image using the `set_image` method, which calculates the necessary image embeddings. Then, prompts can be provided via the `predict` method to efficiently predict masks from those prompts. The model can take as input both **point** and **box prompts**, as well as **masks from the previous iteration of prediction**.

Key features of SAM 2: - **Real-Time Performance:** the model achieves real-time inference speeds, processing approximately 44 frames per second. This makes SAM 2 suitable for applications requiring immediate feedback, such as video editing and augmented reality. - **Zero-Shot Generalization:** SAM 2 can segment objects it has never encountered before, demonstrating strong zero-shot generalization. This is particularly useful in diverse or evolving visual domains where pre-defined categories may not cover all possible objects. - **Interactive Refinement:** users can iteratively refine the segmentation results by providing additional prompts, allowing for precise control over the output. This interactivity is essential for fine-tuning results in applications like video annotation or medical imaging. - **Advanced Handling of Visual Challenges:** SAM 2 includes mechanisms to manage common video segmentation challenges, such as object occlusion and reappearance. It uses a sophisticated memory mechanism to keep track of objects across frames, ensuring continuity even when objects are temporarily obscured or exit and re-enter the scene. - **Memory Mechanism and Occlusion Handling:** the memory mechanism allows SAM 2 to handle temporal dependencies and occlusions in video data. As objects move and interact, SAM 2 records their features in a memory bank. When an object becomes occluded, the model can rely on this memory to predict its position and appearance when it reappears. The occlusion head specifically handles scenarios where objects are not visible, predicting the likelihood of an object being occluded. - **Multi-Mask Ambiguity Resolution**: in situations with ambiguity (e.g., overlapping objects), SAM 2 can

generate multiple mask predictions. This feature is crucial for accurately representing complex scenes where a single mask might not sufficiently describe the scene's nuances.

The following table details the available SAM 2 models, their pre-trained weights, supported tasks, and compatibility with different operating modes like Inference, Validation, Training, and Export.

Model Type	Pre-trained Weights	Tasks Supported	Inference	Validation	Training	Export
SAM 2 tiny	sam2_t.pt	Instance Segmentation				
SAM 2 small	sam2_s.pt	Instance Segmentation				
SAM 2 base	sam2_b.pt	Instance Segmentation				
SAM 2 large	sam2_l.pt	Instance Segmentation				

18.1. SA-V Dataset

The SA-V dataset, developed for SAM 2's training, is one of the largest and most diverse video segmentation datasets available. It includes:

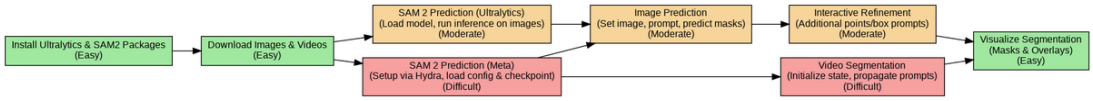
- **51,000+ Videos**: Captured across 47 countries, providing a wide range of real-world scenarios.
- **600,000+ Mask Annotations**: Detailed spatio-temporal mask annotations, referred to as "masklets," covering whole objects and parts.
- **Dataset Scale**: It features 4.5 times more videos and 53 times more annotations than previous largest datasets, offering unprecedented diversity and complexity.

18.2. SAM 2 Architecture

18.3. Installing the Ultralytics Package

Ultralytics is the organization behind the development of YOLO (You Only Look Once) family of models for object detection, segmentation, and pose estimation, and provides tools and pretrained models for computer vision tasks.

```
!pip install -qq ultralytics
```



18.4. Downloading images and videos for testing

```
!mkdir -p images
!wget -qq -P images https://raw.githubusercontent.com/facebookresearch/sam2/main/notebooks/images/truck.jpg
!wget -qq -P images https://raw.githubusercontent.com/facebookresearch/sam2/main/notebooks/images/groceries.jpg

!mkdir -p videos
!wget -qq -P videos https://dl.fbaipublicfiles.com/segment_anything_2/assets/bedroom.zip
!unzip -qq -d videos videos/bedroom.zip
```

18.5. Prompt types and how to combine them

SAM 2 is not a classifier, it does not know "what a car is". It accepts a *prompt* that says "segment the thing at these pixels" and returns a mask. Three prompt modalities are supported.

Point prompts are the simplest: one or more `(x, y)` pixel coordinates, each tagged as *foreground* (inside the object) or *background* (outside). A single foreground point usually gives a reasonable mask. Refining with background points carves away spurious extensions.

Box prompts are the most informative per prompt. A tight bounding box constrains the mask almost fully; SAM 2 only has to decide the fine boundary within the box. This is the right mode when boxes come from a detector upstream.

Mask prompts accept an existing low-resolution mask as a prior and refine it. Useful in interactive pipelines where the user has just approved a coarse prediction and wants the boundary cleaned up.

Prompts can be combined: a box plus a couple of background points inside it is the classic pattern for "segment the person but not the shopping bag they're holding". The trade-off is always between precision (more prompts, more work) and interactive latency (more prompts, slower round-trip).

For video, the same prompt types work at a chosen frame, and SAM 2's memory module then propagates the mask across neighbouring frames without re-prompting. Adding a new prompt on a distant frame refines the track only from that point onward.

18.6. SAM 2 Prediction Example (Ultralytics)

SAM 2 can be utilized across a broad spectrum of tasks, including real-time video editing, medical imaging, and autonomous systems. Its ability to segment both static and dynamic visual data makes it a versatile tool for researchers and developers.

```
from ultralytics import SAM
from IPython.display import Image

# Load a model
model = SAM("sam2_b.pt")

# Display model information (optional)
model.info()
```

```
Creating new Ultralytics Settings v0.0.6 file
View Ultralytics Settings with 'yolo settings' or at
'/root/.config/Ultralytics/settings.json'
Update Settings with 'yolo settings key=value', i.e. 'yolo settings
```

```
runs_dir=path/to/dir'. For help see
https://docs.ultralytics.com/quickstart/#ultralytics-settings.
Downloading
https://github.com/ultralytics/assets/releases/download/v8.3.0/sam2_b.pt to
'sam2_b.pt'...
```

```
100%|##########| 154M/154M [00:01<00:00, 120MB/s]
```

```
Model summary: 566 layers, 80,833,666 parameters, 80,833,666 gradients
```

```
(566, 80833666, 80833666, 0.0)
```

18.7. Plotting our test images





18.8. Run Inference on SAM 2

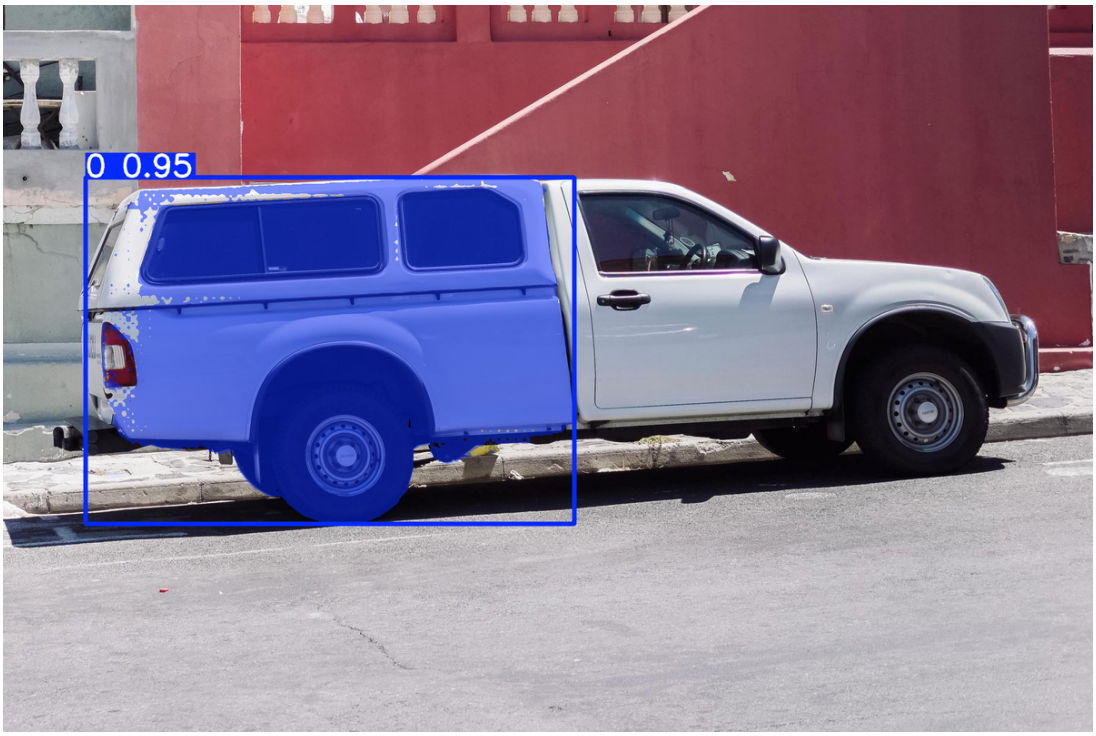
```
from PIL import Image
image_path = '/content/images/truck.jpg'
image = Image.open(image_path)
width, height = image.size

print(f"Shape of the image is: {width}x{height}")

# Segment with bounding box prompt
results = model("/content/images/truck.jpg", bboxes=[250, 250, 900, 900],  points=[500, 800], labels=[1])
for result in results:
    result.show()
    result.save(filename=f"/content/result_truck.jpg")  # save to disk
```

```
Shape of the image is: 1800x1200


image 1/1 /content/images/truck.jpg: 1024x1024 1 0, 3650.7ms
Speed: 148.8ms preprocess, 3650.7ms inference, 188.6ms postprocess per image at
shape (1, 3, 1024, 1024)
```




```
image_path = '/content/images/groceries.jpg'
image = Image.open(image_path)
width, height = image.size

print(f"Shape of the image is: {width}x{height}")

# Segment with bounding box prompt
results = model("/content/images/groceries.jpg",  points=[350, 350], labels=[1])
for result in results:
    result.show()
    result.save(filename=f"/content/result_groceries.jpg")  # save to disk
```

```
Shape of the image is: 800x534

image 1/1 /content/images/groceries.jpg: 1024x1024 1 0, 336.5ms
Speed: 11.4ms preprocess, 336.5ms inference, 0.9ms postprocess per image at
shape (1, 3, 1024, 1024)
```



18.9. SAM 2 Prediction Example (Meta)

18.10. Environment Set-up

In Colab, be sure to select 'GPU' under 'Edit'->'Notebook Settings'->'Hardware accelerator'. Note that it's recommended to use **A100 or L4 GPUs when running in Colab** (T4 GPUs might also work, but could be slow and might run out of memory in some cases).

```
import torch
import torchvision
import sys
import os
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

print("PyTorch version:", torch.__version__)
print("Torchvision version:", torchvision.__version__)
print("CUDA is available:", torch.cuda.is_available())

!{sys.executable} -m pip install -qq opencv-python matplotlib # packages for plotting
!{sys.executable} -m pip install -qq 'git+https://github.com/facebookresearch/sam2.git' # the SAM2 model

!mkdir -p checkpoints/
!wget -qq -P checkpoints/ https://dl.fbaipublicfiles.com/segment_anything_2/092824/sam2.1_hiera_large.pt #
  pretrained large model
```

```
PyTorch version: 2.4.1+cu121
Torchvision version: 0.19.1+cu121
CUDA is available: True
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Preparing metadata (pyproject.toml) ... done
     ---------------------------------------- 42.2/42.2 kB 2.0 MB/s eta 0:00:00
  Preparing metadata (setup.py) ... done
     ---------------------------------------- 117.0/117.0 kB 7.7 MB/s eta
0:00:00
  Preparing metadata (setup.py) ... done
    ---------------------------------------- 154.5/154.5 kB 15.0 MB/s eta 0:00:00
    ---------------------------------------- 79.5/79.5 kB 7.3 MB/s eta 0:00:00
  Building wheel for SAM-2 (pyproject.toml) ... done
  Building wheel for antlr4-python3-runtime (setup.py) ... done
  Building wheel for iopath (setup.py) ... done
```

```
!mkdir sam2_configs
!echo "" > __init__.py
!echo "" > sam2_configs/__init__.py
!wget -qq -P sam2_configs/ https://raw.githubusercontent.com/facebookresearch/sam2/refs/heads/main/sam2/
  configs/sam2.1/sam2.1_hiera_l.yaml
```

18.11. Set-up Device and Configs

```
import hydra

if torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")

hydra.core.global_hydra.GlobalHydra.instance().clear()
hydra.initialize_config_module('sam2_configs', version_base='1.2')

print(f"using device: {device}")
```

```
using device: cuda
```

18.11.1. Loading the SAM 2 video predictor

```
from sam2.build_sam import build_sam2
from sam2.sam2_image_predictor import SAM2ImagePredictor

# Loading in the SAM 2 model with its configuration file
sam2_checkpoint = "/content/checkpoints/sam2.1_hiera_large.pt"
model_cfg = "sam2.1_hiera_l.yaml"

sam2_model = build_sam2(model_cfg, sam2_checkpoint, device=device)

predictor = SAM2ImagePredictor(sam2_model)
```

```
/usr/local/lib/python3.10/dist-packages/sam2/modeling/sam/transformer.py:23:
UserWarning: Flash Attention is disabled as it requires a GPU with Ampere (8.0)
```

```
CUDA capability.
  OLD_GPU, USE_FLASH_ATTN, MATH_KERNEL_ON = get_sdpa_settings()
```

18.12. Setup Helper functions for plotting

```
def show_mask(mask, ax, random_color=False, borders = True):
    if random_color:
        color = np.concatenate([np.random.random(3), np.array([0.6])], axis=0)
    else:
        color = np.array([30/255, 144/255, 255/255, 0.6])
    h, w = mask.shape[-2:]
    mask = mask.astype(np.uint8)
    mask_image =  mask.reshape(h, w, 1) * color.reshape(1, 1, -1)
    if borders:
        import cv2
        contours, _ = cv2.findContours(mask,cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
        # Try to smooth contours
        contours = [cv2.approxPolyDP(contour, epsilon=0.01, closed=True) for contour in contours]
        mask_image = cv2.drawContours(mask_image, contours, -1, (1, 1, 1, 0.5), thickness=2)
    ax.imshow(mask_image)

def show_points(coords, labels, ax, marker_size=375):
    pos_points = coords[labels==1]
    neg_points = coords[labels==0]
    ax.scatter(pos_points[:, 0], pos_points[:, 1], color='green', marker='*', s=marker_size,
   edgecolor='white', linewidth=1.25)
    ax.scatter(neg_points[:, 0], neg_points[:, 1], color='red', marker='*', s=marker_size, edgecolor='white',
   linewidth=1.25)

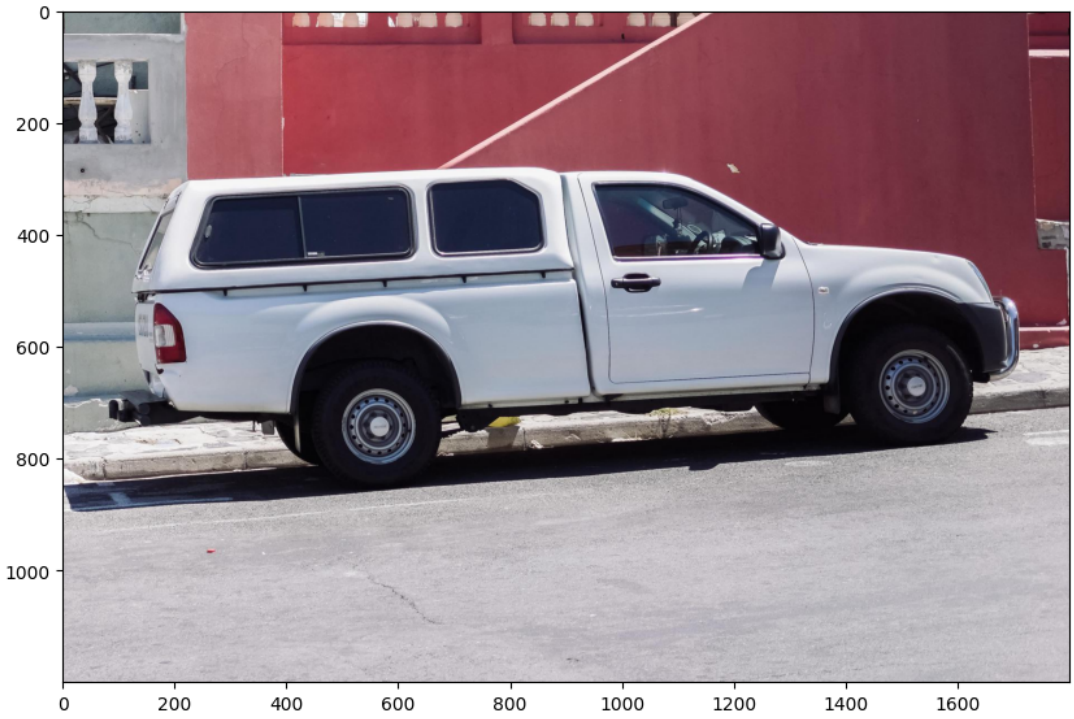
def show_box(box, ax):
    x0, y0 = box[0], box[1]
    w, h = box[2] - box[0], box[3] - box[1]
    ax.add_patch(plt.Rectangle((x0, y0), w, h, edgecolor='green', facecolor=(0, 0, 0, 0), lw=2))

def show_masks(image, masks, scores, point_coords=None, box_coords=None, input_labels=None, borders=True):
    for i, (mask, score) in enumerate(zip(masks, scores)):
        plt.figure(figsize=(10, 10))
        plt.imshow(image)
        show_mask(mask, plt.gca(), borders=borders)
        if point_coords is not None:
            assert input_labels is not None
            show_points(point_coords, input_labels, plt.gca())
        if box_coords is not None:
            # boxes
            show_box(box_coords, plt.gca())
        if len(scores) > 1:
            plt.title(f"Mask {i+1}, Score: {score:.3f}", fontsize=18)
        plt.axis('off')
        plt.show()
```

18.13. Plotting test images

```
image = Image.open('images/truck.jpg')
image = np.array(image.convert("RGB"))
plt.figure(figsize=(10, 10))
plt.imshow(image)
```

```
plt.axis('on')
plt.show()
```




```
image2 = Image.open('images/groceries.jpg')
image2 = np.array(image2.convert("RGB"))
plt.figure(figsize=(10, 10))
plt.imshow(image2)
plt.axis('on')
plt.show()
```



Process the image to produce an image embedding by calling `SAM2ImagePredictor.set_image`. `SAM2ImagePredictor` remembers this embedding and will use it for subsequent mask prediction.

```
predictor.set_image(image2)
```

To select the car, choose a point on it. Points are input to the model in (x,y) format and come with labels 1 (foreground point) or 0 (background point). Multiple points can be input. Here we use only one. The chosen point will be shown as a star on the image.

```
input_point = np.array([[100, 250]])
input_label = np.array([1])
```

```
plt.figure(figsize=(10, 10))
plt.imshow(image2)
show_points(input_point, input_label, plt.gca())
plt.axis('on')
plt.show()
```



18.13.1. Printing out the shape of image embeddings

```
print(predictor._features["image_embed"].shape, predictor._features["image_embed"][-1].shape)
```

```
torch.Size([1, 256, 64, 64]) torch.Size([256, 64, 64])
```

Predict with `SAM2ImagePredictor.predict`. The model returns masks, quality predictions for those masks, and low resolution mask logits that can be passed to the next iteration of prediction.

```
# predictor object already has the image with its embeddings
masks, scores, logits = predictor.predict(
    point_coords=input_point,
    point_labels=input_label,
    multimask_output=True,
)
sorted_ind = np.argsort(scores)[::-1]
masks = masks[sorted_ind]
scores = scores[sorted_ind]
logits = logits[sorted_ind]

masks.shape  # (number_of_masks) x H x W
```

```
(3, 534, 800)
```

With `multimask_output=True` (the default setting), SAM 2 outputs 3 masks, where `scores` gives the model's own estimation of the quality of these masks. This setting is intended for ambiguous input prompts, and helps the model disambiguate different objects consistent with the prompt. When `False`, it will return a single mask. For ambiguous prompts such as a single point, it is recommended to use

`multimask_output=True` even if only a single mask is desired. The best single mask can be chosen by picking the one with the highest score returned in `scores`. This will often result in a better mask.

```
show_masks(image2, masks, scores, point_coords=input_point, input_labels=input_label, borders=True)
```







Using a single point prompt, **SAM** effectively generates outputs at varying levels of granularity, producing masks that capture larger composite objects as confidence decreases.

```
input_point = np.array([[550, 300], [450, 320]])
input_label = np.array([1, 1]) # If specified as 0 it excludes the point

mask_input = logits[np.argmax(scores), :, :]  # Choose the model's best mask

masks, scores, _ = predictor.predict(
    point_coords=input_point,
    point_labels=input_label,
    mask_input=mask_input[None, :, :],
    multimask_output=False,
)
```

```
show_masks(image2, masks, scores, point_coords=input_point, input_labels=input_label)
```



```
input_point = np.array([[550, 300], [450, 320]])
input_label = np.array([0, 1]) # If specified as 0 it excludes the point

mask_input = logits[np.argmax(scores), :, :]  # Choose the model's best mask

masks, scores, _ = predictor.predict(
    point_coords=input_point,
    point_labels=input_label,
    mask_input=mask_input[None, :, :],
    multimask_output=False,
)
```

```
show_masks(image2, masks, scores, point_coords=input_point, input_labels=input_label)
```



18.14. Specifying a specific object with a box

The model can also take a box as input, provided in `xyxy` format.

```
input_box = np.array([420, 180, 520, 340])

masks, scores, _ = predictor.predict(
    point_coords=None,
    point_labels=None,
    box=input_box[None, :],
    multimask_output=False,
)
```

```
show_masks(image2, masks, scores, box_coords=input_box)
```



18.15. Combining points and boxes

Points and boxes may be combined, just by including both types of prompts to the predictor. Here this can be used to select just the grocery bags, instead of the entire car.

```
input_box = np.array([100, 50, 700, 500])
input_point = np.array([[550, 300]])
input_label = np.array([1])

masks, scores, logits = predictor.predict(
    point_coords=input_point,
    point_labels=input_label,
    box=input_box,
    multimask_output=False,
)
```

```
show_masks(image2, masks, scores, box_coords=input_box, point_coords=input_point, input_labels=input_label)
```



```
input_box = np.array([100, 50, 700, 500])
input_point = np.array([[550, 300]])
input_label = np.array([0])

masks, scores, logits = predictor.predict(
    point_coords=input_point,
    point_labels=input_label,
    box=input_box,
    multimask_output=False,
)
```

```
show_masks(image2, masks, scores, box_coords=input_box, point_coords=input_point, input_labels=input_label)
```



18.16. Video segmentation with SAM 2 (Meta)

```
from sam2.build_sam import build_sam2_video_predictor
predictor = build_sam2_video_predictor(model_cfg, sam2_checkpoint, device=device)
```

We assume that the video is stored as a list of JPEG frames with filenames like `.jpg`.

For your custom videos, you can extract their JPEG frames using ffmpeg (https://ffmpeg.org/) as follows:

`ffmpeg -i .mp4 -q:v 2 -start_number 0 /'%05d.jpg'`

where `-q:v` generates high-quality JPEG frames and `-start_number 0` asks ffmpeg to start the JPEG file from `00000.jpg`.

```
# `video_dir` a directory of JPEG frames with filenames like `<frame_index>.jpg`
video_dir = "./videos/bedroom"

# scan all the JPEG frame names in this directory
frame_names = [
    p for p in os.listdir(video_dir)
    if os.path.splitext(p)[-1] in [".jpg", ".jpeg", ".JPG", ".JPEG"]
]
frame_names.sort(key=lambda p: int(os.path.splitext(p)[0]))

# take a look the first video frame
frame_idx = 0
plt.figure(figsize=(9, 6))
plt.title(f"frame {frame_idx}")
plt.imshow(Image.open(os.path.join(video_dir, frame_names[frame_idx])))
```

Initialize the inference state

SAM 2 requires stateful inference for interactive video segmentation, so we need to initialize an **inference state** on this video.

During initialization, it loads all the JPEG frames in `video_path` and stores their pixels in `inference_state` (as shown in the progress bar below).

```
inference_state = predictor.init_state(video_path=video_dir)
```

```
frame loading (JPEG): 100%|##########| 200/200 [00:07<00:00, 27.53it/s]
```

18.16.1. Example 1: Segment & track one object

Step 1: Add a first click on a frame

To get started, let's try to segment the child on the left.

Here we make a **positive click** at (x, y) = (210, 350) with label `1`, by sending their coordinates and labels into the `add_new_points_or_box` API.

Note: label `1` indicates a *positive click (to add a region)* while label `0` indicates a *negative click (to remove a region)*.

```
def show_mask_video(mask, ax, obj_id=None, random_color=False):
    if random_color:
        color = np.concatenate([np.random.random(3), np.array([0.6])], axis=0)
    else:
        cmap = plt.get_cmap("tab10")
        cmap_idx = 0 if obj_id is None else obj_id
        color = np.array([*cmap(cmap_idx)[:3], 0.6])
    h, w = mask.shape[-2:]
    mask_image = mask.reshape(h, w, 1) * color.reshape(1, 1, -1)
    ax.imshow(mask_image)
```

```
ann_frame_idx = 0  # the frame index we interact with
ann_obj_id = 1  # give a unique id to each object we interact with (it can be any integers)

# Let's add a positive click at (x, y) = (210, 350) to get started
points = np.array([[210, 350]], dtype=np.float32)
# for labels, `1` means positive click and `0` means negative click
labels = np.array([1], np.int32)
_, out_obj_ids, out_mask_logits = predictor.add_new_points_or_box(
    inference_state=inference_state,
    frame_idx=ann_frame_idx,
    obj_id=ann_obj_id,
    points=points,
    labels=labels,
)

# show the results on the current (interacted) frame
plt.figure(figsize=(9, 6))
plt.title(f"frame {ann_frame_idx}")
plt.imshow(Image.open(os.path.join(video_dir, frame_names[ann_frame_idx])))
show_points(points, labels, plt.gca())
show_mask_video((out_mask_logits[0] > 0.0).cpu().numpy(), plt.gca(), obj_id=out_obj_ids[0])
```

Step 2: Add a second click to refine the prediction

Hmm, it seems that although we wanted to segment the child on the left, the model predicts the mask for only the shorts – this can happen since there is ambiguity from a single click about what the target object should be. We can refine the mask on this frame via another positive click on the child's shirt.

Here we make a **second positive click** at (x, y) = (250, 220) with label `1` to expand the mask.

Note: we need to send **all the clicks and their labels** (i.e. not just the last click) when calling `add_new_points_or_box`.

```
ann_frame_idx = 0  # the frame index we interact with
ann_obj_id = 1  # give a unique id to each object we interact with (it can be any integers)

# Let's add a 2nd positive click at (x, y) = (250, 220) to refine the mask
# sending all clicks (and their labels) to `add_new_points_or_box`
points = np.array([[210, 350], [250, 220]], dtype=np.float32)
# for labels, `1` means positive click and `0` means negative click
labels = np.array([1, 1], np.int32)
_, out_obj_ids, out_mask_logits = predictor.add_new_points_or_box(
    inference_state=inference_state,
    frame_idx=ann_frame_idx,
    obj_id=ann_obj_id,
    points=points,
    labels=labels,
)

# show the results on the current (interacted) frame
plt.figure(figsize=(9, 6))
plt.title(f"frame {ann_frame_idx}")
plt.imshow(Image.open(os.path.join(video_dir, frame_names[ann_frame_idx])))
show_points(points, labels, plt.gca())
show_mask_video((out_mask_logits[0] > 0.0).cpu().numpy(), plt.gca(), obj_id=out_obj_ids[0])
```

With this 2nd refinement click, now we get a segmentation mask of the entire child on frame 0.

Step 3: Propagate the prompts to get the masklet across the video

To get the masklet throughout the entire video, we propagate the prompts using the `propagate_in_video` API.

```
# run propagation throughout the video and collect the results in a dict
video_segments = {}  # video_segments contains the per-frame segmentation results
for out_frame_idx, out_obj_ids, out_mask_logits in predictor.propagate_in_video(inference_state):
    video_segments[out_frame_idx] = {
        out_obj_id: (out_mask_logits[i] > 0.0).cpu().numpy()
        for i, out_obj_id in enumerate(out_obj_ids)
    }

# render the segmentation results every few frames
vis_frame_stride = 30
plt.close("all")
for out_frame_idx in range(0, len(frame_names), vis_frame_stride):
    plt.figure(figsize=(6, 4))
    plt.title(f"frame {out_frame_idx}")
    plt.imshow(Image.open(os.path.join(video_dir, frame_names[out_frame_idx])))
    for out_obj_id, out_mask in video_segments[out_frame_idx].items():
        show_mask_video(out_mask, plt.gca(), obj_id=out_obj_id)
```

```
propagate in video: 100%|##########| 200/200 [03:01<00:00,  1.10it/s]
```

Step 4: Add new prompts to further refine the masklet

It appears that in the output masklet above, there are some small imperfections in boundary details on frame 150.

With SAM 2 we can fix the model predictions interactively. We can add a **negative click** at (x, y) = (82, 415) on this frame with label `0` to refine the masklet. Here we call the `add_new_points_or_box` API with a different `frame_idx` argument to indicate the frame index we want to refine.

```
ann_frame_idx = 150  # further refine some details on this frame
ann_obj_id = 1  # give a unique id to the object we interact with (it can be any integers)

# show the segment before further refinement
plt.figure(figsize=(9, 6))
plt.title(f"frame {ann_frame_idx} -- before refinement")
plt.imshow(Image.open(os.path.join(video_dir, frame_names[ann_frame_idx])))
show_mask_video(video_segments[ann_frame_idx][ann_obj_id], plt.gca(), obj_id=ann_obj_id)

# Let's add a negative click on this frame at (x, y) = (82, 415) to refine the segment
points = np.array([[82, 410]], dtype=np.float32)
# for labels, `1` means positive click and `0` means negative click
labels = np.array([0], np.int32)
_, _, out_mask_logits = predictor.add_new_points_or_box(
    inference_state=inference_state,
    frame_idx=ann_frame_idx,
    obj_id=ann_obj_id,
    points=points,
    labels=labels,
)

# show the segment after the further refinement
plt.figure(figsize=(9, 6))
plt.title(f"frame {ann_frame_idx} -- after refinement")
plt.imshow(Image.open(os.path.join(video_dir, frame_names[ann_frame_idx])))
show_points(points, labels, plt.gca())
show_mask_video((out_mask_logits > 0.0).cpu().numpy(), plt.gca(), obj_id=ann_obj_id)
```

Step 5: Propagate the prompts (again) to get the masklet across the video

Let's get an updated masklet for the entire video. Here we call `propagate_in_video` again to propagate all the prompts after adding the new refinement click above.

```
# run propagation throughout the video and collect the results in a dict
video_segments = {}  # video_segments contains the per-frame segmentation results
for out_frame_idx, out_obj_ids, out_mask_logits in predictor.propagate_in_video(inference_state):
    video_segments[out_frame_idx] = {
        out_obj_id: (out_mask_logits[i] > 0.0).cpu().numpy()
        for i, out_obj_id in enumerate(out_obj_ids)
    }

# render the segmentation results every few frames
vis_frame_stride = 30
plt.close("all")
for out_frame_idx in range(0, len(frame_names), vis_frame_stride):
    plt.figure(figsize=(6, 4))
```

```
    plt.title(f"frame {out_frame_idx}")
    plt.imshow(Image.open(os.path.join(video_dir, frame_names[out_frame_idx])))
    for out_obj_id, out_mask in video_segments[out_frame_idx].items():
        show_mask_video(out_mask, plt.gca(), obj_id=out_obj_id)
```

```
propagate in video: 100%|##########| 200/200 [02:58<00:00,  1.12it/s]
```

18.17. Homework (Optional)

1. **Point vs. box precision.** Segment the same object first with one foreground point, then with a bounding box. Compare the two resulting masks visually and in terms of IoU against a hand-drawn ground truth.
2. **Iterative refinement.** Starting from one foreground point, add one background point at a time to remove unwanted regions. Record IoU after each click. Plot IoU vs. number of clicks to characterise the refinement curve.
3. **Benchmark on SA-V.** Pick 10 clips from the SA-V validation set, segment one object per clip with a single first-frame box prompt, and report the mean IoU on the last frame of each clip.
4. **Model-size comparison.** Run inference with `sam2_b.pt` and `sam2_l.pt` on the same image + prompt. Report latency and mask IoU against a manually drawn ground truth. When does the extra compute pay off?
5. **Prompt robustness.** For a chosen object, nudge the point prompt by up to 20 pixels in each direction and record how much the output mask changes. Use this to quantify SAM 2's prompt sensitivity.
6. **Video drift study.** Track one object across a 500-frame clip with no refinement prompts, then with a refinement prompt every 100 frames. Compare drift as measured by mean IoU per frame against an oracle mask.

Chapter 19

Sentiment Analysis

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1dMGeC66RFPOvPRsYzEEEvJfArv8MQ3yg?usp=sharing

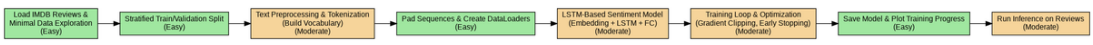
Sentiment analysis is a branch of **natural language processing (NLP)** that focuses on identifying and categorizing opinions expressed in text to determine whether the overall sentiment is `positive`, `negative`, or `neutral`. In this notebook, we use an **LSTM-based** model implemented in PyTorch to predict whether a movie review expresses a positive sentiment, enabling us to automatically assess the tone of customer feedback and opinions.

19.1. Necessary Imports

```
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.nn.functional as F
import matplotlib.pyplot as plt
import string
import re
import nltk
import time
import copy
import seaborn as sns
from IPython.display import Image as IMG
from tqdm import tqdm
from nltk.corpus import stopwords
from torchsummary import summary
from collections import Counter
from torch.utils.data import TensorDataset, DataLoader
from sklearn.model_selection import train_test_split

nltk.download('stopwords')
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```



19.2. Data

We will be using the **IMDB [5] movie reviews dataset**, a widely-used collection of movie reviews labeled as positive or negative. This dataset provides a rich resource for training and evaluating our sentiment analysis model, enabling us to accurately determine whether a given movie review conveys a positive sentiment.

```
%%bash
export PYTHONWARNINGS="ignore"
if [ -f "imdb.csv" ]; then
    echo "imdb.csv already exists. Skipping download."
else
    echo "imdb.csv does not exist. Downloading..."
    gdown --id 19RvCG8Ipeh4f1CFCrXfnCtyubTMOl5ab -O imdb.csv --no-check-certificate --quiet
fi
```

```
imdb.csv does not exist. Downloading...
```

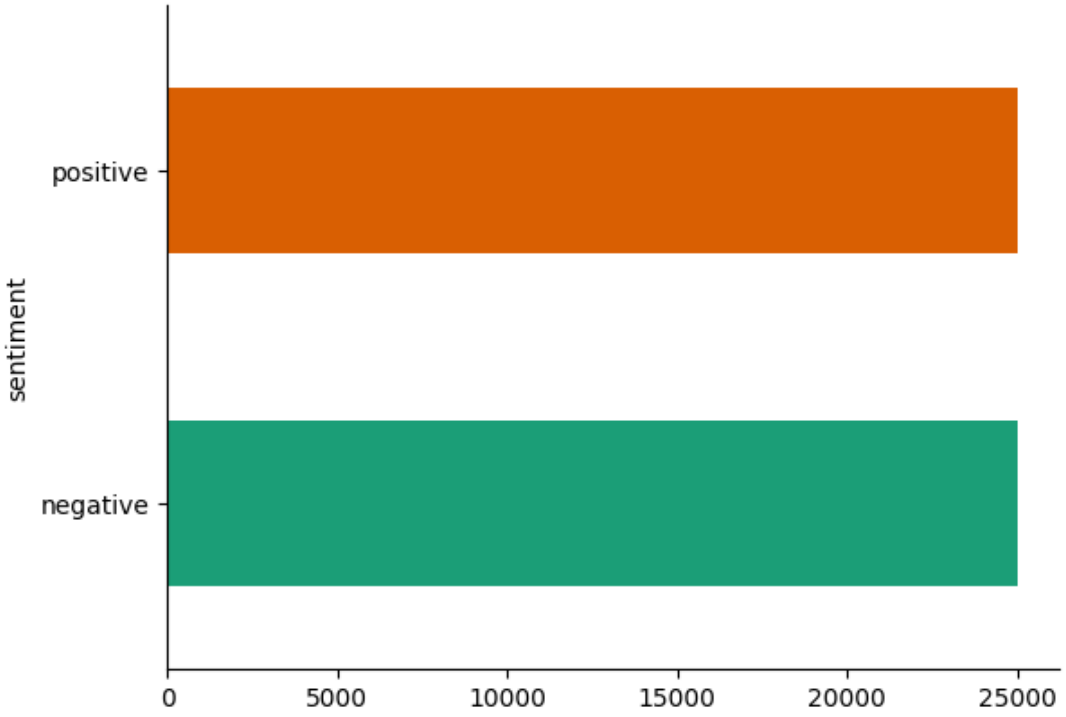
```
base_csv = '/content/imdb.csv'
df = pd.read_csv(base_csv)
df.head()
```

```
review sentiment
0  One of the other reviewers has mentioned that ...  positive
1  A wonderful little production. <br /><br />The...  positive
2  I thought this was a wonderful way to spend ti...  positive
3  Basically there's a family where a little boy ...  negative
4  Petter Mattei's "Love in the Time of Money" is...  positive
```

19.3. Minimal Data Exploration

Let's check data imbalance.

```
df.groupby('sentiment').size().plot(kind='barh', color=sns.palettes.mpl_palette('Dark2'))
plt.gca().spines[['top', 'right',]].set_visible(False)
```




```
for i in range(5):
    print(f"Review {i+1}:\n{df['review'][i]}")
    print(f"Sentiment: {df['sentiment'][i]}\n")
```

```
Review 1:
One of the other reviewers has mentioned that after watching just 1 Oz episode
you'll be hooked. They are right, as this is exactly what happened with me.<br
/><br />The first thin ... [1581 chars truncated]
Sentiment: positive

Review 2:
A wonderful little production. <br /><br />The filming technique is very
unassuming- very old-time-BBC fashion and gives a comforting, and sometimes
discomforting, sense of realism ... [818 chars truncated]
Sentiment: positive

Review 3:
I thought this was a wonderful way to spend time on a too hot summer weekend,
sitting in the air conditioned theater and watching a light-hearted comedy. The
plot is simplistic, bu ... [746 chars truncated]
Sentiment: positive

Review 4:
Basically there's a family where a little boy (Jake) thinks there's a zombie in
his closet & his parents are fighting all the time.<br /><br />This movie is
slower than a soap oper ... [568 chars truncated]
Sentiment: negative
```

```
Review 5:
Petter Mattei's "Love in the Time of Money" is a visually stunning film to
watch. Mr. Mattei offers us a vivid portrait about human relations. This is a
movie that seems to be tell ... [1137 chars truncated]
Sentiment: positive
```

19.3.1. Spilitting the Data into Train and Validation Sets

Stratified splitting is a method to ensure that each subset of the dataset (e.g., training, validation) maintains the same proportion of classes as the overall dataset. This is particularly useful when dealing with imbalanced data, as it helps prevent bias in model training and evaluation by preserving the class distribution in each split.

```
X, y = df['review'].values, df['sentiment'].values

x_train, x_valid, y_train, y_valid = train_test_split(X, y , stratify = y)
print(f'shape of train data is {x_train.shape}')
print(f'shape of valid data is {x_valid.shape}')
```

```
shape of train data is (37500,)
shape of valid data is (12500,)
```

19.3.2. Check Data Imbalance after Splitting

```
train_counts = pd.Series(y_train).value_counts()
valid_counts = pd.Series(y_valid).value_counts()

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
train_counts.plot(kind='barh', color=sns.palettes.mpl_palette('Dark2'))
plt.title('Training Set Sentiment Distribution')
plt.gca().spines[['top', 'right']].set_visible(False)

plt.subplot(1, 2, 2)
valid_counts.plot(kind='barh', color=sns.palettes.mpl_palette('Dark2'))
plt.title('Validation Set Sentiment Distribution')
plt.gca().spines[['top', 'right']].set_visible(False)

plt.tight_layout()
plt.show()

print("Training Set Sentiment Counts:\n", train_counts)
print("\nValidation Set Sentiment Counts:\n", valid_counts)
```




```
Training Set Sentiment Counts:
 positive    18750
negative    18750
Name: count, dtype: int64

Validation Set Sentiment Counts:
 positive    6250
negative    6250
Name: count, dtype: int64
```

19.3.3. Tokenization and Text Preprocessing

Tokenization is a critical step in NLP that involves splitting raw text into smaller, meaningful units (`tokens`) such as words. This process allows us to convert text into numerical representations that can be fed into machine learning models. Moreover, thorough text preprocessing, including cleaning (e.g., removing HTML tags, punctuation, digits, and stopwords), helps reduce noise and improves model performance.

```python
def preprocess_string(s):
    """
    Cleans the input string by removing HTML <br> tags, punctuation, digits,
    and extra whitespace.
    """
    # Remove <br> tags
    s = re.sub(r'<br\s*/?>', ' ', s)
    # Remove all non-word characters (except whitespace)
    s = re.sub(r"[^\w\s]", '', s)
    # Remove digits
    s = re.sub(r"\d", '', s)
    # Replace multiple whitespaces with a single space and strip leading/trailing spaces
    s = re.sub(r"\s+", ' ', s).strip()
    return s

def tokenize(x_train, y_train, x_val, y_val):
    """
    Tokenizes and encodes the training and validation texts.
```

```
    This function cleans the input text, removes stopwords, builds a vocabulary
    from the 1024 most common words in the training set, and converts sentences
    into lists of token IDs. It also encodes labels (assuming 'positive' and 'negative').

    Args:
        x_train (list of str): List of training sentences.
        y_train (list of str): List of training labels.
        x_val (list of str): List of validation sentences.
        y_val (list of str): List of validation labels.

    Returns:
        tokenized_train (np.ndarray): Array of token lists for training.
        encoded_train (np.ndarray): Array of encoded training labels.
        tokenized_val (np.ndarray): Array of token lists for validation.
        encoded_val (np.ndarray): Array of encoded validation labels.
        integer_dict (dict): Mapping from words to unique integers.
    """
    stop_words = set(stopwords.words('english'))
    word_list = []

    # Build vocabulary from the training set
    for sent in x_train:
        clean_sent = preprocess_string(sent.lower())
        for word in clean_sent.split():
            if word not in stop_words and word:
                word_list.append(word)

    corpus = Counter(word_list)
    corpus_common = sorted(corpus, key=corpus.get, reverse=True)
    integer_dict = {w: i+1 for i, w in enumerate(corpus_common)}

    # Function to convert a sentence to a list of token IDs
    def sentence_to_tokens(sent):
        clean_sent = preprocess_string(sent.lower())
        return [integer_dict[word] for word in clean_sent.split() if word in integer_dict]

    tokenized_train = [sentence_to_tokens(sent) for sent in x_train]
    tokenized_val = [sentence_to_tokens(sent) for sent in x_val]

    encoded_train = [1 if label == 'positive' else 0 for label in y_train]
    encoded_val = [1 if label == 'positive' else 0 for label in y_val]

    return (np.array(tokenized_train, dtype=object),
            np.array(encoded_train),
            np.array(tokenized_val, dtype=object),
            np.array(encoded_val),
            integer_dict)
```

```
x_train, y_train, x_valid, y_valid, vocab = tokenize(x_train, y_train, x_valid, y_valid)
```

```
print(f'Length of vocabulary is {len(vocab)}')
```

```
Length of vocabulary is 138742
```

```
vocab = dict(sorted(vocab.items(), key=lambda item: item[1]))
print(vocab)
```

```
{'movie': 1, 'film': 2, 'one': 3, 'like': 4, 'good': 5, 'even': 6, 'would': 7,
'time': 8, 'really': 9, 'see': 10, 'story': 11, 'well': 12, 'much': 13, 'get':
14, 'great': 15, 'also ... [2765893 chars truncated]
```

```
reverse_vocab = {idx: word for word, idx in vocab.items()}
print(reverse_vocab)
```

```
{1: 'movie', 2: 'film', 3: 'one', 4: 'like', 5: 'good', 6: 'even', 7: 'would',
8: 'time', 9: 'really', 10: 'see', 11: 'story', 12: 'well', 13: 'much', 14:
'get', 15: 'great', 16: ' ... [2765893 chars truncated]
```

19.3.4. Visualizing our Inputs as Text and Tokens

```
random_index = np.random.randint(0, len(x_train))

review_tokens = x_train[random_index]
review_text = " ".join([reverse_vocab[token] for token in review_tokens if token in reverse_vocab])

# Print the review text and its tokens
print("Review Text:")
print(review_text)
print("Number of words in the text:", len(review_text.split()))

print("\nReview Tokens:")
print(review_tokens)
print("Number of tokens:", len(review_tokens))

# Retrieve the sentiment label from the DataFrame
sentiment = df['sentiment'][random_index]

print("\nSentiment:")
print(sentiment)
print(f"{1 if sentiment == 'positive' else 0}")
```

```
Review Text:
first pa kettle flicks marjorie main steals show anything funny see ida moore
emily daffy old lady train god always old desk set alfred hitchcock presents new
house also costar hou ... [471 chars truncated]
Number of words in the text: 101

Review Tokens:
[19, 7576, 20237, 1315, 12950, 174, 2053, 41, 132, 64, 10, 9075, 1842, 3211,
7340, 76, 711, 937, 463, 110, 76, 7888, 176, 3995, 1673, 2131, 72, 210, 16,
4754, 210, 585, 9, 470, 142 ... [345 chars truncated]
Number of tokens: 101

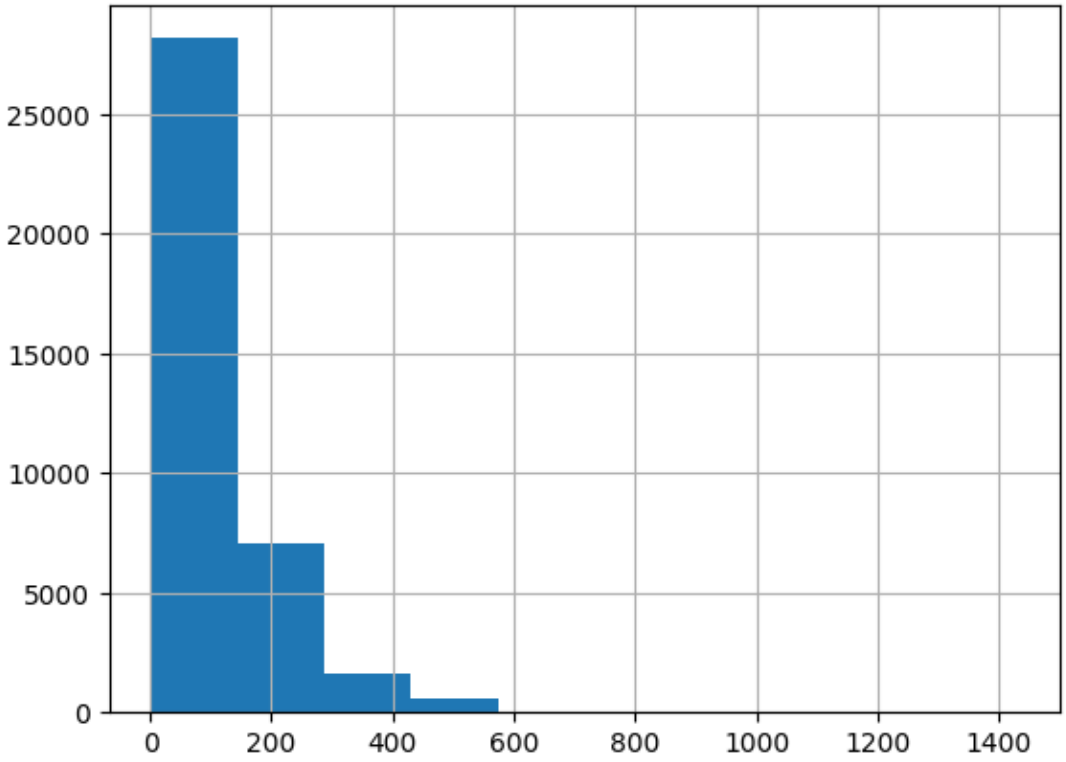
Sentiment:
positive
1
```

19.4. Visualizing Tokenized Data

```
rev_len = [len(i) for i in x_train]
```

```
pd.Series(rev_len).hist()
plt.show()

pd.Series(rev_len).describe()
```




```
count    37500.000000
mean       119.045947
std         89.690008
min          3.000000
25%         64.000000
50%         89.000000
75%        145.000000
max       1428.000000
dtype: float64
```

19.4.1. Adding Padding

We will pad each of the sequences to a trimmed maximum length in the training dataset.

```
def padding_(sentences, seq_len):
    """
    Pads or truncates a list of tokenized sentences to a fixed sequence length.

    This function creates a 2D NumPy array where each row corresponds to a sentence
    padded with zeros (if shorter than seq_len) or truncated (if longer than seq_len).
    The padding is applied to the left, so that the last tokens of each sentence are
    aligned at the right of the array.

    Args:
        sentences (list of list of int): List of tokenized sentences (each sentence is a list of token IDs).
```

```
        seq_len (int): The fixed sequence length to pad or truncate each sentence to.

    Returns:
        np.ndarray: A 2D array of shape (number of sentences, seq_len) containing the padded or truncated
   sentences.
    """
    features = np.zeros((len(sentences), seq_len), dtype=int)
    for ii, review in enumerate(sentences):
        if len(review) != 0:
            features[ii, -len(review):] = np.array(review)[:seq_len]
    return features

# Trim reviews that are longer than 200 tokens
# Pad tokens with zeros that are shorter than 200 tokens
x_train_pad = padding_(x_train, 200)
x_valid_pad = padding_(x_valid, 200)
```

```
print("Shape of the training data:", x_train_pad.shape)
print("Shape of the validation data:", x_valid_pad.shape)
```

```
Shape of the training data: (37500, 200)
Shape of the validation data: (12500, 200)
```

```
random_index = np.random.randint(0, len(x_train_pad))
review_tokens = x_train_pad[random_index]

print(review_tokens)
```

```
[    0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0   162
   959    22     4   959   249   142   107     2    54   325    53   635
    47   513    42   101  5074  7970    36   819  3636    38   154   270
   630 12598  2904  7970     2    66    82     5   728   528   419    50
    40 51357     9     5   106    47    44   325    10  7031   665  1180
  1754     2   730    54  8900  1586  1549     2    79  1987  1354    13
    44  4249   194  3179   310  1733 13429  2990   291   209  3033     9
     5   419    50   117   747  8010  3510   647    78 27237  1787    62
  3179     2 69843     1  6654   352   363  4388   199   866   610 27237
  1787 27237  6693   174   667   467  9048   602    11     8   467   363
  3179   527  3636   287   179   728   419    50     6 51357  1116    85
   265  1586  1549   211    99  4924   376   497]
```

For our model, `left` (or beginning) padding is generally preferred. Since the network uses the output of the last time-step as the representation for sentiment prediction, left padding ensures that the final token corresponds to actual data rather than padding.

With right padding, shorter sequences would have their actual words at the beginning and padding tokens at the end, causing the network to output a final hidden state that is influenced by these non-informative padding tokens.

19.4.2. Batching and Creating a DataLoader

```
# Create Tensor datasets
train_data = TensorDataset(torch.from_numpy(x_train_pad.astype(np.int64)), torch.from_numpy(y_train))
valid_data = TensorDataset(torch.from_numpy(x_valid_pad.astype(np.int64)), torch.from_numpy(y_valid))

batch_size = 64
train_loader = DataLoader(train_data, shuffle=True, batch_size=batch_size)
valid_loader = DataLoader(valid_data, shuffle=True, batch_size=batch_size)
```

```
dataiter = iter(train_loader)
sample_x, sample_y = next(dataiter)

print('Sample input size: ', sample_x.size())
print('Sample input: \n', sample_x)
print('Sample output: \n', sample_y)
```

```
Sample input size:  torch.Size([64, 200])
Sample input:
 tensor([[    0,     0,     0,  ...,   452,   957,   114],
        [    0,     0,     0,  ...,  7880,  1064,   760],
        [    2,  2010,    73,  ...,  1820, 20474,    17],
        ...,
        [ 3610,    49,   826,  ..., 12960,    30,   381],
        [12455, 35110,  2811,  ...,  4212,  1847,  8565],
        [    0,     0,     0,  ...,   390, 10359,  1274]])
Sample output:
 tensor([0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0,
        0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1,
        1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 1])
```

19.5. Padding, truncation, and bidirectional LSTMs

Text sequences vary in length but GPU tensors do not. Two tricks make batching possible.

Padding adds a special token to the end (or beginning) of shorter sequences so every sequence in a batch ends up the same length. The model must be told to ignore padded positions, in PyTorch this is what `pack_padded_sequence` does for RNNs, or `attention_mask` for transformers. Padding in the wrong place (for example, left-padding when the model expects right-padding) silently reduces accuracy because the last hidden state then carries no useful information.

Truncation clips very long sequences to a fixed maximum. Two common policies: take the first `N` tokens (cheap, information-preserving for early-decisive inputs like news leads) or the last `N` tokens (better for reviews where the verdict often comes at the end). The right choice depends on where the signal lives in your corpus.

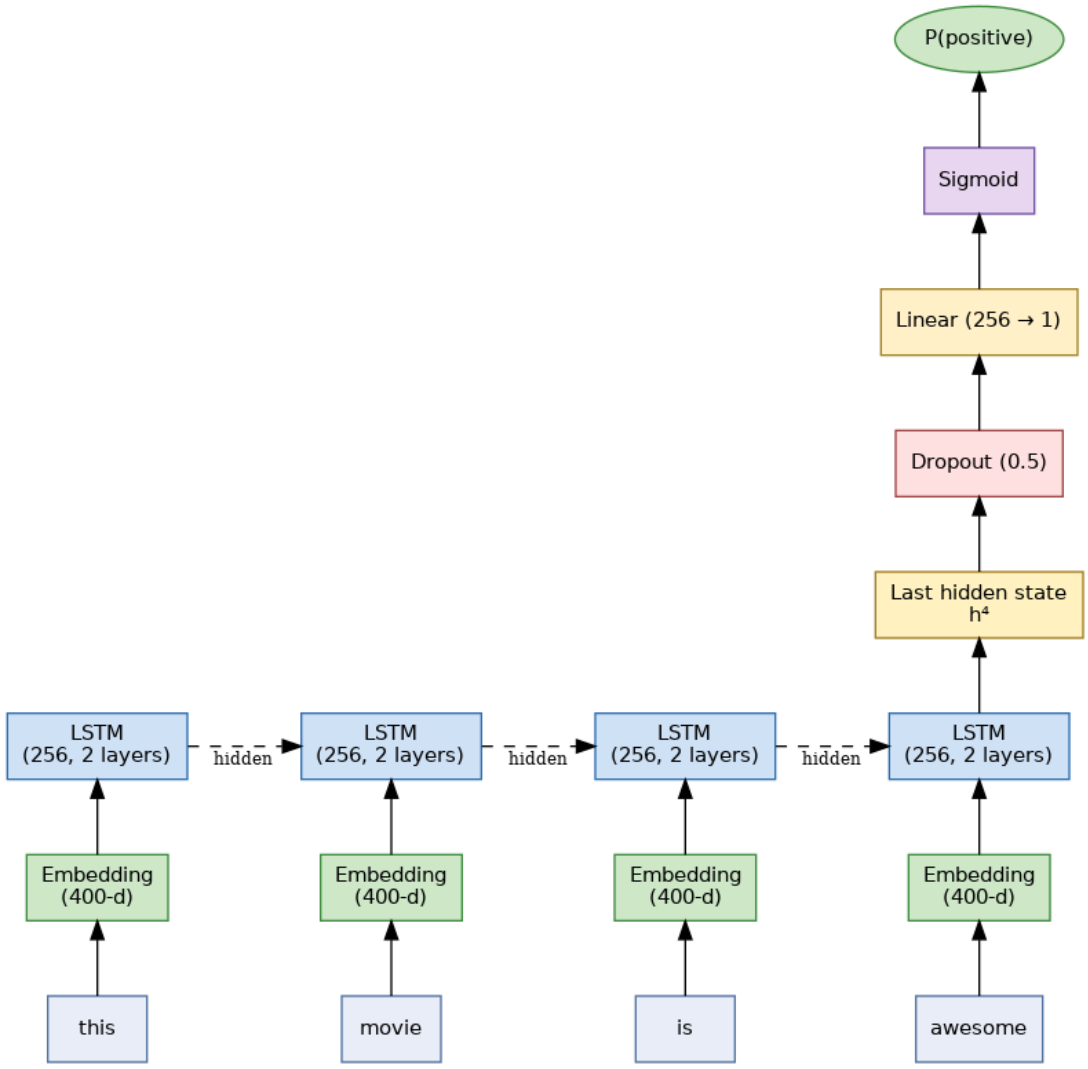
A *bidirectional* LSTM is a pair of LSTMs, one running left-to-right and one right-to-left, with their hidden states concatenated. Each position then has access to context from both directions, which is critical for sentiment words whose polarity flips based on later text (`not great`, `great fun, if you ignore the plot`). For sentiment classification specifically, bidirectionality typically adds two to four accuracy points at a modest compute cost.

19.6. LSTM-Based Model

We add an `embedding layer` because one-hot encoding a large number of classes is computationally inefficient, even when the vocabulary is relatively small. Instead of one-hot encoding, the embedding layer serves as a **lookup table** that maps each word to a dense, lower-dimensional vector.

Although you could pre-train this layer using methods like `Word2Vec` and load those embeddings, it is often sufficient to initialize a new embedding layer for dimensionality reduction and allow the network to learn the optimal representations during training.

We are going to create something **similar** to this:




```python
class SentimentLSTM(nn.Module):
    def __init__(self, vocab_size, output_size, embedding_dim, hidden_dim, n_layers):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, n_layers, batch_first=True, dropout=0.5)
        self.dropout = nn.Dropout(0.5)
        self.fc = nn.Linear(hidden_dim, output_size)
```

```
    def forward(self, x, hidden):
        batch_size = x.size(0)
        embeds = self.embedding(x)
        lstm_out, hidden = self.lstm(embeds, hidden)
        lstm_out = lstm_out[:, -1, :]
        out = self.dropout(lstm_out)
        out = self.fc(out)
        return out, hidden

    def init_hidden(self, batch_size, device):
        weight = next(self.parameters()).data
        hidden = (
            weight.new(self.n_layers, batch_size, self.hidden_dim).zero_().to(device),
            weight.new(self.n_layers, batch_size, self.hidden_dim).zero_().to(device)
        )
        return hidden
```

Hyperparameters

The main hyperparameters in the number of LSTM layer num, embedding dimension and the hidden dimension:

```
no_layers = 2
vocab_size = len(vocab) + 1
embedding_dim = 400
output_dim = 1
hidden_dim = 256

model = SentimentLSTM(vocab_size, output_dim, embedding_dim, hidden_dim, no_layers)
model.to(device)
```

```
SentimentLSTM(
  (embedding): Embedding(138743, 400)
  (lstm): LSTM(400, 256, num_layers=2, batch_first=True)
  (dropout): Dropout(p=0.3, inplace=False)
  (fc): Linear(in_features=256, out_features=1, bias=True)
  (sig): Sigmoid()
)
```

19.7. Training our Neural Network

We use `BCEWithLogitsLoss` as our loss function because our model outputs raw logits rather than probabilities. This loss function applies the sigmoid activation internally in a numerically stable manner.

We employ the `Adam` optimizer for efficient parameter updates along with a learning rate scheduler that gradually decreases the learning rate. Our custom acc function is designed to assess whether the predicted sentiment matches the target label.

```
criterion = nn.BCEWithLogitsLoss()
optimizer = torch.optim.AdamW(model.parameters(), lr=0.001, weight_decay=1e-5)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', patience=2, factor=0.5)

def acc(pred, label):
    pred = torch.round(pred.squeeze()).int()
    label = label.squeeze().int()
    return torch.sum(pred == label).item()
```

This **training loop** trains and validates the model over multiple epochs. During each `epoch`, the loop:

- **Trains the model** on the training data, performing forward and backward passes, and applies gradient clipping.
- **Evaluates the model** on the validation set to compute average loss and accuracy.
- **Steps the learning rate scheduler** based on validation performance.
- **Tracks training and validation metrics** and saves the model state whenever the validation loss decreases.
- **Implements early stopping** if no improvement in validation loss is observed for a specified number of epochs.

After training, the `best model weights` (with the lowest validation loss) are loaded for further evaluation.

```
clip = 5
epochs = 10
early_stop_patience = 3
valid_loss_min = np.Inf

epoch_tr_loss = []
epoch_vl_loss = []
epoch_tr_acc = []
epoch_vl_acc = []

start_time = time.time()

for epoch in range(epochs):
    print(f"Epoch {epoch+1}/{epochs}")
    print("-" * 50)

    # ---------------------------
    # Training Phase
    # ---------------------------
    model.train()
    train_losses = []
    train_correct = 0.0

    for inputs, labels in tqdm(train_loader, desc="Training", leave=False):
        current_batch_size = inputs.size(0)
        h = model.init_hidden(current_batch_size, device)
        inputs, labels = inputs.to(device), labels.to(device)
        h = tuple([each.detach() for each in h])

        optimizer.zero_grad()

        output, h = model(inputs, h)

        loss = criterion(output.squeeze(), labels.float())
        train_losses.append(loss.item())
        train_correct += acc(output, labels)

        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), clip)
        optimizer.step()

    scheduler.step(avg_val_loss)
    avg_train_loss = np.mean(train_losses)
    avg_train_acc = train_correct / len(train_loader.dataset)
```

```
    # ---------------------------
    # Validation Phase
    # ---------------------------
    model.eval()
    val_losses = []
    val_correct = 0.0

    for inputs, labels in tqdm(valid_loader, desc="Validation", leave=False):
        current_batch_size = inputs.size(0)
        val_h = model.init_hidden(current_batch_size, device)
        inputs, labels = inputs.to(device), labels.to(device)
        val_h = tuple([each.detach() for each in val_h])

        with torch.no_grad():
            output, val_h = model(inputs, val_h)
            loss = criterion(output.squeeze(), labels.float())
            val_losses.append(loss.item())
            val_correct += acc(output, labels)

    avg_val_loss = np.mean(val_losses)
    avg_val_acc = val_correct / len(valid_loader.dataset)

    # Store the epoch's metrics in history
    epoch_tr_loss.append(avg_train_loss)
    epoch_vl_loss.append(avg_val_loss)
    epoch_tr_acc.append(avg_train_acc)
    epoch_vl_acc.append(avg_val_acc)

    # Print Epoch Summary
    print(f"Train Loss: {avg_train_loss:.4f}, Validation Loss: {avg_val_loss:.4f}")
    print(f"Train Accuracy: {avg_train_acc*100:.2f}%, Validation Accuracy: {avg_val_acc*100:.2f}%")
    print(f"Current Learning Rate: {scheduler.get_last_lr()[0]:.6f}")

    # Check for improvement and save the best model if validation loss decreases
    if avg_val_loss <= valid_loss_min:
        best_model_wts = copy.deepcopy(model.state_dict())
        print(f"Validation loss decreased ({valid_loss_min:.6f} → {avg_val_loss:.6f}). Saving model...")
        valid_loss_min = avg_val_loss
        epochs_no_improve = 0
    else:
        epochs_no_improve += 1
        print("No improvement in validation loss.")

    print("=" * 50)

    # Early stopping check
    if epochs_no_improve >= early_stop_patience:
        print("Early stopping triggered.")
        break

total_time = time.time() - start_time
print(f"Training complete in {total_time // 60:.0f}m {total_time % 60:.0f}s")
print(f"Best Validation Loss: {valid_loss_min:.4f}")
model.load_state_dict(best_model_wts)
```

```
Epoch 1/10
--------------------------------------------------
```

```
Train Loss: 0.5510, Validation Loss: 0.5764
Train Accuracy: 89.73%, Validation Accuracy: 82.54%
Current Learning Rate: 0.001000
Validation loss decreased (inf -> 0.576392). Saving model...
==================================================
Epoch 2/10
--------------------------------------------------
```

```
Train Loss: 0.5581, Validation Loss: 0.5684
Train Accuracy: 87.93%, Validation Accuracy: 85.21%
Current Learning Rate: 0.001000
Validation loss decreased (0.576392 -> 0.568366). Saving model...
==================================================
Epoch 3/10
--------------------------------------------------
```

```
Train Loss: 0.5579, Validation Loss: 0.5680
Train Accuracy: 88.36%, Validation Accuracy: 85.45%
Current Learning Rate: 0.001000
Validation loss decreased (0.568366 -> 0.567957). Saving model...
==================================================
Epoch 4/10
--------------------------------------------------
```

```
Train Loss: 0.5468, Validation Loss: 0.5685
Train Accuracy: 90.57%, Validation Accuracy: 87.43%
Current Learning Rate: 0.000500
No improvement in validation loss.
==================================================
Epoch 5/10
--------------------------------------------------
```

```
Train Loss: 0.5395, Validation Loss: 0.5627
Train Accuracy: 92.28%, Validation Accuracy: 87.70%
Current Learning Rate: 0.000500
Validation loss decreased (0.567957 -> 0.562726). Saving model...
==================================================
Epoch 6/10
--------------------------------------------------
```

```
Train Loss: 0.5363, Validation Loss: 0.5630
Train Accuracy: 92.99%, Validation Accuracy: 87.31%
Current Learning Rate: 0.000500
No improvement in validation loss.
==================================================
Epoch 7/10
--------------------------------------------------
```

```
Train Loss: 0.5352, Validation Loss: 0.5666
Train Accuracy: 93.09%, Validation Accuracy: 85.42%
Current Learning Rate: 0.000250
No improvement in validation loss.
==================================================
Epoch 8/10
--------------------------------------------------
```

```
Train Loss: 0.5326, Validation Loss: 0.5647
Train Accuracy: 93.65%, Validation Accuracy: 87.59%
Current Learning Rate: 0.000250
No improvement in validation loss.
==================================================
Early stopping triggered.
Training complete in 7m 41s
Best Validation Loss: 0.5627
```

```
<All keys matched successfully>
```

19.7.1. Saving and Loading the Model

```
torch.save(best_model_wts, 'best_model.pt')

# Later, load the saved weights into your model
model.load_state_dict(torch.load('best_model.pt', weights_only = True))
```

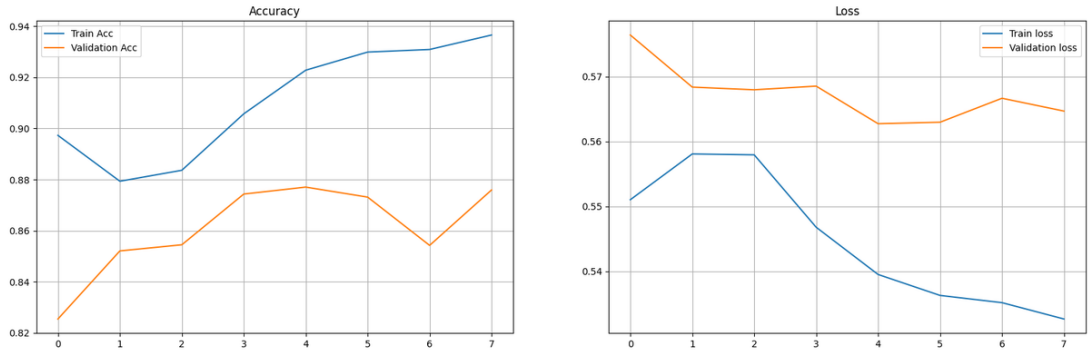
```
<All keys matched successfully>
```

19.8. Plotting our Results

```
fig = plt.figure(figsize = (20, 6))
plt.subplot(1, 2, 1)
plt.plot(epoch_tr_acc, label='Train Acc')
plt.plot(epoch_vl_acc, label='Validation Acc')
plt.title("Accuracy")
plt.legend()
plt.grid()

plt.subplot(1, 2, 2)
plt.plot(epoch_tr_loss, label='Train loss')
plt.plot(epoch_vl_loss, label='Validation loss')
plt.title("Loss")
plt.legend()
plt.grid()

plt.show()
```



19.9. Running Inference on our Model

```
def predict_text(text):
    """
    Predict the sentiment score for the given text using the trained model.

    The function preprocesses and tokenizes the text, pads the sequence to a fixed length,
    initializes the model's hidden state, and returns the predicted sentiment as a scalar.

    Args:
        text (str): The input text (e.g., a movie review).

    Returns:
        float: The predicted sentiment score.
    """
    word_seq = np.array([vocab[preprocess_string(word)] for word in text.split()
                         if preprocess_string(word) in vocab.keys()])
    word_seq = np.expand_dims(word_seq, axis=0)
    pad = torch.from_numpy(padding_(word_seq, 200))
    inputs = pad.to(device)
    batch_size = 1
    h = model.init_hidden(batch_size, device)
    h = tuple([each.data for each in h])
    output, h = model(inputs, h)
    return output.item()
```

19.9.1. Try an example from our data

```
index = 41149
print(df['review'][index])
print('=' * 70)
print(f'Actual sentiment is  : {df["sentiment"][index]}')
print('=' * 70)
pro = predict_text(df['review'][index])
status = "positive" if pro > 0.5 else "negative"
pro = (1 - pro) if status == "negative" else pro
print(f'Predicted sentiment is {status} with a confidence of {pro}')
```

```
The Karen Carpenter Story shows a little more about singer Karen Carpenter's
complex life. Though it fails in giving accurate facts, and details.<br /><br
/>Cynthia Gibb (portrays  ... [500 chars truncated]
======================================================================
```

```
Actual sentiment is  : positive
======================================================================
Predicted sentiment is positive with a confidence of 0.9999221563339233
```

```
index = 41150
print(df['review'][index])
print('=' * 70)
print(f'Actual sentiment is  : {df["sentiment"][index]}')
print('=' * 70)
pro = predict_text(df['review'][index])
status = "positive" if pro > 0.5 else "negative"
pro = (1 - pro) if status == "negative" else pro
print(f'Predicted sentiment is {status} with a confidence of {pro}')
```

```
It had all the clichés of movies of this type and no substance. The plot went
nowhere and at the end of the movie I felt like a sucker for watching it. The
production was good; how ... [498 chars truncated]
======================================================================
Actual sentiment is  : negative
======================================================================
Predicted sentiment is negative with a confidence of 0.9999806053419888
```

```
index = 41151
print(df['review'][index])
print('=' * 70)
print(f'Actual sentiment is  : {df["sentiment"][index]}')
print('=' * 70)
pro = predict_text(df['review'][index])
status = "positive" if pro > 0.5 else "negative"
pro = (1 - pro) if status == "negative" else pro
print(f'Predicted sentiment is {status} with a confidence of {pro}')
```

```
I am so happy and surprised that there is so much interest in this movie! Jack
Frost was my introduction into the films produced and distributed by A-pix
entertainment, and without ... [504 chars truncated]
======================================================================
Actual sentiment is  : negative
======================================================================
Predicted sentiment is negative with a confidence of 0.999901515278907
```

19.10. Testing our Model with Arbitrary Reviews

```
input = "It wasn't great."
print(input)
print('=' * 70)
pro = predict_text(input)
status = "positive" if pro > 0.5 else "negative"
pro = (1 - pro) if status == "negative" else pro
print(f'Predicted sentiment is {status} with a confidence of {pro}')
```

```
It wasn't great.
======================================================================
Predicted sentiment is negative with a confidence of 0.8965626657009125
```

```
input = "It was bad."
print(input)
print('=' * 70)
pro = predict_text(input)
status = "positive" if pro > 0.5 else "negative"
pro = (1 - pro) if status == "negative" else pro
print(f'Predicted sentiment is {status} with a confidence of {pro}')
```

```
It was bad.
======================================================================
Predicted sentiment is negative with a confidence of 0.9999084457813296
```

19.11. Homework (Optional)

1. **Bidirectional vs. unidirectional.** Train the model with `bidirectional=True` and compare final validation accuracy. Report both numbers and the increase in parameter count.
2. **Padding side.** Swap the padding so it is inserted at the beginning of each sequence instead of the end. Retrain and report the change in accuracy. Speculate on why the direction matters for an LSTM.
3. **Truncation length sweep.** Train variants that truncate at 100, 200 and 500 tokens. Plot accuracy and training time vs. truncation length and pick the best operating point.
4. **Embedding initialisation.** Replace the randomly initialised embedding with pretrained GloVe vectors (`glove.6B.300d`). Train the classifier on top and report the gain.
5. **Class imbalance robustness.** Subsample the training set so that only 20 percent of reviews are negative. Retrain and report per-class precision and recall. Propose and test one mitigation (class weights, oversampling, focal loss).
6. **Interpretability.** For three correctly classified and three misclassified test examples, show the tokens and their contribution to the output (e.g. via integrated gradients or a simple leave-one-out). Comment on whether the model is attending to sentiment-bearing words.

Chapter 20

Time Series Prediction

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1KT0h_mSWLy78nyvKIcKQnhhDnYscu4Yh?usp=sharing

Time series prediction involves forecasting future values based on historical data points collected at consistent time intervals. This is important for applications such as financial market analysis, weather forecasting, and demand prediction.

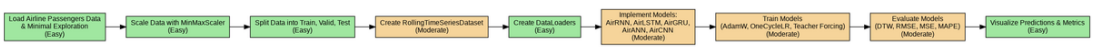
Recurrent Neural Networks (`RNNs`) are well-suited for time series forecasting because they maintain a hidden state that captures temporal dependencies, allowing them to learn patterns and trends from sequential data more effectively than traditional feedforward networks.

20.1. Necessary Imports

```
!pip install --quiet fastdtw
```

```
import torch
import copy
import math
import pandas as pd
import numpy as np
import torch.optim as optim
import torch.utils.data as data
import matplotlib.pyplot as plt
from IPython.display import Image as IMG
from torch import nn, Tensor
from torch.utils.data import Dataset, DataLoader
from scipy.spatial.distance import euclidean
from fastdtw import fastdtw
from sklearn.metrics import mean_squared_error, mean_absolute_error, mean_absolute_percentage_error
from sklearn.preprocessing import MinMaxScaler
from torch.optim.lr_scheduler import ReduceLROnPlateau

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```



20.2. Data

We will be using the **Airline Passengers** dataset, a widely-used time series dataset that contains monthly totals of international airline passengers from 1949 to 1960. This dataset provides a rich resource for

training and evaluating our time series forecasting model, enabling us to capture seasonal trends and long-term patterns in air travel demand.

```
%%bash
export PYTHONWARNINGS="ignore"
if [ -f "airline-passengers.csv" ]; then
    echo "airline-passengers.csv already exists. Skipping download."
else
    echo "airline-passengers.csv does not exist. Downloading..."
    wget https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv
   --no-check-certificate --quiet
fi
```

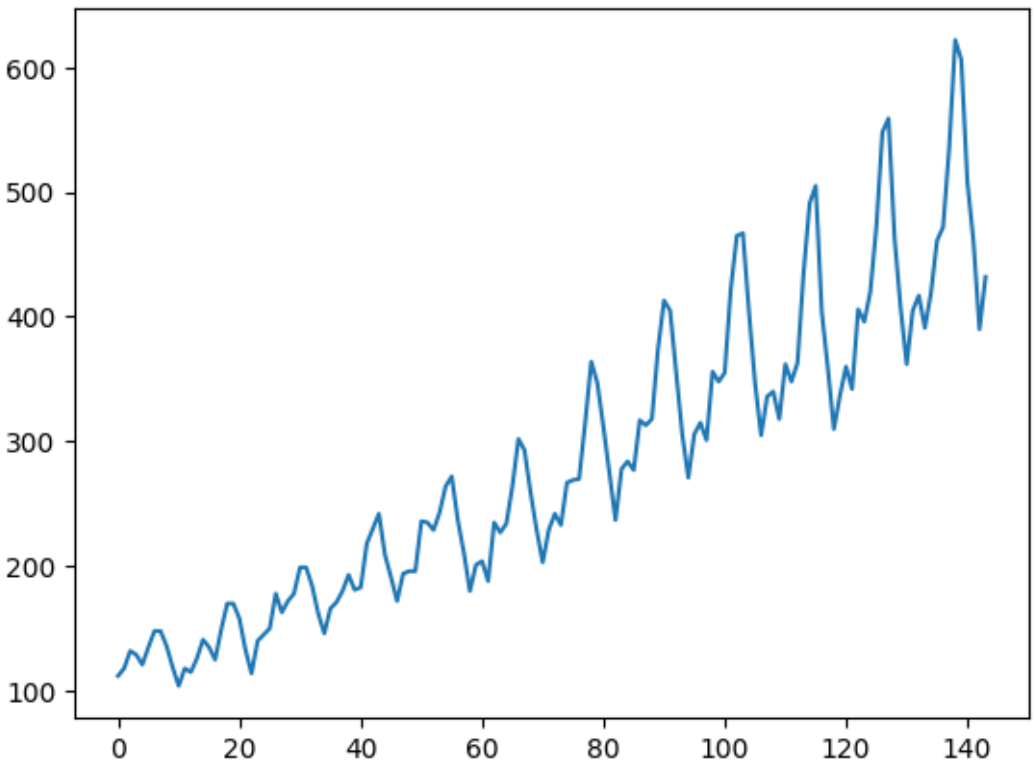
```
airline-passengers.csv does not exist. Downloading...
```

```
base_csv = '/content/airline-passengers.csv'
df = pd.read_csv(base_csv)
df.head()
```

```
Month  Passengers
0  1949-01         112
1  1949-02         118
2  1949-03         132
3  1949-04         129
4  1949-05         121
```

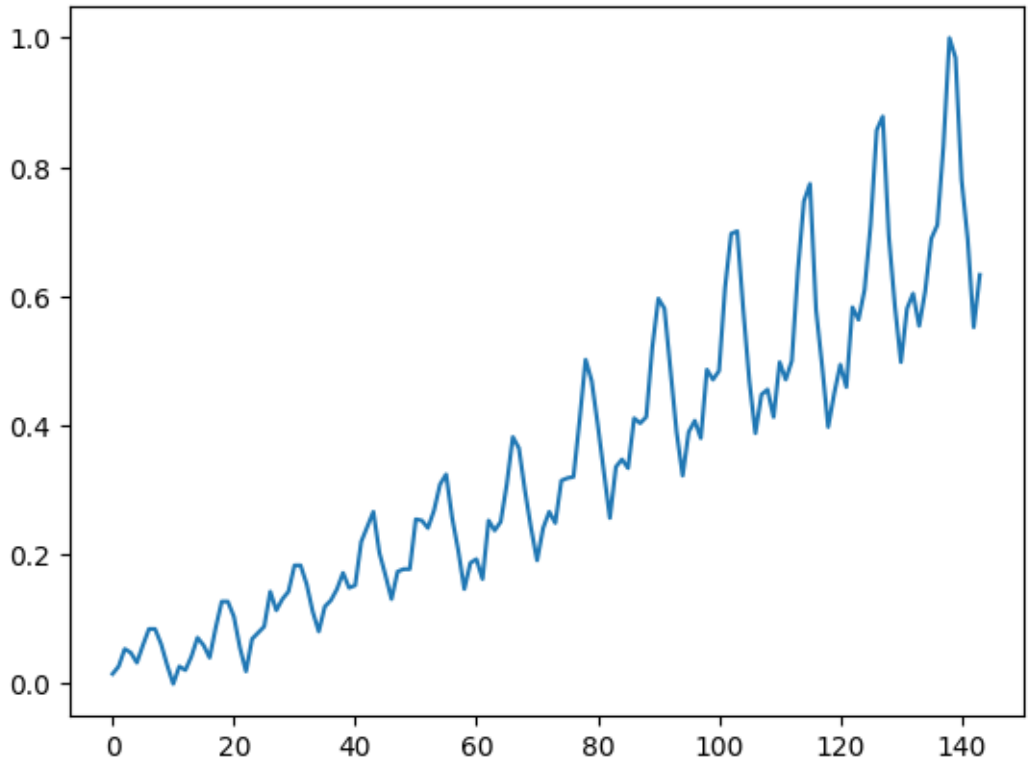
20.2.1. Minimal Data Exploration

```
timeseries = df[["Passengers"]].values.astype('float32')
plt.plot(timeseries)
plt.show()
```



```
scaler = MinMaxScaler(feature_range=(0, 1))
timeseries = scaler.fit_transform(timeseries.reshape(-1, 1)).astype('float32')
```

```
plt.plot(timeseries)
plt.show()
```



20.2.2. Spilitting the Data into Train and Validation Sets

We partition our aggregated sentence pair corpus into three subsets: **60% for training**, which the model will learn from, **20% for validation**, used to fine-tune hyperparameters and monitor performance, and **20% for testing**, reserved for evaluating the final model's generalization ability.

```
train_size = int(len(timeseries) * 0.6)
valid_size = int(len(timeseries) * 0.2)
test_size = len(timeseries) - train_size - valid_size

train, valid, test = (
    timeseries[:train_size],
    timeseries[train_size : train_size + valid_size],
    timeseries[train_size + valid_size :],
)
```

Lookback (`lookback`) defines how many past time steps are used as input, while **Horizon (`horizon`)** determines how many future steps the model predicts.

```
lookback = 6
horizon = 2
```

20.3. Train, validation and test splits for time series

Time-series data is not exchangeable: tomorrow depends on today and today depends on yesterday. Randomly shuffling rows before splitting leaks the future into the training set and produces accuracy numbers that look good in the notebook but collapse in production.

The correct split is *chronological*. Take the earliest 60 percent of observations for training, the next 20 for validation, and the final 20 for test. Never shuffle before the split. Within a split, the sliding-window dataset can shuffle the windows, that preserves temporal integrity because each window is self-contained, but the split boundary must always be a point in time.

For models with a large lookback window, the first `L` predictions of the validation and test splits are *contaminated*: their input window includes training observations. Depending on how strict you want to be, either discard those predictions or account for them when reporting metrics. Rolling-origin evaluation (train on `[0, T]`, evaluate on `T+1`. Train on `[0, T+1]`, evaluate on `T+2`; . . .) is the gold standard used in forecasting competitions and gives a much more honest picture than a single fixed split.

20.4. Creating a Dataset and a Dataloader

To efficiently feed our data into the model, we use **PyTorch's Dataset and DataLoader classes**. This class handles the lookback mechanism that creates sequences of past observations as inputs and their corresponding future values as targets.

Using `Dataset` allows us to structure our time series data in a format that is compatible with PyTorch models, ensuring that each sample maintains a fixed sequence length.

```
class RollingTimeSeriesDataset(Dataset):
    def __init__(self, data, lookback, horizon=5, stride=1):
        self.data = torch.tensor(data, dtype=torch.float32)
        self.lookback = lookback
        self.horizon = horizon
        self.stride = stride

    def __len__(self):
        return (len(self.data) - self.lookback - self.horizon) // self.stride + 1

    def __getitem__(self, index):
        i = index * self.stride
        x = self.data[i:i + self.lookback]
        y = self.data[i + self.lookback : i + self.lookback + self.horizon].squeeze(-1)
        return x, y

train_dataset = RollingTimeSeriesDataset(train, lookback, horizon)
valid_dataset = RollingTimeSeriesDataset(valid, lookback, horizon)
test_dataset = RollingTimeSeriesDataset(test, lookback, horizon)

batch_size = 32
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
valid_loader = DataLoader(valid_dataset, batch_size=batch_size, shuffle=False)
```

```
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# Print sample batch to verify
for x_batch, y_batch in train_loader:
    print(f"Input batch shape: {x_batch.shape}")
    print(f"Target batch shape: {y_batch.shape}")
    break
```

```
Input batch shape: torch.Size([32, 6, 1])
Target batch shape: torch.Size([32, 2])
```

20.5. Models

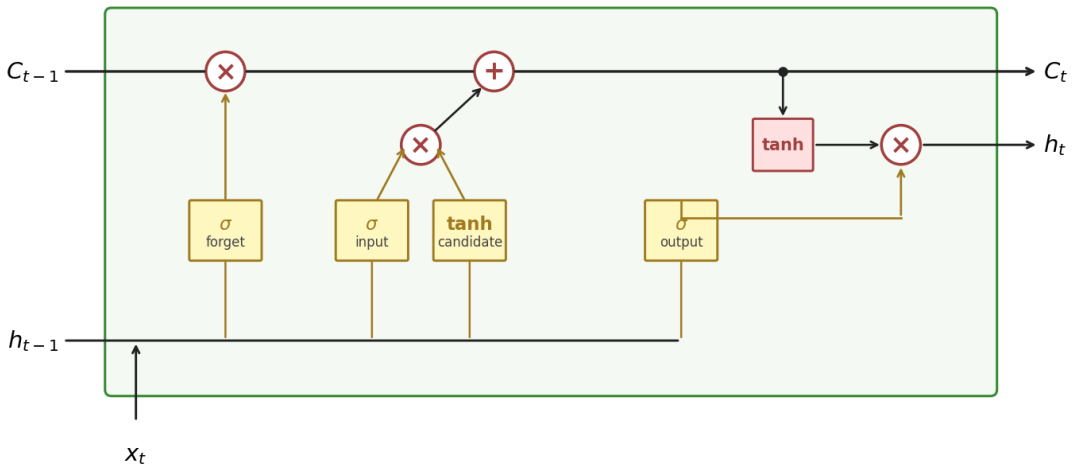
Vanilla RNNs process sequential data by maintaining a hidden state that is updated at each time step. However, they struggle with long-term dependencies due to the vanishing gradient problem. Learn more in the PyTorch RNN documentation.

```
class AirRNN(nn.Module):
    """
    A simple Recurrent Neural Network (RNN) for time series forecasting.

    This model consists of a two-layer RNN followed by a linear layer to predict
    the next `horizon` time steps based on past `lookback` steps.
    """
    def __init__(self):
        super().__init__()
        self.rnn = nn.RNN(input_size=1, hidden_size=64, num_layers=2, batch_first=True)
        self.linear = nn.Linear(64, horizon)

    def forward(self, x):
        x, _ = self.rnn(x)
        return self.linear(x[:, -1, :])
```

LSTM cell: forget / input / candidate / output gates



LSTMs are designed to handle long-term dependencies in sequential data by using gated mechanisms (input, forget, and output gates) to control information flow, reducing the vanishing gradient problem seen in vanilla RNNs. Learn more in the PyTorch LSTM documentation.

```
class AirLSTM(nn.Module):
    """
    A Long Short-Term Memory (LSTM) network for time series forecasting.

    This model consists of a two-layer bidirectional LSTM with dropout for regularization,
    followed by a linear layer to predict the next `horizon` time steps.
    """
    def __init__(self):
        super().__init__()
        self.lstm = nn.LSTM(input_size=1, hidden_size=64, num_layers=2, batch_first=True)
        self.linear = nn.Linear(64, horizon)

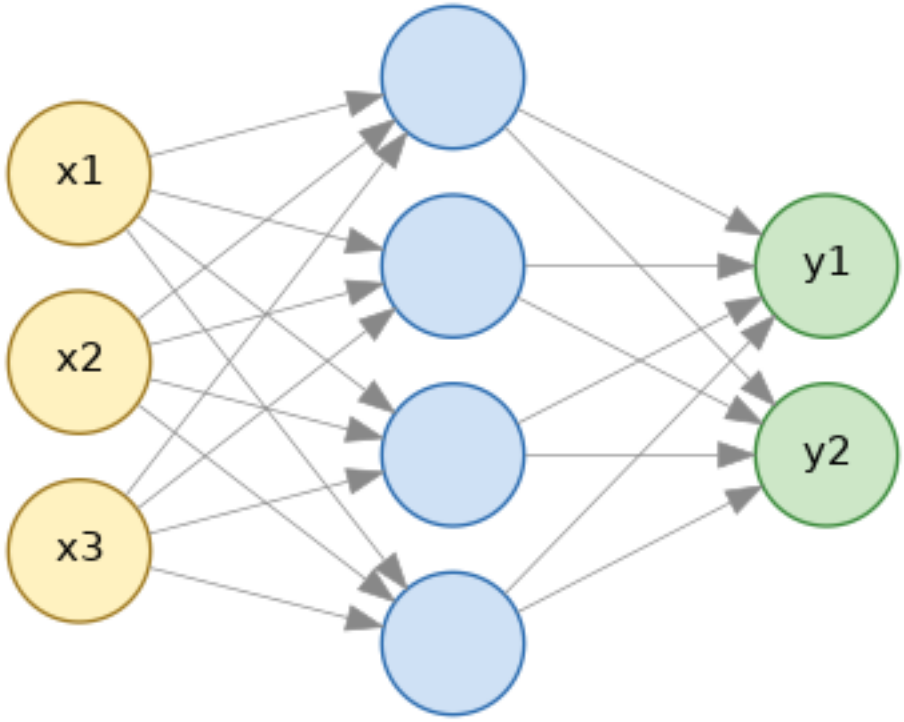
    def forward(self, x):
        x, _ = self.lstm(x)
        return self.linear(x[:, -1, :])
```

GRUs are a simplified variant of LSTMs that use gating mechanisms to control information flow without separate memory cells, making them computationally more efficient while still mitigating the vanishing gradient problem. Learn more in the PyTorch GRU documentation.

```
class AirGRU(nn.Module):
    """
    A Gated Recurrent Unit (GRU) network for time series forecasting.

    This model consists of a two-layer GRU followed by a linear layer
    to predict the next `horizon` time steps.
    """
    def __init__(self):
        super().__init__()
        self.gru = nn.GRU(input_size=1, hidden_size=64, num_layers=2, batch_first=True)
        self.linear = nn.Linear(64, horizon)

    def forward(self, x):
        x, _ = self.gru(x)
        return self.linear(x[:, -1, :])
```

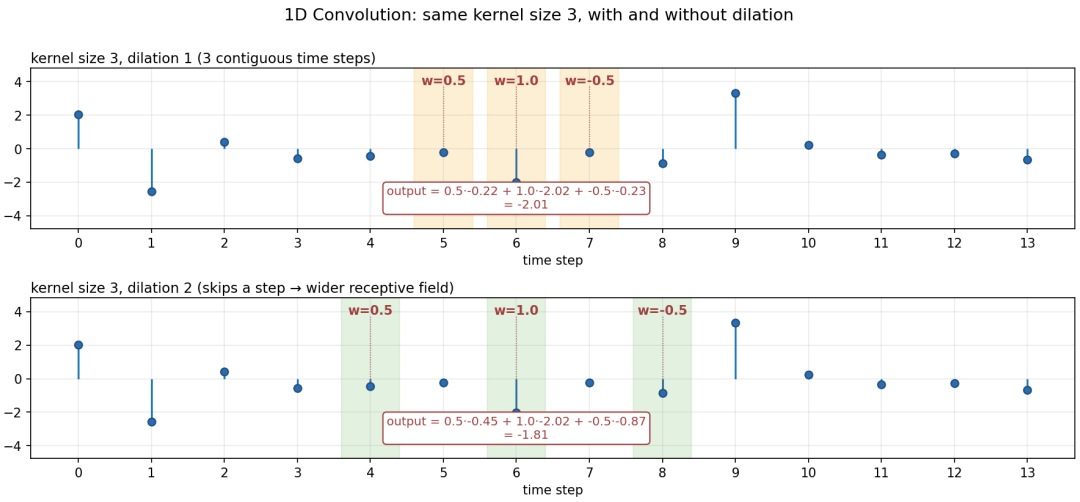


ANNs are the simplest type of neural networks, using fully connected layers to learn patterns in data. While they are effective for many tasks, they may struggle with sequential dependencies compared to RNN-based architectures. Learn more in the PyTorch Linear documentation.

```
class AirANN(nn.Module):
    """
    A simple Artificial Neural Network (ANN) for time series forecasting.

    This model consists of two fully connected (dense) layers. The first layer
    maps the `lookback` input features to 32 hidden units using ReLU activation,
    and the second layer maps to the `horizon` output values.
    """
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(lookback, 32)
        self.fc2 = nn.Linear(32, horizon)

    def forward(self, x):
        x = x.squeeze(-1)
        x = torch.relu(self.fc1(x))
        return self.fc2(x)
```



1D CNNs are widely used for time series forecasting because they can efficiently extract meaningful features from sequences. With dilation, they can model longer-term dependencies while maintaining computational efficiency. Learn more in the PyTorch 1D CNN documentation.

```python
class AirCNN(nn.Module):
    """
    A 1D Convolutional Neural Network (CNN) for time series forecasting.

    This model uses two convolutional layers with increasing dilation factors
    to capture temporal patterns effectively. The first convolution layer
    extracts local features, while the second one captures longer dependencies.
    The final output is obtained using a fully connected layer.
    """
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv1d(in_channels=1, out_channels=32, kernel_size=3, dilation=1, padding=1)
        self.conv2 = nn.Conv1d(in_channels=32, out_channels=64, kernel_size=3, dilation=2, padding=2)
        self.fc = nn.Linear(64, horizon)

    def forward(self, x):
        x = x.transpose(1, 2)  # Convert shape for Conv1D (batch, channels, sequence_length)
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = torch.mean(x, dim=2)  # Global Average Pooling
        return self.fc(x)
```

20.6. Train

The training function optimizes the model using the **AdamW optimizer** and incorporates a **OneCycleLR scheduler**, which dynamically adjusts the learning rate throughout training to prevent divergence and speed up convergence (Smith, 2019). For sequential models like RNNs, LSTMs, and GRUs, **teacher forcing** is applied, a technique where the actual target values replace some predicted values during training, helping the model learn long-term dependencies more effectively (Williams & Zipser, 1989). The model tracks validation loss, and if no improvement is observed over a given number of epochs (patience), early stopping is triggered to prevent overfitting, ensuring the best model weights are saved and restored.

```
def train_model(model, train_loader, valid_loader, n_epochs=3000, patience=50, teacher_forcing_ratio=0.5):
    """
    Trains a given neural network model for time series forecasting.

    Args:
        model (nn.Module): The model to train (RNN, LSTM, GRU, ANN, or CNN).
        train_loader (DataLoader): Dataloader for the training set.
        valid_loader (DataLoader): Dataloader for the validation set.
        n_epochs (int, optional): Maximum number of training epochs. Defaults to 3000.
        patience (int, optional): Number of epochs to wait before early stopping. Defaults to 50.
        teacher_forcing_ratio (float, optional): Probability of using ground truth instead of predicted value
                                                 for recurrent models. Defaults to 0.

    Returns:
        model (nn.Module): The trained model with the best validation performance.

    This function trains the model using the AdamW optimizer with a learning rate schedule (`OneCycleLR`)
    and applies teacher forcing for sequential models like RNNs, LSTMs, and GRUs. It also implements
    early stopping to prevent overfitting by monitoring validation loss. The best model weights are saved
    and loaded at the end of training.
    """

    optimizer = optim.AdamW(model.parameters(), lr=0.0001)
    scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr=0.005,
  steps_per_epoch=len(train_loader), epochs=n_epochs)
    loss_fn = nn.MSELoss()

    best_loss = float("inf")
    best_model_wts = None
    early_stop_counter = 0

    train_losses = []
    valid_losses = []

    is_recurrent = isinstance(model, (AirRNN, AirLSTM, AirGRU))

    for epoch in range(n_epochs):
        model.train()
        train_loss = 0

        for x_batch, y_batch in train_loader:
            x_batch, y_batch = x_batch.to(device), y_batch.to(device)
            optimizer.zero_grad()

            if is_recurrent:  # Apply Teacher Forcing only for RNNs, LSTMs, and GRUs
                batch_size = x_batch.size(0)
                input_seq = x_batch.clone()
                predictions = torch.zeros(batch_size, horizon).to(device)

                for t in range(horizon):
                    output = model(input_seq)

                    if output.shape[-1] == horizon:
                        output_step = output[:, t:t+1]
                    else:
                        output_step = output

                    if np.random.rand() < teacher_forcing_ratio:
                        input_seq = torch.cat((input_seq[:, 1:], y_batch[:, t:t+1].unsqueeze(-1)), dim=1)
                    else:
```

```
                    input_seq = torch.cat((input_seq[:, 1:], output_step.unsqueeze(-1)), dim=1)

                predictions[:, t] = output_step.squeeze(-1)

            loss = loss_fn(predictions, y_batch)

        else:
            y_pred = model(x_batch)
            loss = loss_fn(y_pred, y_batch)

        loss.backward()
        optimizer.step()
        scheduler.step()
        train_loss += loss.item()

    # Validation Step
    model.eval()
    valid_loss = 0
    with torch.no_grad():
        for x_batch, y_batch in valid_loader:
            x_batch, y_batch = x_batch.to(device), y_batch.to(device)
            y_pred = model(x_batch)
            valid_loss += loss_fn(y_pred, y_batch).item()

    train_loss /= len(train_loader)
    valid_loss /= len(valid_loader)

    train_losses.append(train_loss)
    valid_losses.append(valid_loss)

    if valid_loss < best_loss:
        best_loss = valid_loss
        best_model_wts = model.state_dict().copy()
        early_stop_counter = 0
    else:
        early_stop_counter += 1

    if early_stop_counter >= patience:
        print(f"Early stopping at epoch {epoch+1}")
        break

    if epoch % 50 == 0:
        print(f"Epoch {epoch+1}: Train Loss = {train_loss:.4f}, Valid Loss = {valid_loss:.4f}")

if best_model_wts:
    model.load_state_dict(best_model_wts)
    torch.save(best_model_wts, "best_model.pth")

plt.figure(figsize=(10, 5))
plt.plot(train_losses, label="Train Loss", color="blue")
plt.plot(valid_losses, label="Validation Loss", color="orange")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Training & Validation Loss Over Time")
plt.legend()
plt.show()

return model
```

20.7. Evaluating our Model

Dynamic Time Warping (DTW)

DTW measures the similarity between two time series by allowing flexible alignments, making it particularly useful for sequences with temporal shifts. Unlike traditional distance metrics, DTW computes the optimal warping path to minimize discrepancies. A lower **DTW distance** indicates a closer alignment between predicted and actual sequences, while higher values suggest significant deviations. Resource

Root Mean Squared Error (RMSE)

RMSE evaluates the model's predictive accuracy by calculating the square root of the average squared differences between actual and predicted values. It penalizes larger errors more heavily, making it sensitive to outliers. A lower RMSE indicates better performance, with **0** representing a perfect prediction. Resource

Mean Squared Error (MSE)

MSE is the mean of the squared differences between actual and predicted values. As it amplifies large errors due to squaring, it is useful for optimizing models but less interpretable than RMSE due to its squared units. Lower MSE values indicate more accurate predictions. Resource

Mean Absolute Percentage Error (MAPE)

MAPE expresses the prediction error as a percentage of actual values, making it scale-independent and easily interpretable. However, it can be problematic when actual values approach zero, leading to inflated errors. Lower **MAPE (%)** signifies higher forecasting accuracy. Resource

```
def evaluate_model(model, data, lookback, horizon, device, test_start):
    """
    Evaluates the trained model using multiple error metrics and visualizes the results.

    This function computes predictions using a rolling window approach, averages overlapping predictions,
    and evaluates performance using:
    - Dynamic Time Warping (DTW): Measures sequence similarity while considering time shifts.
    - Root Mean Squared Error (RMSE): Standard deviation of residuals.
    - Mean Squared Error (MSE): Penalizes larger errors more heavily.
    - Mean Absolute Percentage Error (MAPE): Measures relative error percentage.

    The function also plots actual vs. predicted values across training, validation, and test sets.

    Args:
        model (torch.nn.Module): Trained model for time series forecasting.
        data (np.array or torch.Tensor): Normalized time series data.
        lookback (int): Number of past timesteps used as input.
        horizon (int): Number of future steps to predict.
        device (torch.device): Device (CPU or GPU) for inference.
        test_start (int): Index where the test dataset begins.

    Returns:
        tuple:
            - np.array: Averaged rolling predictions for visualization.
            - dict: Error metrics including DTW, RMSE, MSE, and MAPE.
    """
```

```
    model.eval()
    data = torch.tensor(data, dtype=torch.float32)
    total_predictions = np.zeros((len(data),))
    count = np.zeros((len(data),))

    colors = ["red"] * train_size + ["yellow"] * valid_size + ["green"] * test_size

    with torch.no_grad():
        for i in range(len(data) - lookback - horizon + 1):
            x_input = data[i : i + lookback].unsqueeze(0).to(device)
            y_pred = model(x_input).cpu().numpy().flatten()
            total_predictions[i + lookback : i + lookback + horizon] += y_pred
            count[i + lookback : i + lookback + horizon] += 1

    averaged_predictions = np.divide(total_predictions, count, where=count > 0)

    y_pred_test = averaged_predictions[test_start:]
    y_true_test = data[test_start:].cpu().numpy().flatten()

    time_indices = np.arange(len(y_true_test))
    y_pred_test_pairs = list(zip(time_indices, y_pred_test))
    y_true_test_pairs = list(zip(time_indices, y_true_test))
    dtw_distance, _ = fastdtw(y_true_test_pairs, y_pred_test_pairs, dist=euclidean)

    # Compute Error Metrics
    rmse = np.sqrt(mean_squared_error(y_true_test, y_pred_test))
    mse = mean_squared_error(y_true_test, y_pred_test)
    mape = mean_absolute_percentage_error(y_true_test, y_pred_test)

    print("\nModel Evaluation Metrics")
    print("=" * 40)
    print(f"DTW Distance: {dtw_distance:.4f}")
    print(f"RMSE        : {rmse:.4f}")
    print(f"MSE         : {mse:.4f}")
    print(f"MAPE        : {mape:.4%}")
    print("=" * 40)

    plt.figure(figsize=(12, 6))
    plt.plot(data.cpu().numpy(), label="Actual Data", color="black", linestyle="dashed", linewidth=1.2)

    plt.plot(averaged_predictions[:train_size], label="Train Predictions", color="red", linewidth=1.5)
    plt.plot(range(train_size, train_size + valid_size), averaged_predictions[train_size:train_size +
valid_size],
             label="Validation Predictions", color="yellow", linewidth=1.5)
    plt.plot(range(train_size + valid_size, len(averaged_predictions)), averaged_predictions[train_size +
valid_size:],
             label="Test Predictions", color="green", linewidth=1.5)

    plt.xlabel("Time Steps")
    plt.ylabel("Normalized Passengers")
    plt.title("Time Series Forecasting: Actual vs. Predicted")
    plt.legend()
    plt.grid(True, linestyle="--", alpha=0.6)
    plt.show()

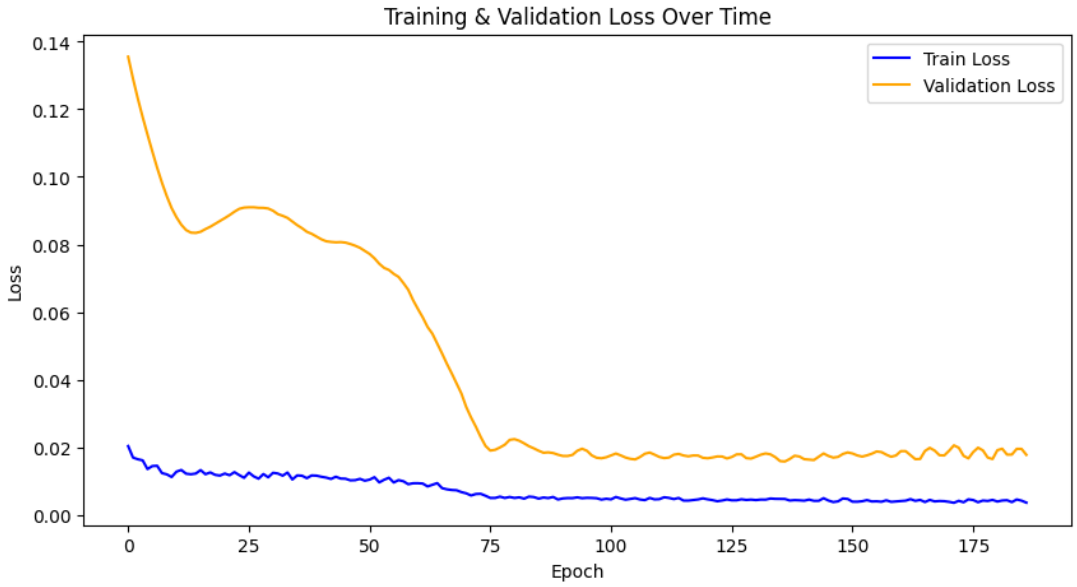
    return averaged_predictions, {"DTW": dtw_distance, "RMSE": rmse, "MSE": mse, "MAPE": mape}
```

20.7.1. LSTM

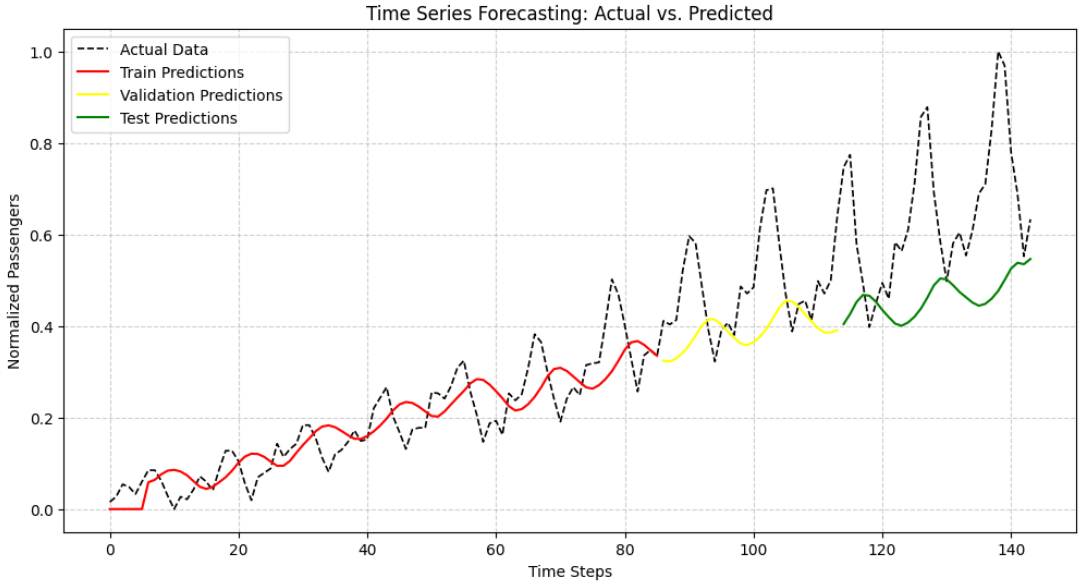
```
lstm_model = AirLSTM().to(device)
lstm_model = train_model(lstm_model, train_loader, valid_loader)

test_start = train_size + valid_size
rolling_predictions, metrics = evaluate_model(lstm_model, timeseries, lookback, horizon, device, test_start)
```

```
Epoch 1: Train Loss = 0.0203, Valid Loss = 0.1355
Epoch 51: Train Loss = 0.0104, Valid Loss = 0.0772
Epoch 101: Train Loss = 0.0046, Valid Loss = 0.0176
Epoch 151: Train Loss = 0.0039, Valid Loss = 0.0182
Early stopping at epoch 187
```




```
Model Evaluation Metrics
========================================
DTW Distance: 5.8203
RMSE        : 0.2422
MSE         : 0.0587
MAPE        : 26.6247%
========================================
```

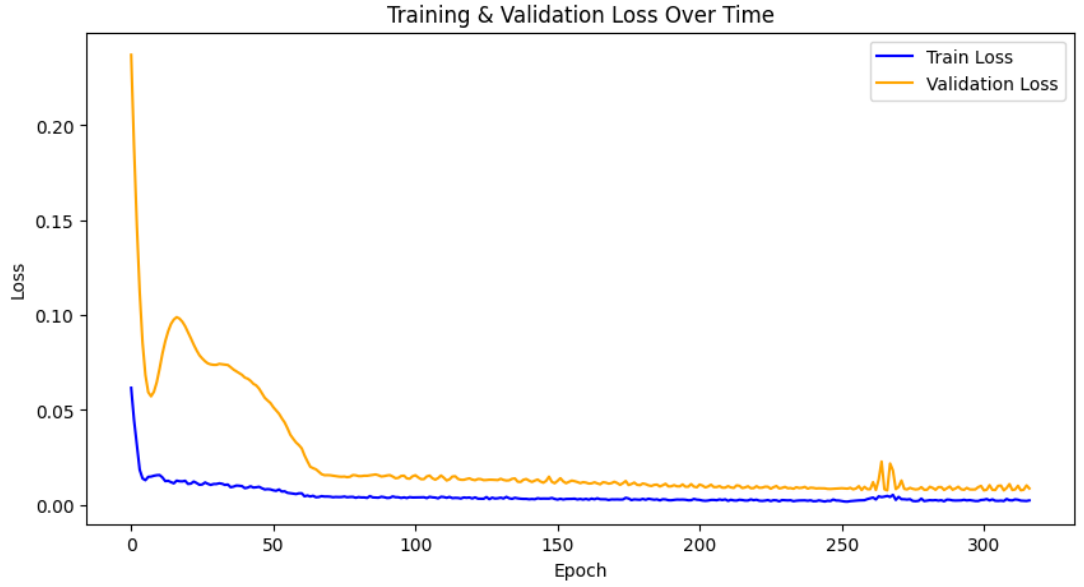


20.7.2. Vanilla RNN

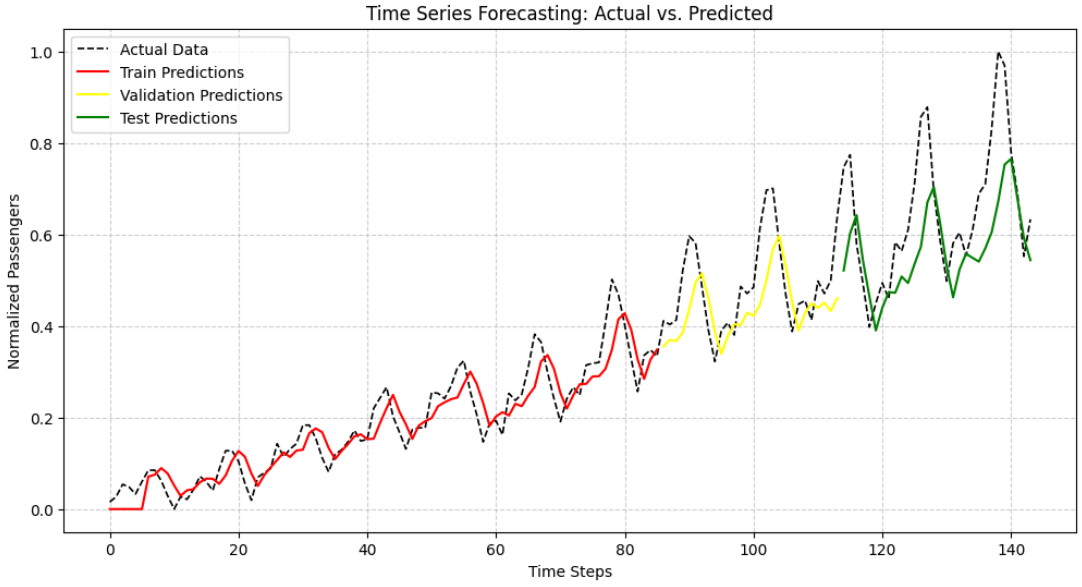
```
rnn_model = AirRNN().to(device)
rnn_model = train_model(rnn_model, train_loader, valid_loader)

test_start = train_size + valid_size
rolling_predictions, metrics = evaluate_model(rnn_model, timeseries, lookback, horizon, device, test_start)
```

```
Epoch 1: Train Loss = 0.0616, Valid Loss = 0.2372
Epoch 51: Train Loss = 0.0077, Valid Loss = 0.0515
Epoch 101: Train Loss = 0.0039, Valid Loss = 0.0156
Epoch 151: Train Loss = 0.0028, Valid Loss = 0.0126
Epoch 201: Train Loss = 0.0026, Valid Loss = 0.0107
Epoch 251: Train Loss = 0.0020, Valid Loss = 0.0087
Epoch 301: Train Loss = 0.0027, Valid Loss = 0.0076
Early stopping at epoch 317
```



```
Model Evaluation Metrics
========================================
DTW Distance: 3.2164
RMSE        : 0.1373
MSE         : 0.0189
MAPE        : 15.1573%
========================================
```



20.7.3. GRU

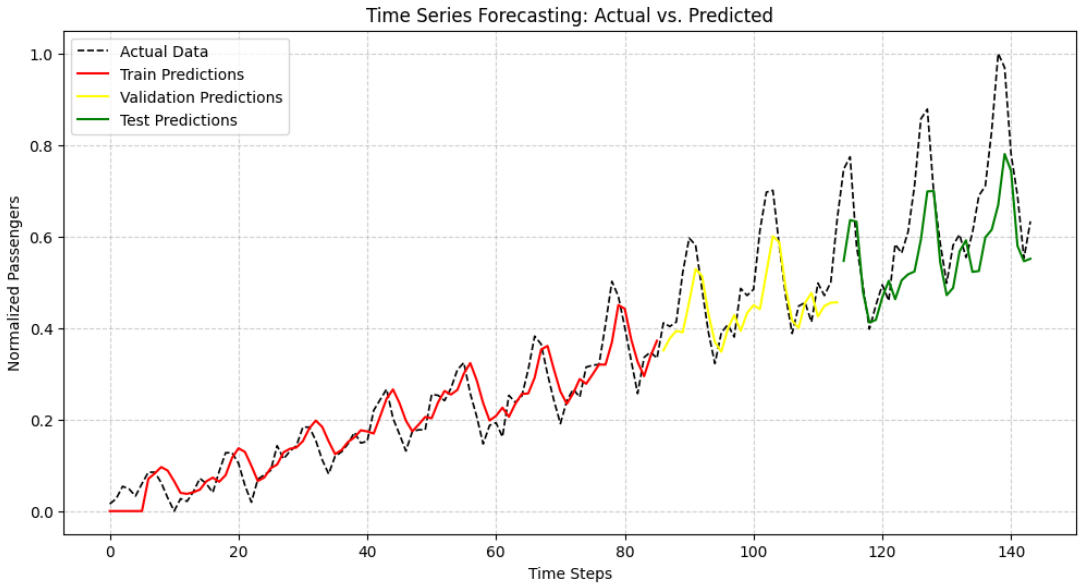
```
gru_model = AirGRU().to(device)
gru_model = train_model(gru_model, train_loader, valid_loader)

test_start = train_size + valid_size
rolling_predictions, metrics = evaluate_model(gru_model, timeseries, lookback, horizon, device, test_start)
```

```
Epoch 1: Train Loss = 0.0556, Valid Loss = 0.2277
Epoch 51: Train Loss = 0.0099, Valid Loss = 0.0636
Epoch 101: Train Loss = 0.0041, Valid Loss = 0.0166
Epoch 151: Train Loss = 0.0036, Valid Loss = 0.0147
Epoch 201: Train Loss = 0.0035, Valid Loss = 0.0116
Epoch 251: Train Loss = 0.0026, Valid Loss = 0.0095
Epoch 301: Train Loss = 0.0029, Valid Loss = 0.0103
Epoch 351: Train Loss = 0.0019, Valid Loss = 0.0102
Early stopping at epoch 364
```




```
Model Evaluation Metrics
========================================
DTW Distance: 3.0014
RMSE        : 0.1295
MSE         : 0.0168
MAPE        : 13.8546%
========================================
```

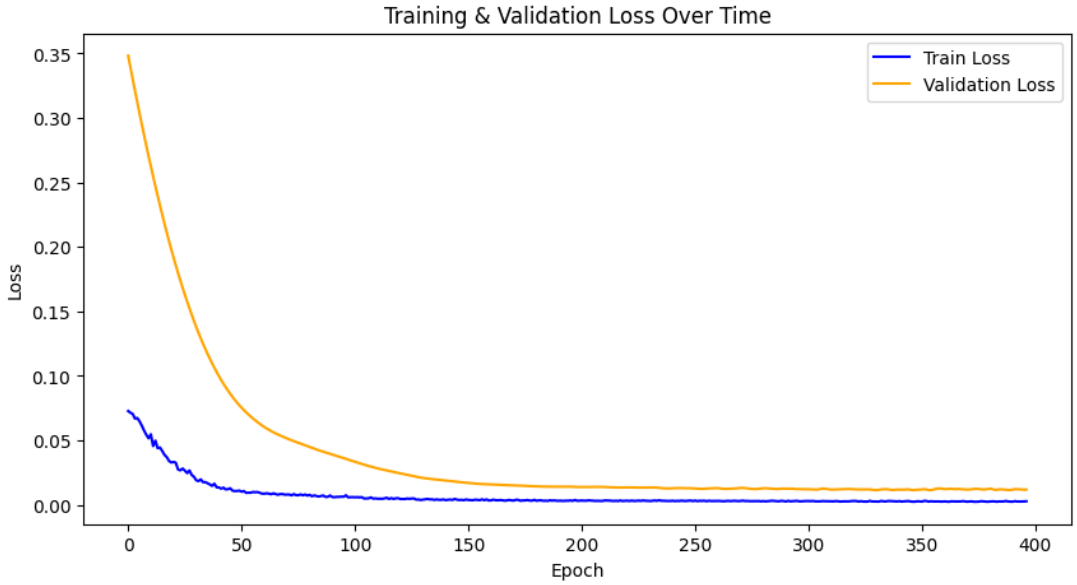


20.7.4. ANN

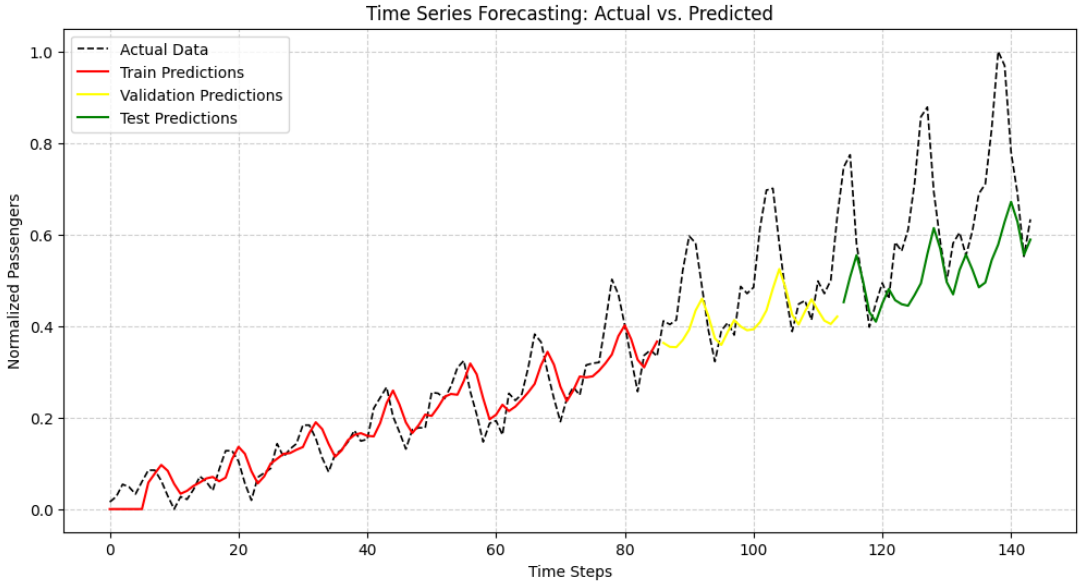
```
ann_model = AirANN().to(device)
ann_model = train_model(ann_model, train_loader, valid_loader)
```

```
test_start = train_size + valid_size
rolling_predictions, metrics = evaluate_model(ann_model, timeseries, lookback, horizon, device, test_start)
```

```
Epoch 1: Train Loss = 0.0727, Valid Loss = 0.3482
Epoch 51: Train Loss = 0.0102, Valid Loss = 0.0753
Epoch 101: Train Loss = 0.0058, Valid Loss = 0.0333
Epoch 151: Train Loss = 0.0037, Valid Loss = 0.0169
Epoch 201: Train Loss = 0.0033, Valid Loss = 0.0138
Epoch 251: Train Loss = 0.0033, Valid Loss = 0.0127
Epoch 301: Train Loss = 0.0029, Valid Loss = 0.0122
Epoch 351: Train Loss = 0.0025, Valid Loss = 0.0120
Early stopping at epoch 397
```




```
Model Evaluation Metrics
========================================
DTW Distance: 4.1342
RMSE        : 0.1854
MSE         : 0.0344
MAPE        : 18.5699%
========================================
```



20.7.5. CNN

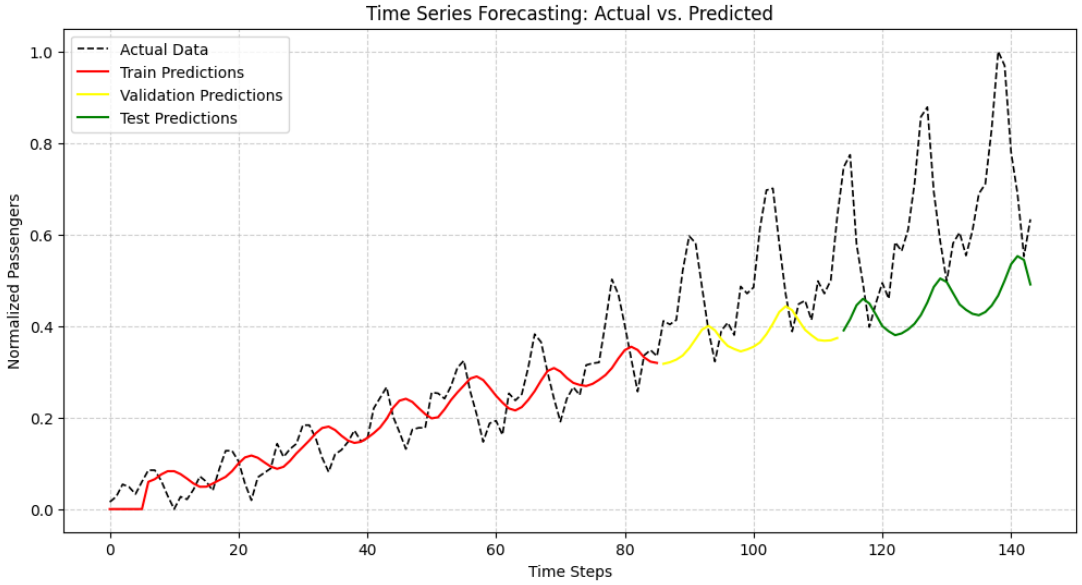
```
cnn_model = AirCNN().to(device)
cnn_model = train_model(cnn_model, train_loader, valid_loader)

test_start = train_size + valid_size
rolling_predictions, metrics = evaluate_model(cnn_model, timeseries, lookback, horizon, device, test_start)
```

```
Epoch 1: Train Loss = 0.0450, Valid Loss = 0.2077
Epoch 51: Train Loss = 0.0070, Valid Loss = 0.0354
Epoch 101: Train Loss = 0.0041, Valid Loss = 0.0173
Early stopping at epoch 143
```



```
Model Evaluation Metrics
========================================
DTW Distance: 6.2141
RMSE        : 0.2525
MSE         : 0.0638
MAPE        : 28.7658%
========================================
```



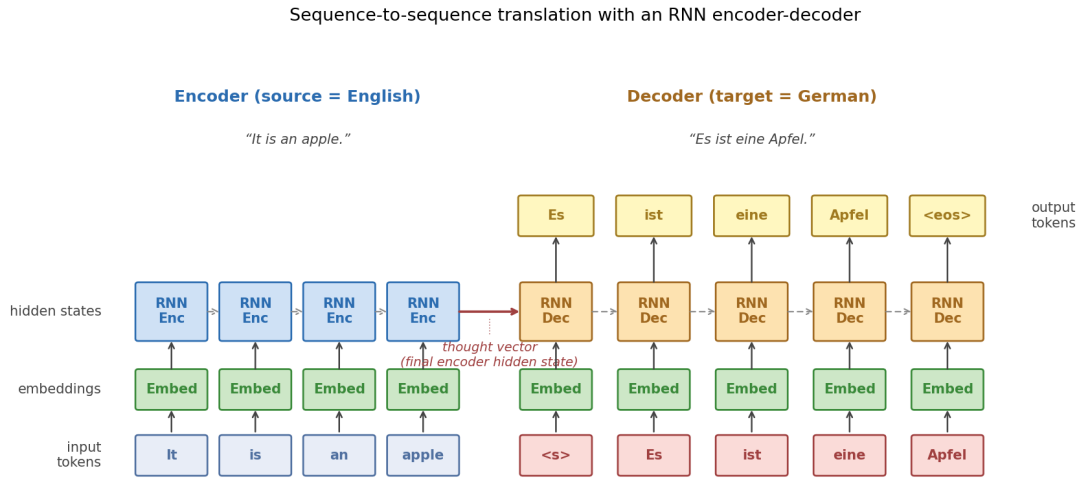
20.8. Homework (Optional)

1. **Rolling-origin evaluation.** Re-evaluate the LSTM on the airline-passenger series using 5 rolling-origin splits (train up to month `T`, evaluate month `T+1`, shift `T` forward). Compare the reported test RMSE with the single-split number and discuss the gap.
2. **Lookback study.** Train the same model with lookback windows of 3, 6, 12 and 24 months. Report test RMSE for each and identify the sweet spot.
3. **Model comparison.** For a fixed lookback, train the five model classes already in the notebook (RNN, LSTM, GRU, ANN, CNN) and report RMSE + parameter count + seconds per epoch. Produce a small table and argue which model wins on this dataset.
4. **Differencing for stationarity.** Apply first-order differencing to the target (`y_t - y_{t-1}`) before training, and re-invert after prediction. Report RMSE on the original scale and compare with the non-differenced baseline.
5. **Exogenous features.** Augment the input with month-of-year as a one-hot feature. Re-train and report whether the seasonal signal improves accuracy.
6. **Multi-step horizon.** Extend the model to predict the next 3 and 12 months in one shot instead of one. Report RMSE per horizon and plot the error as a function of horizon length.

Chapter 21

Neural Machine Translation

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1ewmlGuoJpNCILr2kR-rl39XCZaW1saRU?usp=sharing



Neural machine translation utilizes end-to-end deep learning models, typically based on encoder-decoder architectures enhanced by attention mechanisms, to map sequences from a source language directly to a target language. This approach enables the model to capture long-range dependencies and contextual subtleties within the text, thereby producing translations that are both fluent and semantically accurate, surpassing the performance of traditional statistical or rule-based translation systems.

21.1. Necessary Imports

```
import subprocess
import os
import pandas as pd
import nltk
import math
import copy
import numpy as np
import torch
import torch.nn as nn
import torch.nn.init as init
import matplotlib.pyplot as plt
from IPython.display import Image as IMG
from torch.optim import Adam, AdamW
from nltk.translate.bleu_score import sentence_bleu, corpus_bleu
```

```
from torch.utils.data import Dataset, DataLoader
from tqdm.notebook import tqdm
from nltk.tokenize import WordPunctTokenizer
from collections import Counter

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```



21.2. Data

For this practice, we will use a **bilingual dataset** containing sentences in multiple languages, available for download from Tatoeba and ManyThings. We will then filter this dataset to extract only the **Hungarian** and **English** sentence pairs, allowing us to focus on building and evaluating a neural machine translation model for these two languages.

```
%%bash
export PYTHONWARNINGS="ignore"
if [ -d "data" ]; then
    echo "data directory already exists. Skipping download."
else
    echo "data directory does not exist. Downloading..."
    mkdir -p data
    wget https://downloads.tatoeba.org/exports/sentences.tar.bz2 --quiet -P data
    tar xvjf data/sentences.tar.bz2 -C data
    wget https://downloads.tatoeba.org/exports/links.tar.bz2 --quiet -P data
    tar xvjf data/links.tar.bz2 -C data
    rm -rf data/sentences.tar.bz2
    rm -rf data/links.tar.bz2
fi
```

```
data directory does not exist. Downloading...
sentences.csv
links.csv
```

```
#ISO 639-3 code for a source language
source_lang = 'eng'
target_lang = 'hun'
data_dir = './'

# read all data
sentences = pd.read_csv('data/sentences.csv', names=['id', 'lang', 'text'], header=None, delimiter='\t')
links = pd.read_csv('data/links.csv', names=['sent_id', 'tran_id'], header=None, delimiter='\t')

# extract source - target connected
source_sentences = sentences[sentences.lang == source_lang]
target_sentences = sentences[sentences.lang == target_lang]
```

```
sentences.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 12539050 entries, 0 to 12539049
Data columns (total 3 columns):
 #   Column  Dtype
---  ------  -----
 0   id      int64
 1   lang    object
```

```
 2   text    object
dtypes: int64(1), object(2)
memory usage: 287.0+ MB
```

21.3. Data Exploration

Let's find out how each of the english sentences are linked to their hungarian variants through the `links` dataset.

```
source_sentences.head(5)
```

```
id lang                                             text
1232  1276  eng                               Let's try something.
1233  1277  eng                             I have to go to sleep.
1235  1280  eng  Today is June 18th and it is Muiriel's birthday!
1236  1282  eng                                 Muiriel is 20 now.
1237  1283  eng                         The password is "Muiriel".
```

```
source_sentences.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 1962151 entries, 1232 to 12539012
Data columns (total 3 columns):
 #   Column  Dtype
---  ------  -----
 0   id      int64
 1   lang    object
 2   text    object
dtypes: int64(1), object(2)
memory usage: 59.9+ MB
```

```
target_sentences.head(5)
```

```
id lang                                                text
313814  331032  hun                                    Ez nem lep meg.
363647  382214  hun  Minden emberi lény szabadon születik és egyenl...
381981  400937  hun                                          Hogy vagy?
382135  401101  hun                                           Köszönöm!
382138  401104  hun                                  Köszönöm szépen!
```

```
target_sentences.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 416601 entries, 313814 to 12534951
Data columns (total 3 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   id      416601 non-null  int64
 1   lang    416601 non-null  object
 2   text    416601 non-null  object
dtypes: int64(1), object(2)
memory usage: 12.7+ MB
```

```
links.head(5)
```

```
sent_id  tran_id
0        1     2481
```

```
1        1     5350
2        1   180624
3        1   344899
4        1   380381
```

```
ID = 1276

# We have around 108 translations for a given sentence
print(links[links['sent_id'] == ID])
```

```
sent_id   tran_id
15496     1276        77
15497     1276      2481
15498     1276      3091
15499     1276      4702
15500     1276      5350
...        ...       ...
15599     1276  12255186
15600     1276  12276948
15601     1276  12481488
15602     1276  12528549
15603     1276  12681229

[108 rows x 2 columns]
```

```
# Filter out the english sentences with ID of 1276
english_sentence = source_sentences[source_sentences['id'] == ID]
# Find all translations for sentence with ID of 1276 (108 items)
link_rows = links[links['sent_id'] == ID]
# Filter out all the translation IDs from the `link_rows` DataFrame
target_ids = link_rows['tran_id'].unique()
# Find the hungarian sentences whos ID is in the list of `target_ids`
hungarian_sentence = target_sentences[target_sentences['id'].isin(target_ids)]
```

```
print(english_sentence)
print()
print(hungarian_sentence)
```

```
id lang                 text
1232  1276  eng  Let's try something.

              id lang                    text
3525284  3753714  hun  Próbáljunk ki valamit.
```

21.4. Creating a Joint Dataset of Translations

```
source_sentences = source_sentences.merge(links, left_on='id', right_on='sent_id')
```

```
source_sentences.head(5)
```

```
id lang                 text  sent_id  tran_id
0  1276  eng  Let's try something.     1276       77
1  1276  eng  Let's try something.     1276     2481
2  1276  eng  Let's try something.     1276     3091
3  1276  eng  Let's try something.     1276     4702
4  1276  eng  Let's try something.     1276     5350
```

```
source_sentences.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7171781 entries, 0 to 7171780
Data columns (total 5 columns):
 #   Column   Dtype
---  ------   -----
 0   id       int64
 1   lang     object
 2   text     object
 3   sent_id  int64
 4   tran_id  int64
dtypes: int64(3), object(2)
memory usage: 273.6+ MB
```

```
bilang_sentences = source_sentences.merge(target_sentences, left_on='tran_id', right_on='id',
   suffixes=[source_lang, target_lang])
```

```
bilang_sentences.head(5)
```

```
ideng langeng                                        texteng  sent_id  \
0   1276     eng                             Let's try something.     1276
1   1277     eng                           I have to go to sleep.     1277
2   1277     eng                           I have to go to sleep.     1277
3   1277     eng                           I have to go to sleep.     1277
4   1280     eng  Today is June 18th and it is Muiriel's birthday!     1280

   tran_id    idhun langhun                                          texthun
0  3753714  3753714     hun                            Próbáljunk ki valamit.
1   414529   414529     hun                               Mennem kell aludni.
2   522344   522344     hun                               Aludni kell mennem.
3   534629   534629     hun                                 Le kell feküdnöm.
4  7902836  7902836     hun  Ma június 18-a van, és ez Muiriel születésnapja!
```

```
bilang_sentences.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 175318 entries, 0 to 175317
Data columns (total 8 columns):
 #   Column   Non-Null Count   Dtype
---  ------   --------------   -----
 0   ideng    175318 non-null  int64
 1   langeng  175318 non-null  object
 2   texteng  175318 non-null  object
 3   sent_id  175318 non-null  int64
 4   tran_id  175318 non-null  int64
 5   idhun    175318 non-null  int64
 6   langhun  175318 non-null  object
 7   texthun  175318 non-null  object
dtypes: int64(4), object(4)
memory usage: 10.7+ MB
```

```
bilang_sentences = bilang_sentences[['text' + source_lang, 'text' + target_lang]]
```

```
bilang_sentences.head(5)
```

```
texteng  \
0                             Let's try something.
1                           I have to go to sleep.
```

```
2                            I have to go to sleep.
3                            I have to go to sleep.
4  Today is June 18th and it is Muiriel's birthday!

                                            texthun
0                            Próbáljunk ki valamit.
1                               Mennem kell aludni.
2                               Aludni kell mennem.
3                                 Le kell feküdnöm.
4  Ma június 18-a van, és ez Muiriel születésnapja!
```

21.5. Save the New Subset

```
file_name = os.path.join(data_dir, '{source}-{target}.csv'.format(source=source_lang, target=target_lang))
bilang_sentences.to_csv(file_name, index=False, sep='\t')
```

```
!rm -rf data
```

21.6. Let's Load it Back in

```
corpus = pd.read_csv(os.path.join(data_dir, '{}-{}.csv'.format(source_lang, target_lang)), delimiter='\t')
print("Number of sentence pairs in the corpus:", len(corpus))
```

```
Number of sentence pairs in the corpus: 175318
```

```
corpus.head(5)
```

```
texteng  \
0                              Let's try something.
1                            I have to go to sleep.
2                            I have to go to sleep.
3                            I have to go to sleep.
4  Today is June 18th and it is Muiriel's birthday!

                                            texthun
0                            Próbáljunk ki valamit.
1                               Mennem kell aludni.
2                               Aludni kell mennem.
3                                 Le kell feküdnöm.
4  Ma június 18-a van, és ez Muiriel születésnapja!
```

21.7. Creating our Vocabulary

Creating a **vocabulary** is a crucial aspect of **natural language processing**, as it enables the efficient mapping between textual tokens and their corresponding numerical representations. Special tokens such as `SOS` (start-of-sentence), `EOS` (end-of-sentence), `UNK` (unknown), and `PAD` (padding) are essential for managing sequence boundaries, handling words that are not present in the vocabulary, and maintaining consistent input lengths for model training and text generation.

```
class Vocab:
    """
    A simple vocabulary class for mapping words to indices and vice versa.

    Attributes:
```

```
        index2word (dict): Maps indices to words.
        word2index (dict): Maps words to indices.
        Special tokens:
            SOS_token (str): Start-of-sentence token.
            EOS_token (str): End-of-sentence token.
            UNK_token (str): Unknown token for out-of-vocabulary words.
            PAD_token (str): Padding token.
    """
    def __init__(self):
        # Define special tokens and their fixed indices
        self.SOS_token = '<sos>'
        self.EOS_token = '<eos>'
        self.UNK_token = '<unk>'
        self.PAD_token = '<pad>'

        self.PAD_idx = 0
        self.SOS_idx = 1
        self.EOS_idx = 2
        self.UNK_idx = 3

        self.index2word = {
            self.SOS_idx: self.SOS_token,
            self.EOS_idx: self.EOS_token,
            self.UNK_idx: self.UNK_token,
            self.PAD_idx: self.PAD_token
        }
        self.word2index = {word: idx for idx, word in self.index2word.items()}

    def index_words(self, words):
        """
        Adds a list of words to the vocabulary.

        Args:
            words (iterable): An iterable of words to be added.
        """
        for word in words:
            self.index_word(word)

    def index_word(self, word):
        """
        Adds a single word to the vocabulary if it's not already present.

        Args:
            word (str): The word to add.
        """
        if word not in self.word2index:
            idx = len(self.index2word)
            self.word2index[word] = idx
            self.index2word[idx] = word

    def __len__(self):
        """
        Returns:
            int: Total number of words in the vocabulary.
        """
        assert len(self.index2word) == len(self.word2index), "Mappings are inconsistent."
        return len(self.index2word)

    def decode_words(self, indices):
        """
```

```
        Converts a list of indices into their corresponding words.

        Args:
            indices (list or iterable): List of word indices.

        Returns:
            list: List of words corresponding to the indices. Unknown indices are replaced with UNK_token.
        """
        return [self.index2word.get(i, self.UNK_token) for i in indices]

    def to_file(self, filename):
        """
        Saves the vocabulary (excluding special tokens) to a file with one word per line.

        Args:
            filename (str): Path to the output file.
        """
        # Exclude the first four special tokens
        words = [word for idx, word in sorted(self.index2word.items()) if idx >= 4]
        try:
            with open(filename, 'w', encoding='utf-8') as f:
                for word in words:
                    f.write(word + '\n')
        except IOError as e:
            print(f"Error saving vocabulary to {filename}: {e}")

    @classmethod
    def from_file(cls, filename):
        """
        Loads a vocabulary from a file where each line contains one word.

        Args:
            filename (str): Path to the file containing vocabulary words.

        Returns:
            Vocab: A new Vocab instance with the words loaded.
        """
        vocab = cls()  # Initialize with special tokens
        try:
            with open(filename, 'r', encoding='utf-8') as f:
                words = [line.strip() for line in f if line.strip()]
                vocab.index_words(words)
        except IOError as e:
            print(f"Error loading vocabulary from {filename}: {e}")
        return vocab

    def __repr__(self):
        """
        Returns a string representation of the vocabulary.
        """
        return f"Vocab(size={len(self)}, special_tokens={list(self.index2word.values())[:4]})"
```

21.8. Tokenization

Tokenization is a critical step in NLP that involves splitting raw text into smaller, meaningful units (`tokens`) such as words. This process allows us to convert text into numerical representations that can be fed into machine learning models. Moreover, thorough text preprocessing, including cleaning (e.g.,

removing HTML tags, punctuation, digits, and stopwords), helps reduce noise and improves model performance.

```
def preprocess_corpus(sents, tokenizer, min_word_count):
    """
    Preprocesses a corpus by tokenizing sentences, converting them to lowercase,
    and replacing words that occur less frequently than 'min_word_count' with the UNK token.

    Args:
        sents (iterable of str): Iterable of sentence strings.
        tokenizer (callable): A function that splits a sentence into tokens.
        min_word_count (int): Minimum frequency a word must have to be retained.

    Returns:
        list of list of str: A list where each element is a tokenized sentence.
    """
    # Tokenize and lowercase each sentence
    sents_tokenized = [[w.lower() for w in tokenizer(sent)] for sent in sents]

    # Count word frequencies across the entire corpus
    word_counts = Counter(token for sent in sents_tokenized for token in sent)

    # Replace infrequent words with the UNK token
    processed = []
    for tokens in sents_tokenized:
        new_tokens = [token if word_counts[token] >= min_word_count else '<unk>' for token in tokens]
        processed.append(new_tokens)

    return processed

def read_vocab(sents):
    """
    Builds a vocabulary from a list of tokenized sentences.

    Args:
        sents (iterable of list of str): Iterable of tokenized sentences.

    Returns:
        Vocab: A Vocab object populated with words from the sentences.
    """
    vocab = Vocab()
    for sent in sents:
        vocab.index_words(sent)
    return vocab
```

The `WordPunctTokenizer` is a simple yet effective tool that splits text into individual word and punctuation tokens. For this English-Hungarian machine translation task, it is particularly useful because it uniformly handles the tokenization of both languages.

```
max_length = 10
min_word_count = 1

tokenizers = {
    'eng': WordPunctTokenizer().tokenize,
    'hun': WordPunctTokenizer().tokenize
}

source_sents = preprocess_corpus(corpus['text' + source_lang], tokenizers[source_lang], min_word_count)
target_sents = preprocess_corpus(corpus['text' + target_lang], tokenizers[target_lang], min_word_count)
```

```
# Filter out sentence pairs where either sentence exceeds the maximum length.
filtered_pairs = sorted({
    (tuple(s), tuple(t))
    for s, t in zip(source_sents, target_sents)
    if len(s) <= max_length and len(t) <= max_length
})
source_sents, target_sents = zip(*filtered_pairs)

source_vocab = read_vocab(source_sents)
target_vocab = read_vocab(target_sents)

# Save the vocabularies to files.
source_vocab.to_file(os.path.join(data_dir, f'{source_lang}.vocab.txt'))
target_vocab.to_file(os.path.join(data_dir, f'{target_lang}.vocab.txt'))
```

```
# Print a summary of the processed corpus.
print('Corpus length: {}\nSource vocabulary size: {}\nTarget vocabulary size: {}'.format(
    len(source_sents), len(source_vocab.word2index), len(target_vocab.word2index)
))

examples = list(zip(source_sents, target_sents))[80:90]
for src, tgt in examples:
    print(f'Source: "{" ".join(src)}", target: "{" ".join(tgt)}"')
```

```
Corpus length: 149526
Source vocabulary size: 15523
Target vocabulary size: 56828
Source: "a child is missing .", target: "eltűnt egy gyerek ."
Source: "a child needs a mother .", target: "a gyereknek anyára van szüksége ."
Source: "a child needs a mother .", target: "egy gyereknek szüksége van anyára
."
Source: "a chinch bug is climbing the window .", target: "egy poloska mászik az
ablakon ."
Source: "a clan of fairies lives under there .", target: "tündérek serege él
azalatt ."
Source: "a cloud passed the sun .", target: "egy felhő haladt el a nap előtt ."
Source: "a cobra bit tom .", target: "tomit megmarta egy kobra ."
Source: "a coconut fell on tom ' s head .", target: "a kókuszdió tamás fejére
zuhant ."
Source: "a coke , please .", target: "egy kólát kérek ."
Source: "a cold wind blows .", target: "hideg szél fúj ."
```

```
print("Number of source sentences (english):", len(source_sents))
print("Number of target sentences (hungarian):", len(target_sents))
```

```
Number of source sentences (english): 149526
Number of target sentences (hungarian): 149526
```

21.8.1. Eliminating Duplicate Entries

```
def aggregate_source_target_pairs(source_sents, target_sents):
    """
    Group target sentences by their corresponding unique source sentence.

    Args:
        source_sents (iterable): Source sentences.
```

```
        target_sents (iterable): Target sentences.

    Returns:
        tuple: (unique_sources, grouped_targets) where unique_sources is a tuple of unique source sentences,
               and grouped_targets is a tuple of lists of target sentences for each source.
    """
    source_to_target = {}
    for source, target in zip(source_sents, target_sents):
        source_to_target.setdefault(source, []).append(target)

    unique_sources = tuple(source_to_target.keys())
    grouped_targets = tuple(source_to_target.values())
    return unique_sources, grouped_targets

unique_source, unique_targets = aggregate_source_target_pairs(source_sents, target_sents)
print("Number of unique source sentences:", len(unique_source))
print("Number of unique target sentences:", len(unique_targets))
```

```
Number of unique source sentences: 122507
Number of unique target sentences: 122507
```

```
print(target_sents)
print(unique_targets)
```

```
(('a', 'radioaktivitás', 'kémiai', 'fogalom', '.'), ('a', '100', 'évet',
'évszázadnak', 'nevezik', '.'), ('1975', 'az', 'az', 'év', ',', 'amikor',
'születtem', '.'), ('kétezerhuszo ... [6702363 chars truncated]
([('a', 'radioaktivitás', 'kémiai', 'fogalom', '.')], [('a', '100', 'évet',
'évszázadnak', 'nevezik', '.')], [('1975', 'az', 'az', 'év', ',', 'amikor',
'születtem', '.')], [('kétez ... [6947377 chars truncated]
```

```
examples_filtered = list(zip(unique_source, unique_targets))[80:100]
for src, tgts in examples_filtered:
    src_text = " ".join(src)
    tgt_texts = [ " ".join(tgt) for tgt in tgts ]
    print(f'Source: "{src_text}", Targets: {tgt_texts}')
```

```
Source: "a cold wind was blowing .", Targets: ['hideg szél fújt .']
Source: "a commission !", Targets: ['jutalék !']
Source: "a compound word consist of two smaller words .", Targets: ['egy
összetett szó két szóból áll .']
Source: "a computer is a wonderful thing .", Targets: ['egy számítógép
csodálatos dolog .']
Source: "a convergent sequence is bounded .", Targets: ['egy konvergens sorozat
korlátos is .']
Source: "a cool wind is blowing from the sea .", Targets: ['hűvös szél fúj a
tenger felől .']
Source: "a corpse was found at the bayou just yesterday .", Targets: ['egy
holttestet találtak tegnap a mocsárban .']
Source: "a couple more !", Targets: ['még egy párat !']
Source: "a cow has a long tail .", Targets: ['hosszú farka van a teheneknek .']
Source: "a crocodile ate thomas .", Targets: ['egy krokodil megette tamást .']
Source: "a crocodile ate tom .", Targets: ['egy krokodil megette tamást .']
Source: "a crocodile has eaten tom .", Targets: ['egy krokodil megette tamást
.']
Source: "a crow is as black as coal .", Targets: ['a varjú olyan fekete , mint a
```

```
szén .']
Source: "a crying girl opened the door .", Targets: ['egy síró lány nyitott
ajtót .', 'egy síró lány nyitotta ki az ajtót .', 'síró lany nyitott ajtót .']
Source: "a cube has six sides .", Targets: ['a kockának hat oldala van .']
Source: "a cunt remains a cunt !", Targets: ['a picsa picsa marad .']
Source: "a cup of coffee , please .", Targets: ['egy csésze kávét kérek .']
Source: "a cup of coffee costs a krone .", Targets: ['egy csésze kávé egy
koronába kerül .']
Source: "a cup of coffee costs one crown .", Targets: ['egy csésze kávé egy
koronába kerül .']
Source: "a cup of tea , please .", Targets: ['egy csésze tea lesz . köszönöm !']
```

21.9. Splitting the Date

We partition our aggregated sentence pair corpus into three subsets: **80% for training**, which the model will learn from, **6% for validation**, used to fine-tune hyperparameters and monitor performance, and **14% for testing**, reserved for evaluating the final model's generalization ability.

To structure our training data effectively, we **unwrap multi-target sentence pairs**, ensuring that each unique source sentence is paired individually with each of its corresponding target translations. This means that if a source sentence has multiple valid translations, we create separate training instances for each one:

(Source) "The weather is nice." -> (Target) "Az idő szép."

(Source) "The weather is nice." -> (Target) "Szép az idő."

(Source) "The weather is nice." -> (Target) "Kellemes az időjárás."

However, for the **validation set**, we preserve the original multi-target structure. This allows us to evaluate model predictions against all valid translations, ensuring a fairer and more flexible assessment of translation quality:

(Source) "The weather is nice." -> (Targets) ["Az idő szép.", "Szép az idő.", "Kellemes az időjárás."]

By keeping multiple references intact in validation, we can measure how well the model produces **any of the correct translations** rather than being restricted to a single expected output.

```
seed = 42
np.random.seed(seed)

source_length = len(unique_source)
indices = np.random.permutation(source_length)

train_split = int(source_length * 0.8)
valid_split = int(source_length * 0.86)

training_indices = indices[:train_split]
valid_indices = indices[train_split:valid_split]
test_indices = indices[valid_split:]
```

```
training_source = [unique_source[i] for i in training_indices]
valid_source = [unique_source[i] for i in valid_indices]
test_source = [unique_source[i] for i in test_indices]

training_target = [unique_targets[i] for i in training_indices]
valid_target = [unique_targets[i] for i in valid_indices]
test_target = [unique_targets[i] for i in test_indices]

# -------------------------
# 2. UNWRAP MULTI-TARGET TRAINING INSTANCES
# -------------------------
train_source_unwrapped = []
train_target_unwrapped = []

for src, tgt_list in zip(training_source, training_target):
    for tgt in tgt_list:
        train_source_unwrapped.append(src)
        train_target_unwrapped.append(tgt)

training_source = train_source_unwrapped
training_target = train_target_unwrapped
```

Visualizing Training Pairs

```
for i in range(10, 20):
    print(f"Training Source: {' '.join(training_source[i])}")
    print(f"Training Target: {' '.join(training_target[i])}\n")
```

```
Training Source: i have no objection to that .
Training Target: én nem kifogásolom azt .

Training Source: drive faster .
Training Target: lépj a gázra !

Training Source: drive faster .
Training Target: vezess gyorsabban !

Training Source: why isn ' t this working ?
Training Target: ez miért nem működik ?

Training Source: what is the difference ?
Training Target: mi a különbség ?

Training Source: are these real ?
Training Target: ezek valódiak ?

Training Source: your id , please .
Training Target: kérem az igazolványodat !

Training Source: this is a tree in the spring .
Training Target: ez egy tavaszi fa .

Training Source: you ' re a good student .
Training Target: jó diák vagy .

Training Source: you ' re a good student .
Training Target: jó tanuló vagy .
```

21.10. Padding Sequences and Converting them to Tensors

Before feeding our data into the model, we need to convert sentences into **fixed-length tensors of word indices**. This ensures that our model can process batches efficiently, as deep learning frameworks like PyTorch require **uniform input sizes** for batch operations.

Since sentences can have varying lengths, we use **padding** to bring them to a consistent length. We apply **end-padding** (rather than prefix-padding) so that shorter sentences do not affect the structure of the earlier words, which can be particularly beneficial for sequence-based models like LSTMs. Additionally, each sequence is **wrapped with special tokens**: `(start-of-sequence) at the beginning` and (end-of-sequence) at the end. This helps the decoder understand when a sentence starts and when it should stop generating words.

```
EOS_idx = target_vocab.word2index[target_vocab.EOS_token]
SOS_idx = target_vocab.word2index[target_vocab.SOS_token]
PAD_idx = target_vocab.word2index[target_vocab.PAD_token]

def indexes_from_sentence(vocab, sentence):
    """
    Converts a tokenized sentence into a list of corresponding indices from the vocabulary.

    Args:
        vocab (Vocab): Vocabulary object that maps words to indices.
        sentence (list of str): A tokenized sentence.

    Returns:
        list of int: List of word indices corresponding to the input sentence.
    """
    return [vocab.word2index[word] for word in sentence]

def tensor_from_sentence(vocab, sentence, max_seq_length):
    """
    Converts a tokenized sentence into a fixed-length tensor of token indices with padding.

    Args:
        vocab (Vocab): Vocabulary object containing word-index mappings.
        sentence (list of str): Tokenized sentence to be converted.
        max_seq_length (int): Maximum length for the sequence, including <SOS> and <EOS>.

    Returns:
        torch.Tensor: LongTensor of shape (max_seq_length,) containing token indices.
    """
    indexes = indexes_from_sentence(vocab, sentence)
    indexes.append(EOS_idx)  # Append end-of-sequence token
    indexes.insert(0, SOS_idx)  # Insert start-of-sequence token

    # Pad sequence to max length
    if len(indexes) < max_seq_length:
        indexes += [PAD_idx] * (max_seq_length - len(indexes))

    return torch.LongTensor(indexes).to(device)

def tensors_from_pair(source_sent, target_sent, max_seq_length):
    """
    Creates a tuple of (source_tensor, target_tensor) for a given source-target sentence pair.

    Args:
```

```
        source_sent (list of str): Tokenized source sentence.
        target_sent (list of str): Tokenized target sentence.
        max_seq_length (int): Maximum sequence length for both source and target.

    Returns:
        tuple: (source_tensor, target_tensor), each of shape (max_seq_length,).
    """
    source_tensor = tensor_from_sentence(source_vocab, source_sent, max_seq_length)
    target_tensor = tensor_from_sentence(target_vocab, target_sent, max_seq_length)
    return source_tensor, target_tensor

# Define sequence length including <SOS> and <EOS>
max_seq_length = max_length + 2

training = []
for source_sent, target_sent in zip(training_source, training_target):
    training.append(tensors_from_pair(source_sent, target_sent, max_seq_length))

x_training, y_training = zip(*training)
x_training = torch.stack(x_training).to(device)
y_training = torch.stack(y_training).to(device)

validation = []
for src, tgt_list in zip(valid_source, valid_target):
    source_tensor = tensor_from_sentence(source_vocab, src, max_seq_length)

    # Convert all target translations for this source into tensors
    target_tensors = [tensor_from_sentence(target_vocab, tgt, max_seq_length) for tgt in tgt_list]

    # Append a single (source_tensor, list_of_target_tensors) pair
    validation.append((source_tensor, target_tensors))

x_valid, y_valid = zip(*validation)
x_valid = torch.stack(x_valid).to(device)
y_valid = list(y_valid)

testing = []
for src, tgt_list in zip(test_source, test_target):
    source_tensor = tensor_from_sentence(source_vocab, src, max_seq_length)
    # Convert all target translations for this source into tensors
    target_tensors = [tensor_from_sentence(target_vocab, tgt, max_seq_length) for tgt in tgt_list]

    # Append a single (source_tensor, list_of_target_tensors) pair
    testing.append((source_tensor, target_tensors))

x_test, y_test = zip(*testing)
x_test = torch.stack(x_test).to(device)
y_test = list(y_test)
```

```
print("Training Sentence Shape: ", x_training.shape)
print("Training Target Shape: ", y_training.shape)
print("Validation Sentence Shape: ", x_valid.shape)
print("Validation Target Count: ", len(y_valid))
print("Test Sentence Shape: ", x_test.shape)
```

```
Training Sentence Shape:  torch.Size([119548, 12])
Training Target Shape:  torch.Size([119548, 12])
Validation Sentence Shape:  torch.Size([7351, 12])
Validation Target Count:  7351
Test Sentence Shape:  torch.Size([17151, 12])
```

21.11. Creating a Dataset and a Dataloader

To efficiently feed our data into the model, we use **PyTorch's Dataset and DataLoader classes**. Since our training and validation data have different structures, **one-to-one mappings in training** vs. **one-to-many mappings in validation**, we create a custom dataset to handle both cases.

The **`TranslationDataset`** class is designed to: - Load **preprocessed source and target tensors**. - Support both **training (single target per source)** and **validation (multiple targets per source).**

Additionally, we define a **custom `collate_fn` function**, which: - **Stacks** source sentences into a batch tensor. - **Handles validation targets properly**, preserving multiple references as lists instead of flattening them.

```
class TranslationDataset(Dataset):
    """
    PyTorch Dataset for loading preprocessed sentence tensor pairs.

    Supports:
    - Training: Each source sentence has one corresponding target sentence.
    - Validation: Each source sentence has multiple correct target sentences.

    Args:
        source_tensor (torch.Tensor): Tensor containing source sentences.
        target_tensor (torch.Tensor or list): Tensor of target sentences (for training) or list of multiple
   target tensors (for validation).
        is_validation (bool): If True, the dataset contains multiple target translations per source.
    """
    def __init__(self, source_tensor, target_tensor):
        self.source_tensor = source_tensor
        self.target_tensor = target_tensor

    def __len__(self):
        return len(self.source_tensor)

    def __getitem__(self, idx):
        return self.source_tensor[idx], self.target_tensor[idx]

def collate_fn(batch):
    """
    Collate function to handle batching for both training and validation datasets.

    Args:
        batch (list of tuples): Each tuple contains (source_tensor, target_tensor or list_of_target_tensors).

    Returns:
        - torch.Tensor: Batch of source sentences.
        - torch.Tensor (for training) or list of lists (for validation).
    """
    sources, targets = zip(*batch)

    # Stack sources as normal (tensor batch)
    sources = torch.stack(sources)

    # If targets are lists (validation case), keep them as lists
    if isinstance(targets[0], list):
        return sources, list(targets)  # List of target lists
```

```
    # Otherwise, stack targets normally (training case)
    return sources, torch.stack(targets)
```

```
# Create Dataset Instances
train_dataset = TranslationDataset(x_training, y_training)
valid_dataset = TranslationDataset(x_valid, y_valid)

# Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True, collate_fn=collate_fn)
valid_loader = DataLoader(valid_dataset, batch_size=64, shuffle=True, collate_fn=collate_fn)
```

Visualizing our Training Dataset

```
for source_batch, target_batch in train_loader:
    print(f"Training Batch Shapes: Source {source_batch.shape}, Target {target_batch.shape}\n")

    # Convert first 3 examples into readable text
    for i in range(min(3, len(source_batch))):
        src_text = " ".join(
            [source_vocab.index2word[idx.item()] for idx in source_batch[i]]
        )
        tgt_text = " ".join(
            [target_vocab.index2word[idx.item()] for idx in target_batch[i]]
        )
        print(f"Source: {src_text}")
        print(f"Target: {tgt_text}\n")
    break
```

```
Training Batch Shapes: Source torch.Size([64, 12]), Target torch.Size([64, 12])

Source: <sos> tom didn ' t want to work in australia . <eos>
Target: <sos> tamás nem akart ausztráliában dolgozni . <eos> <pad> <pad> <pad>
<pad>

Source: <sos> you seem to be afraid of my dog . <eos> <pad>
Target: <sos> úgy tűnik , félsz a kutyámtól . <eos> <pad> <pad> <pad>

Source: <sos> dad painted the walls white . <eos> <pad> <pad> <pad> <pad>
Target: <sos> apu fehérre festette a falakat . <eos> <pad> <pad> <pad> <pad>
```

Visualizing our Validation Dataset

```
for source_batch, target_batch in valid_loader:
    print(f"Validation Batch Shapes: Source {source_batch.shape}, Targets per Source: {len(target_batch)}\n")

    # Process the first 10 examples for readability
    for i in range(min(10, len(source_batch))):
        src_text = " ".join(
            [source_vocab.index2word[idx.item()] for idx in source_batch[i]]
        )
        tgt_texts = [
            " ".join([target_vocab.index2word[idx.item()] for idx in tgt])
            for tgt in target_batch[i]
        ]

        print(f"Source:\n{src_text}")
        print(f"Possible Translations:")
        for j, tgt in enumerate(tgt_texts, start=1):
```

```
            print(f"{tgt}")
        print()
    break
```

```
Validation Batch Shapes: Source torch.Size([64, 12]), Targets per Source: 64

Source:
<sos> i made him paint the house . <eos> <pad> <pad> <pad>
Possible Translations:
<sos> kifestettem vele a házat . <eos> <pad> <pad> <pad> <pad> <pad>
<sos> kifestettettem vele a házat . <eos> <pad> <pad> <pad> <pad> <pad>

Source:
<sos> he will travel abroad next year . <eos> <pad> <pad> <pad>
Possible Translations:
<sos> jövőre külföldre utazik majd . <eos> <pad> <pad> <pad> <pad> <pad>

Source:
<sos> tom and mary got married last spring . <eos> <pad> <pad>
Possible Translations:
<sos> tom és mari most tavasszal házasodtak össze . <eos> <pad> <pad>

Source:
<sos> we left a margin for error in our estimates . <eos>
[... 26 lines omitted ...]
Source:
<sos> stay here with us . <eos> <pad> <pad> <pad> <pad> <pad>
Possible Translations:
<sos> maradj itt velünk ! <eos> <pad> <pad> <pad> <pad> <pad> <pad>
<sos> maradjatok velünk itt ! <eos> <pad> <pad> <pad> <pad> <pad> <pad>

Source:
<sos> i ' ve done everything on my list . <eos> <pad>
Possible Translations:
<sos> mindent elintéztem a listámról . <eos> <pad> <pad> <pad> <pad> <pad>
```

21.12. Model

The **encoder** in an Encoder-Decoder network is a **Recurrent Neural Network (RNN)**, designed to process sequential data such as sentences. Unlike standard feedforward networks, RNNs **maintain a hidden state** that allows them to capture dependencies between elements in a sequence by using the output from the previous timestep as additional input for the next timestep.

However, **vanilla RNNs struggle with long sequences** due to vanishing and exploding gradient issues, making it difficult to learn dependencies over extended text. To address this, we use **Long Short-Term Memory (LSTM) networks**, which include a gating mechanism to regulate information flow, enabling them to **preserve long-range dependencies** effectively. PyTorch provides an efficient implementation via `nn.LSTM`, which we use in this model.

To further improve the encoder's ability to **capture relationships in text**, we use a **bidirectional LSTM**. Unlike standard LSTMs, which process input in a left-to-right manner, bidirectional LSTMs **process**

sequences in both directions, allowing the network to **capture both past and future context** for each word.

Additionally, we incorporate an **embedding layer** (`nn.Embedding`) to convert word indices into dense vector representations. This allows words to be represented in a **continuous space**, where semantically similar words are positioned closer together. In this implementation, we train these embeddings from scratch using the machine translation supervision signal. However, one could also use **pre-trained word embeddings** (such as FastText or Word2Vec) for better initialization, particularly when working with low-resource languages or small datasets.

21.13. Encoder

```
class EncoderRNN(nn.Module):
    """
    Encoder RNN for sequence-to-sequence models using a bidirectional LSTM.

    Args:
        vocab_size (int): Size of the vocabulary.
        hidden_size (int): Dimensionality of the hidden state.
        n_layers (int, optional): Number of LSTM layers. Default is 1.
    """
    def __init__(self, vocab_size, hidden_size, n_layers=1):
        super(EncoderRNN, self).__init__()
        self.vocab_size = vocab_size
        self.hidden_size = hidden_size
        self.n_layers = n_layers

        self.embedding = nn.Embedding(vocab_size, hidden_size)
        init.normal_(self.embedding.weight, 0.0, 0.2)

        # Because we use a bidirectional LSTM, the hidden state output will be doubled.
        # Therefore, we reduce the LSTM's output dimensionality to half.
        self.lstm = nn.LSTM(
            input_size=hidden_size,
            hidden_size=hidden_size // 2,
            num_layers=n_layers,
            batch_first=True,
            bidirectional=True
        )

    def forward(self, word_inputs, hidden):
        """
        Forward pass through the encoder.

        Args:
            word_inputs (Tensor): Tensor of shape (batch_size, seq_length) containing word indices.
            hidden (tuple): Tuple of (h_0, c_0) initial hidden states.

        Returns:
            output (Tensor): Output tensor of shape (batch_size, seq_length, hidden_size).
            hidden (tuple): Tuple of hidden states after processing.
        """
        embedded = self.embedding(word_inputs)
        output, hidden = self.lstm(embedded, hidden)
        return output, hidden

    def init_hidden(self, batch_size, device):
```

```
        """
        Initialize the hidden and cell states with zeros.

        Args:
            batch_size (int): Batch size.
            device (torch.device): The device to initialize the hidden states on.

        Returns:
            tuple: (h_0, c_0) each of shape (n_layers * 2, batch_size, hidden_size // 2).
        """
        h_0 = torch.zeros(self.n_layers * 2, batch_size, self.hidden_size // 2, device=device)
        c_0 = torch.zeros(self.n_layers * 2, batch_size, self.hidden_size // 2, device=device)
        return (h_0, c_0)
```

21.14. Decoder

```
class DecoderRNN(nn.Module):
    """
    Decoder RNN for sequence-to-sequence models using an LSTM.

    Args:
        vocab_size (int): Size of the vocabulary.
        hidden_size (int): Dimensionality of the hidden state (should match the encoder output).
        n_layers (int, optional): Number of LSTM layers. Default is 1.
    """
    def __init__(self, vocab_size, hidden_size, n_layers=1):
        super(DecoderRNN, self).__init__()
        self.vocab_size = vocab_size
        self.hidden_size = hidden_size
        self.n_layers = n_layers

        self.embedding = nn.Embedding(vocab_size, hidden_size)
        init.normal_(self.embedding.weight, 0.0, 0.2)

        self.lstm = nn.LSTM(
            input_size=hidden_size,
            hidden_size=hidden_size,
            num_layers=n_layers,
            batch_first=True,
            bidirectional=False
        )

    def forward(self, word_inputs, hidden):
        """
        Forward pass through the decoder for one time step.

        Args:
            word_inputs (Tensor): Tensor of shape (batch_size,) containing word indices.
            hidden (tuple): Tuple of (h_0, c_0) hidden states.

        Returns:
            output (Tensor): Output tensor of shape (batch_size, 1, hidden_size).
            hidden (tuple): Updated hidden states.
        """
        # Unsqueeze to add the time dimension: (batch_size, 1)
        embedded = self.embedding(word_inputs).unsqueeze(1)
        output, hidden = self.lstm(embedded, hidden)
        return output, hidden
```

Testing our Model with Fake Data

```
vocab_size = 10
hidden_dim = 10
n_layers = 2

encoder_test = EncoderRNN(vocab_size, hidden_dim, n_layers).to(device)
print(encoder_test)

batch_size = 1
encoder_hidden = encoder_test.init_hidden(batch_size, device)

test_seq_length = 3
word_input = torch.LongTensor([[1, 2, 3]]).to(device)

# Run the forward pass
encoder_outputs, encoder_hidden = encoder_test(word_input, encoder_hidden)

print(f"Encoder Outputs Shape: {encoder_outputs.shape}  (Expected: [1, {test_seq_length}, {hidden_dim}])")
print(f"Hidden State (h) Shape: {encoder_hidden[0].shape}  (Expected: [{n_layers*2}, 1, {hidden_dim//2}])")
print(f"Cell State (c) Shape: {encoder_hidden[1].shape}  (Expected: [{n_layers*2}, 1, {hidden_dim//2}])")

assert encoder_outputs.shape == (batch_size, test_seq_length, hidden_dim), "Encoder output shape mismatch!"
assert encoder_hidden[0].shape == (n_layers*2, batch_size, hidden_dim//2), "Hidden state shape mismatch!"
assert encoder_hidden[1].shape == (n_layers*2, batch_size, hidden_dim//2), "Cell state shape mismatch!"
```

```
EncoderRNN(
  (embedding): Embedding(10, 10)
  (lstm): LSTM(10, 5, num_layers=2, batch_first=True, bidirectional=True)
)
Encoder Outputs Shape: torch.Size([1, 3, 10])  (Expected: [1, 3, 10])
Hidden State (h) Shape: torch.Size([4, 1, 5])  (Expected: [4, 1, 5])
Cell State (c) Shape: torch.Size([4, 1, 5])  (Expected: [4, 1, 5])
```

```
decoder_test = DecoderRNN(vocab_size, hidden_dim, n_layers).to(device)
print(decoder_test)

decoder_hidden_h = encoder_hidden[0].view(n_layers, 2, batch_size, hidden_dim // 2).permute(0, 2, 1, 3).
  reshape(n_layers, batch_size, hidden_dim)
decoder_hidden_c = encoder_hidden[1].view(n_layers, 2, batch_size, hidden_dim // 2).permute(0, 2, 1, 3).
  reshape(n_layers, batch_size, hidden_dim)

for i in range(test_seq_length):
    input_word = word_input[:, i]
    decoder_output, decoder_hidden = decoder_test(input_word, (decoder_hidden_h, decoder_hidden_c))
    decoder_hidden_h, decoder_hidden_c = decoder_hidden

    print(f"Step {i+1}")
    print(f"Decoder Output Shape: {decoder_output.shape}  (Expected: [1, 1, {hidden_dim}])")
    print(f"Hidden State Shape: {decoder_hidden_h.shape}  (Expected: [{n_layers}, 1, {hidden_dim}])")
    print(f"Cell State Shape: {decoder_hidden_c.shape}  (Expected: [{n_layers}, 1, {hidden_dim}])\n")

assert decoder_output.shape == (batch_size, 1, hidden_dim), "Decoder output shape mismatch!"
assert decoder_hidden_h.shape == (n_layers, batch_size, hidden_dim), "Decoder hidden state shape mismatch!"
assert decoder_hidden_c.shape == (n_layers, batch_size, hidden_dim), "Decoder cell state shape mismatch!"
```

```
DecoderRNN(
  (embedding): Embedding(10, 10)
  (lstm): LSTM(10, 10, num_layers=2, batch_first=True)
```

```
)
Step 1
Decoder Output Shape: torch.Size([1, 1, 10])  (Expected: [1, 1, 10])
Hidden State Shape: torch.Size([2, 1, 10])  (Expected: [2, 1, 10])
Cell State Shape: torch.Size([2, 1, 10])  (Expected: [2, 1, 10])

Step 2
Decoder Output Shape: torch.Size([1, 1, 10])  (Expected: [1, 1, 10])
Hidden State Shape: torch.Size([2, 1, 10])  (Expected: [2, 1, 10])
Cell State Shape: torch.Size([2, 1, 10])  (Expected: [2, 1, 10])

Step 3
Decoder Output Shape: torch.Size([1, 1, 10])  (Expected: [1, 1, 10])
Hidden State Shape: torch.Size([2, 1, 10])  (Expected: [2, 1, 10])
Cell State Shape: torch.Size([2, 1, 10])  (Expected: [2, 1, 10])
```

21.15. Seq2Seq

The **Seq2Seq** module manages the interaction between the **encoder** and **decoder**, ensuring that the decoder's outputs are transformed into a probability distribution over the vocabulary.

It implements two forward functions:
- **Training mode** uses **Teacher Forcing**, where the decoder receives the correct target words at each step to speed up learning.
- **Inference mode** generates words step-by-step, using the model's own predictions as input, since ground-truth targets are unavailable.

```python
class Seq2seq(nn.Module):
    """
    A sequence-to-sequence model that coordinates an encoder and decoder to perform machine translation.

    The module includes separate forward passes for training (with teacher forcing) and inference.
    """
    def __init__(self, input_vocab_size, output_vocab_size, hidden_size, n_layers, device):
        super(Seq2seq, self).__init__()

        self.n_layers = n_layers
        self.hidden_size = hidden_size
        self.device = device

        self.encoder = EncoderRNN(input_vocab_size, hidden_size, n_layers).to(device)
        self.decoder = DecoderRNN(output_vocab_size, hidden_size, n_layers).to(device)

        self.W = nn.Linear(hidden_size, output_vocab_size).to(device)
        init.normal_(self.W.weight, 0.0, 0.2)
        self.softmax = nn.Softmax(dim=1)

    def _forward_encoder(self, x):
        """
        Pass the input through the encoder and transform its hidden state to initialize the decoder.

        Args:
            x (Tensor): Input tensor of shape (batch_size, seq_length).

        Returns:
            tuple: (decoder_hidden_h, decoder_hidden_c) each of shape (n_layers, batch_size, hidden_size).
        """
```

```
        batch_size = x.shape[0]
        init_hidden = self.encoder.init_hidden(batch_size, self.device)
        encoder_outputs, encoder_hidden = self.encoder(x, init_hidden)
        encoder_hidden_h, encoder_hidden_c = encoder_hidden

        # Encoder hidden state shape: (n_layers*2, batch_size, hidden_size/2)
        # We want to merge the two directions to form a tensor of shape: (n_layers, batch_size, hidden_size)
        decoder_hidden_h = encoder_hidden_h.view(self.n_layers, 2, batch_size, -1).permute(0, 2, 1, 3).
reshape(self.n_layers, batch_size, self.hidden_size)
        decoder_hidden_c = encoder_hidden_c.view(self.n_layers, 2, batch_size, -1).permute(0, 2, 1, 3).
reshape(self.n_layers, batch_size, self.hidden_size)
        return decoder_hidden_h, decoder_hidden_c

    def forward_train(self, x, y):
        decoder_hidden_h, decoder_hidden_c = self._forward_encoder(x)
        outputs = []

        for t in range(0, y.shape[1]):
            input_token = y[:, t].to(self.device)
            decoder_output, decoder_hidden = self.decoder(input_token, (decoder_hidden_h, decoder_hidden_c))
            decoder_hidden_h, decoder_hidden_c = decoder_hidden
            h = self.W(decoder_output)
            outputs.append(h)

        return torch.cat(outputs, dim=1)

    def forward(self, x):
        """
        Forward pass for inference (without teacher forcing).

        Args:
            x (Tensor): Input tensor (batch_size, seq_length).

        Returns:
            list: A list of generated token ID sequences (one per batch element).
        """
        decoder_hidden_h, decoder_hidden_c = self._forward_encoder(x)
        batch_size = x.shape[0]

        current_y = torch.full((batch_size,), 1, dtype=torch.long, device=self.device)

        results = [[] for _ in range(batch_size)]
        finished = torch.zeros(batch_size, dtype=torch.bool, device=self.device)

        max_steps = 100
        step = 0

        while not finished.all() and step < max_steps:
            decoder_output, decoder_hidden = self.decoder(current_y, (decoder_hidden_h, decoder_hidden_c))
            decoder_hidden_h, decoder_hidden_c = decoder_hidden

            h = self.W(decoder_output)
            y_probs = self.softmax(h.squeeze(1))
            _, next_y = torch.max(y_probs, dim=1)

            for i in range(batch_size):
                if not finished[i]:
                    results[i].append(next_y[i].item())
                    if next_y[i].item() == 2:  # <EOS> token
                        finished[i] = True
```

```
            current_y = next_y
            step += 1

        return results
```

21.16. Teacher forcing, free running, and exposure bias

A Seq2Seq decoder emits one target token at a time, conditioned on the previously generated token. At training time this creates a choice.

Teacher forcing feeds the **ground-truth** previous token at every step. Training converges quickly because every step has a clean input. The gradient flows only through the single-step conditional probability. This is the default and what the training loop in this notebook uses.

Free running feeds the model's **own** previous prediction at every step. Training is slower and noisier because early predictions corrupt the rest of the sequence, but the learner sees the distribution it will actually encounter at inference.

The mismatch between teacher-forced training and free-running inference is called *exposure bias*: the model has never seen its own mistakes during training, so small errors compound. Two standard mitigations exist.

Scheduled sampling is a smooth interpolation: with probability `p` feed the ground-truth token, otherwise feed the model's prediction. Start with `p = 1.0` and anneal toward a lower value over the course of training so the model gradually learns to recover from its own errors.

Beam search at inference, instead of greedy single-token decoding, keeps the top-`k` partial sequences and explores each. Beam size 4-8 typically adds 1-2 BLEU points over greedy decoding at proportional computational cost.

21.17. Training

We initialize the **Seq2Seq model** with a hidden size of **300** and **2 LSTM layers**.

For optimization, we use **AdamW**, a variant of Adam that improves weight decay handling, along with **CrossEntropyLoss**, which ignores " tokens to prevent them from affecting training. A **learning rate scheduler (ReduceLROnPlateau)** is applied to adjust the learning rate dynamically when validation loss stops improving.

```
model = Seq2seq(
    input_vocab_size=len(source_vocab),
    output_vocab_size=len(target_vocab),
    hidden_size=512,
    n_layers=3,
    device=device
).to(device)

optimizer = AdamW(model.parameters(), lr=0.0001)
criterion = nn.CrossEntropyLoss(ignore_index = PAD_idx)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, mode='min', patience=2, factor=0.5)
```

Performance Metrics

```
def bleu(n):
    """
    Create a BLEU score function with uniform n-gram weights.

    This function returns a lambda that computes the BLEU score for a given set
    of reference translations and hypothesis translations using uniform weights for
    n-grams up to n (and zero weight for higher n-grams up to 4).

    Args:
        n (int): The maximum n-gram order to consider (between 1 and 4).

    Returns:
        function: A lambda function that accepts two arguments:
                  - refs: list of reference translations (each a list of token lists)
                  - hyps: list of hypothesis translations (each a list of tokens)
                  and returns the computed BLEU score.
    """
    weights = [1.0 / n] * n + [0.0] * (4 - n)
    return lambda refs, hyps: corpus_bleu(refs, hyps, weights)

def accuracy(list_of_references, list_of_hypothesis):
    """
    Calculate the sentence-level accuracy based on exact match with any reference.

    For each hypothesis, this function checks if the hypothesis (converted to a tuple)
    exists in the set of corresponding reference translations (each reference is expected
    to be iterable and comparable as a tuple). The accuracy is the proportion of hypotheses
    that match at least one reference exactly.

    Args:
        list_of_references (list): A list where each element is an iterable of reference
                                   translations (token lists) for a sentence.
        list_of_hypothesis (list): A list of hypothesis translations (each a list of tokens)
                                   generated by the model.

    Returns:
        float: The accuracy as a fraction between 0 and 1.
    """
    total = 0.0
    for refs, hyp in zip(list_of_references, list_of_hypothesis):
        total += 1.0 if tuple(hyp) in set(map(tuple, refs)) else 0.0
    return total / len(list_of_references)

score_functions = {'BLEU-{}'.format(i): bleu(i) for i in range(1, 5)}
score_functions['Accuracy'] = accuracy

def score(model, valid_loader, target_vocab, desc='Scoring...'):
    """
    Compute evaluation scores (e.g., BLEU, Accuracy) for the validation set.

    Assumes that each batch from valid_loader contains:
    - `x_batch`: Source sequences.
    - `y_batch`: List of multiple reference translations per source sentence.

    Args:
        model (nn.Module): The seq2seq model.
        valid_loader (DataLoader): Validation DataLoader yielding (source, targets) pairs.
        target_vocab (Vocab): Vocabulary object to map indices to words.
        desc (str): Progress description.
```

```
    Returns:
        dict: Dictionary mapping score names to computed scores.
    """
    model.eval()
    list_of_hypothesis = []
    list_of_references = []

    for x_batch, y_batch in tqdm(valid_loader, desc=desc):
        x_batch = x_batch.to(device)

        # Get predicted sequences from the model (no teacher forcing)
        predicted_sequences = model.forward(x_batch)

        for pred, refs in zip(predicted_sequences, y_batch):
            pred_clean = [
                target_vocab.index2word.get(idx, '<unk>')
                for idx in pred
                if target_vocab.index2word.get(idx, '<unk>') not in {'<sos>', '<eos>', '<pad>'}
            ]

            refs_clean = [
                [
                    target_vocab.index2word.get(idx.item(), '<unk>')
                    for idx in ref if target_vocab.index2word.get(idx.item(), '<unk>') not in {'<sos>',
'<eos>', '<pad>'}
                ]
                for ref in refs
            ]

            list_of_hypothesis.append(pred_clean)
            list_of_references.append(refs_clean)

    print("\n**Sample Predictions & References:**")
    for i in range(min(3, len(list_of_hypothesis))):
        print(f"Predicted: {' '.join(list_of_hypothesis[i])}")
        print(f"References: {[' '.join(ref) for ref in list_of_references[i]]}\n")

    # Compute BLEU scores and accuracy
    scores = {name: func(list_of_references, list_of_hypothesis) for name, func in score_functions.items()}

    return scores
```

21.18. Main Training Loop

```
num_epochs = 15
early_stop_after = 5
early_stop_counter = 0
best_score = 0.0
scores_history = []
loss_history = []

for epoch in range(num_epochs):
    model.train()
    total_loss = 0.0

    train_iterator = tqdm(train_loader, desc=f"Epoch {epoch+1}/{num_epochs} Training", leave=False)
    for step, (x_batch, y_batch) in enumerate(train_iterator, start=1):
        x_batch = x_batch.to(device)
```

```
        y_batch = y_batch.to(device)

        y_tf = y_batch[:, :-1]
        y_true = y_batch[:, 1:]

        H = model.forward_train(x_batch, y_tf)

        H_flat = H.reshape(-1, H.size(-1))
        y_true_flat = y_true.reshape(-1)

        loss = criterion(H_flat, y_true_flat)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step()
        total_loss += loss.item()
        train_iterator.set_postfix(loss=loss.item())

    avg_loss = total_loss / len(train_loader)
    loss_history.append(avg_loss)
    print(f"Epoch {epoch+1}/{num_epochs} - Training Loss: {avg_loss:.4f}")

    # Validation phase with tqdm
    model.eval()
    val_scores = score(model, valid_loader, target_vocab, desc=f"Validation Epoch {epoch+1}")
    scores_history.append(val_scores)

    scores_str = "\n".join([f"{name}: {score:.4f}" for name, score in val_scores.items()])
    print(f"Epoch {epoch+1} Validation Scores:\n{scores_str}")

    metric = val_scores['BLEU-3']
    if metric > best_score:
        early_stop_counter = 0
        best_score = metric
        best_model_wts = copy.deepcopy(model.state_dict())
        print("New best model found. Reset early stop counter.")
    else:
        early_stop_counter += 1
        print(f"No improvement for {early_stop_counter} epochs.")
        if early_stop_counter >= early_stop_after:
            print("Early stopping triggered.")
            break
```

```
Epoch 1/15 Training:   0%|          | 0/1868 [00:00<?, ?it/s]
```

```
Epoch 1/15 - Training Loss: 4.0324
```

```
Validation Epoch 1:   0%|          | 0/115 [00:00<?, ?it/s]
```

```
**Sample Predictions & References:**
Predicted: tom tom , ugye ?
References: ['tom a közeledben lakik , nem ?']

Predicted: mondtam neked .
References: ['segítséget kértem .']

Predicted: tamás azt mondta , hogy mary most ?
References: ['tomi elmondta már marinak ?']

Epoch 1 Validation Scores:
```

```
BLEU-1: 0.4337
BLEU-2: 0.2028
BLEU-3: 0.1054
BLEU-4: 0.0593
Accuracy: 0.0084
New best model found. Reset early stop counter.
```

```
Epoch 2/15 Training:   0%|          | 0/1868 [00:00<?, ?it/s]
```

```
Epoch 2/15 - Training Loss: 3.5265
```

```
Validation Epoch 2:   0%|          | 0/115 [00:00<?, ?it/s]
```

```
**Sample Predictions & References:**
Predicted: tom egy nagy egész bort .
References: ['tom mindannyiunkat bolonddá tett .']

Predicted: hol hol dolgozni dolgozni ?
References: ['hol szoktál csalit venni pecázáshoz ?']

Predicted: nem akarok semmit semmit .
References: ['nem kérek cserébe semmit .']

Epoch 2 Validation Scores:
BLEU-1: 0.4763
BLEU-2: 0.2472
BLEU-3: 0.1407
BLEU-4: 0.0836
Accuracy: 0.0190
New best model found. Reset early stop counter.
```

```
Epoch 3/15 Training:   0%|          | 0/1868 [00:00<?, ?it/s]
```

```
Epoch 3/15 - Training Loss: 3.1263
```

```
Validation Epoch 3:   0%|          | 0/115 [00:00<?, ?it/s]
```

```
**Sample Predictions & References:**
Predicted: a munka már már lett , amikor voltam .
References: ['a munka befejeződött , mielőtt megérkeztem .']

Predicted: valaki ellopta a kocsimat .
References: ['valaki ellopta az útlevelemet .']

Predicted: én tomra még tom otthon volt .
References: ['azt hittem , tom még otthon van .']

Epoch 3 Validation Scores:
BLEU-1: 0.4896
BLEU-2: 0.2706
BLEU-3: 0.1662
BLEU-4: 0.1065
Accuracy: 0.0335
New best model found. Reset early stop counter.
```

```
Epoch 4/15 Training:   0%|          | 0/1868 [00:00<?, ?it/s]
```

```
Epoch 4/15 - Training Loss: 2.7879
```

```
Validation Epoch 4:   0%|          | 0/115 [00:00<?, ?it/s]
```

```
**Sample Predictions & References:**
Predicted: ez egy rossz történet .
References: ['olyan , mint egy rossz álom .']

Predicted: a szavak egymásnak .
References: ['szerelmesek lesznek egymásba .']

Predicted: hogyan szeretnél ?
References: ['hogy szeretnéd őket ?']

Epoch 4 Validation Scores:
BLEU-1: 0.5119
BLEU-2: 0.2935
BLEU-3: 0.1871
BLEU-4: 0.1242
Accuracy: 0.0433
New best model found. Reset early stop counter.
```

```
Epoch 5/15 Training:   0%|          | 0/1868 [00:00<?, ?it/s]
```

```
Epoch 5/15 - Training Loss: 2.4879
```

```
Validation Epoch 5:   0%|          | 0/115 [00:00<?, ?it/s]
```

```
**Sample Predictions & References:**
Predicted: mondd meg , mi a neved !
References: ['mondd meg nekem a nevedet !']

Predicted: beszélj a családodnak !
References: ['beszélj a főnököddel .']

Predicted: kapd be !
References: ['egyenesedj fel .', 'egyenesítsd ki .', 'hozd rendbe .', 'igazítsd
meg .', 'rakd rendbe .', 'szedd össze magad .']

Epoch 5 Validation Scores:
BLEU-1: 0.5181
BLEU-2: 0.3065
BLEU-3: 0.2005
BLEU-4: 0.1353
Accuracy: 0.0536
New best model found. Reset early stop counter.
```

```
Epoch 6/15 Training:   0%|          | 0/1868 [00:00<?, ?it/s]
```

```
Epoch 6/15 - Training Loss: 2.2174
```

```
Validation Epoch 6:   0%|          | 0/115 [00:00<?, ?it/s]
```

```
**Sample Predictions & References:**
Predicted: hallottuk a segítségét .
References: ['sikolyokat hallottunk .']

Predicted: kisebb egy házat .
References: ['reszkettem , mint egy falevél .']

Predicted: tom egy kis kis töltött volt a házban .
References: ['tom kagylóhéjat gyűjtött a tengerparton .']
```

```
Epoch 6 Validation Scores:
BLEU-1: 0.5231
BLEU-2: 0.3121
BLEU-3: 0.2061
BLEU-4: 0.1412
Accuracy: 0.0586
New best model found. Reset early stop counter.
```

```
Epoch 7/15 Training:   0%|          | 0/1868 [00:00<?, ?it/s]
```

```
Epoch 7/15 - Training Loss: 1.9671
```

```
Validation Epoch 7:   0%|          | 0/115 [00:00<?, ?it/s]
```

```
**Sample Predictions & References:**
Predicted: kilyukasztatta a bal bal falra .
References: ['a bal fülét kilyukasztatta .']

Predicted: kávé a teát kávét .
References: ['jobban szereti a teát , mint a kávét .']

Predicted: miért jelent el az iskolában , amikor mikor történt ?
References: ['miért csúsznak az utak , mikor esik az eső ?']

Epoch 7 Validation Scores:
BLEU-1: 0.5299
BLEU-2: 0.3226
BLEU-3: 0.2189
BLEU-4: 0.1534
Accuracy: 0.0709
New best model found. Reset early stop counter.
```

```
Epoch 8/15 Training:   0%|          | 0/1868 [00:00<?, ?it/s]
```

```
Epoch 8/15 - Training Loss: 1.7374
```

```
Validation Epoch 8:   0%|          | 0/115 [00:00<?, ?it/s]
```

```
**Sample Predictions & References:**
Predicted: meg akartam ölni .
References: ['meg akartam ölni .', 'meg akartam ölni őt .', 'meg akartam őt ölni
.']

Predicted: teljesen őrült van .
References: ['teljesen be van csavarodva .', 'teljesen hibbant .', 'totál be van
kattanva .']

Predicted: beszélünk erről vagy nem ?
References: ['beszélünk erről vagy sem ?']

Epoch 8 Validation Scores:
BLEU-1: 0.5248
BLEU-2: 0.3222
BLEU-3: 0.2194
BLEU-4: 0.1534
Accuracy: 0.0741
New best model found. Reset early stop counter.
```

```
Epoch 9/15 Training:   0%|          | 0/1868 [00:00<?, ?it/s]
```

```
Epoch 9/15 - Training Loss: 1.5256
```

```
Validation Epoch 9:   0%|          | 0/115 [00:00<?, ?it/s]
```

```
**Sample Predictions & References:**
Predicted: tom átment az összes születésnapi kórusban .
References: ['tom barna papírral becsomagolta a csomagot .', 'tom
csomagolópapírba csomagolta a csomagot .', 'tom csomagolópapírral csomagolta be
a csomagot .']

Predicted: maradj távol a kalapodat !
References: ['ne közelíts a barátnőmhöz !']

Predicted: mindannyian szegények vagyunk .
References: ['mi mind szinglik vagyunk .', 'mi mindannyian egyedülállók vagyunk
.']

Epoch 9 Validation Scores:
BLEU-1: 0.5312
BLEU-2: 0.3299
BLEU-3: 0.2255
BLEU-4: 0.1582
Accuracy: 0.0778
New best model found. Reset early stop counter.
```

```
Epoch 10/15 Training:   0%|          | 0/1868 [00:00<?, ?it/s]
```

```
Epoch 10/15 - Training Loss: 1.3292
```

```
Validation Epoch 10:   0%|          | 0/115 [00:00<?, ?it/s]
```

```
**Sample Predictions & References:**
Predicted: mit tegyek a telefonodat ?
References: ['mit vesszthetek ?']

Predicted: tom korán vesz lefeküdni .
References: ['tamás korán lefeküdt .']

Predicted: tom nem lett idősebb , mint mari .
References: ['tom nem lehet idősebb marynél .']

Epoch 10 Validation Scores:
BLEU-1: 0.5422
BLEU-2: 0.3389
BLEU-3: 0.2345
BLEU-4: 0.1666
Accuracy: 0.0853
New best model found. Reset early stop counter.
```

```
Epoch 11/15 Training:   0%|          | 0/1868 [00:00<?, ?it/s]
```

```
Epoch 11/15 - Training Loss: 1.1512
```

```
Validation Epoch 11:   0%|          | 0/115 [00:00<?, ?it/s]
```

```
**Sample Predictions & References:**
Predicted: sok sok bolt a bolt bezárt .
References: ['sok múzeum van hétfőn zárva .']

Predicted: ez a film igazinak készült .
References: ['sírtam azon a filmen .']

Predicted: mire ezt meg mit tennünk ?
References: ['ez mit kellene , hogy jelentsen ?']

Epoch 11 Validation Scores:
BLEU-1: 0.5449
BLEU-2: 0.3423
BLEU-3: 0.2390
BLEU-4: 0.1708
Accuracy: 0.0895
New best model found. Reset early stop counter.
```

```
Epoch 12/15 Training:   0%|          | 0/1868 [00:00<?, ?it/s]
```

```
Epoch 12/15 - Training Loss: 0.9882
```

```
Validation Epoch 12:   0%|          | 0/115 [00:00<?, ?it/s]
```

```
**Sample Predictions & References:**
Predicted: tom éppen most érkezett el , hogy megérkeztünk .
References: ['épp megjött tom , amikor mi indulni akartunk .']

Predicted: minden otthon otthon lett .
References: ['az összes diák hazament .']

Predicted: folytassuk meg egy filmet .
References: ['távozzunk angolosan !']

Epoch 12 Validation Scores:
BLEU-1: 0.5481
BLEU-2: 0.3462
BLEU-3: 0.2420
BLEU-4: 0.1734
Accuracy: 0.0947
New best model found. Reset early stop counter.
```

```
Epoch 13/15 Training:   0%|          | 0/1868 [00:00<?, ?it/s]
```

```
Epoch 13/15 - Training Loss: 0.8432
```

```
Validation Epoch 13:   0%|          | 0/115 [00:00<?, ?it/s]
```

```
**Sample Predictions & References:**
Predicted: csodás elég kedves , igaz ?
References: ['jó messze van hokkaido , mi ?']

Predicted: eltörte a karját csapott .
References: ['focizás közben eltörte a karját .']

Predicted: még nem mondtam semmit tomnak .
References: ['tamásnak még semmit sem mondtam .']

Epoch 13 Validation Scores:
BLEU-1: 0.5462
BLEU-2: 0.3458
```

```
BLEU-3: 0.2435
BLEU-4: 0.1769
Accuracy: 0.0951
New best model found. Reset early stop counter.
```

```
Epoch 14/15 Training:   0%|          | 0/1868 [00:00<?, ?it/s]
```

```
Epoch 14/15 - Training Loss: 0.7142
```

```
Validation Epoch 14:   0%|          | 0/115 [00:00<?, ?it/s]
```

```
**Sample Predictions & References:**
Predicted: ez egy nagyon ostoba döntés volt .
References: ['nagyon ostoba döntés volt .']

Predicted: ez nem működik ?
References: ['nem működik ?']

Predicted: tudtam , hogy mindenki ott van .
References: ['mindenkit ismertem , aki ott volt .']

Epoch 14 Validation Scores:
BLEU-1: 0.5470
BLEU-2: 0.3487
BLEU-3: 0.2469
BLEU-4: 0.1803
Accuracy: 0.1004
New best model found. Reset early stop counter.
```

```
Epoch 15/15 Training:   0%|          | 0/1868 [00:00<?, ?it/s]
```

```
Epoch 15/15 - Training Loss: 0.6051
```

```
Validation Epoch 15:   0%|          | 0/115 [00:00<?, ?it/s]
```

```
**Sample Predictions & References:**
Predicted: mari sok vizet iszik .
References: ['mary sok vizet iszik .']

Predicted: mutass valamit nekem .
References: ['valami mást mutass nekem !']

Predicted: nem adtak meg semmilyen semmilyen részletet .
References: ['nem adtak meg további részleteket .']

Epoch 15 Validation Scores:
BLEU-1: 0.5439
BLEU-2: 0.3455
BLEU-3: 0.2446
BLEU-4: 0.1785
Accuracy: 0.1020
No improvement for 1 epochs.
```

21.19. Plotting our Results

```
def plot_training_progress(loss_history, scores_history, score_functions, epoch):
    """
    Plots training loss and evaluation metrics (BLEU, Accuracy) over epochs.

    Args:
```

```
        loss_history (list): List of training loss values per epoch.
        scores_history (list of dicts): List of evaluation scores per epoch (each dict contains score names as
    keys).
        score_functions (dict): Dictionary containing scoring function names.
        epoch (int): The last completed epoch.
    """
    num_metrics = len(score_functions)
    total_plots = num_metrics + 1
    ncols = 2
    nrows = int(np.ceil(total_plots / ncols))

    fig, axes = plt.subplots(nrows, ncols, figsize=(14, 4 * nrows), dpi=80)
    axes = axes.flatten()
    epochs = list(range(1, epoch + 2))

    epochs_ticks = list(range(1, epoch + 2, max(1, epoch // 10)))

    # ---- Plot Loss History ----
    axes[0].plot(epochs, loss_history, color='dodgerblue', linewidth=2)
    axes[0].set_title("Training Loss Over Epochs", fontsize=14, fontweight='bold')
    axes[0].set_xlabel("Epochs")
    axes[0].set_xticks(epochs_ticks)
    axes[0].set_ylabel("Loss")
    axes[0].grid(True)

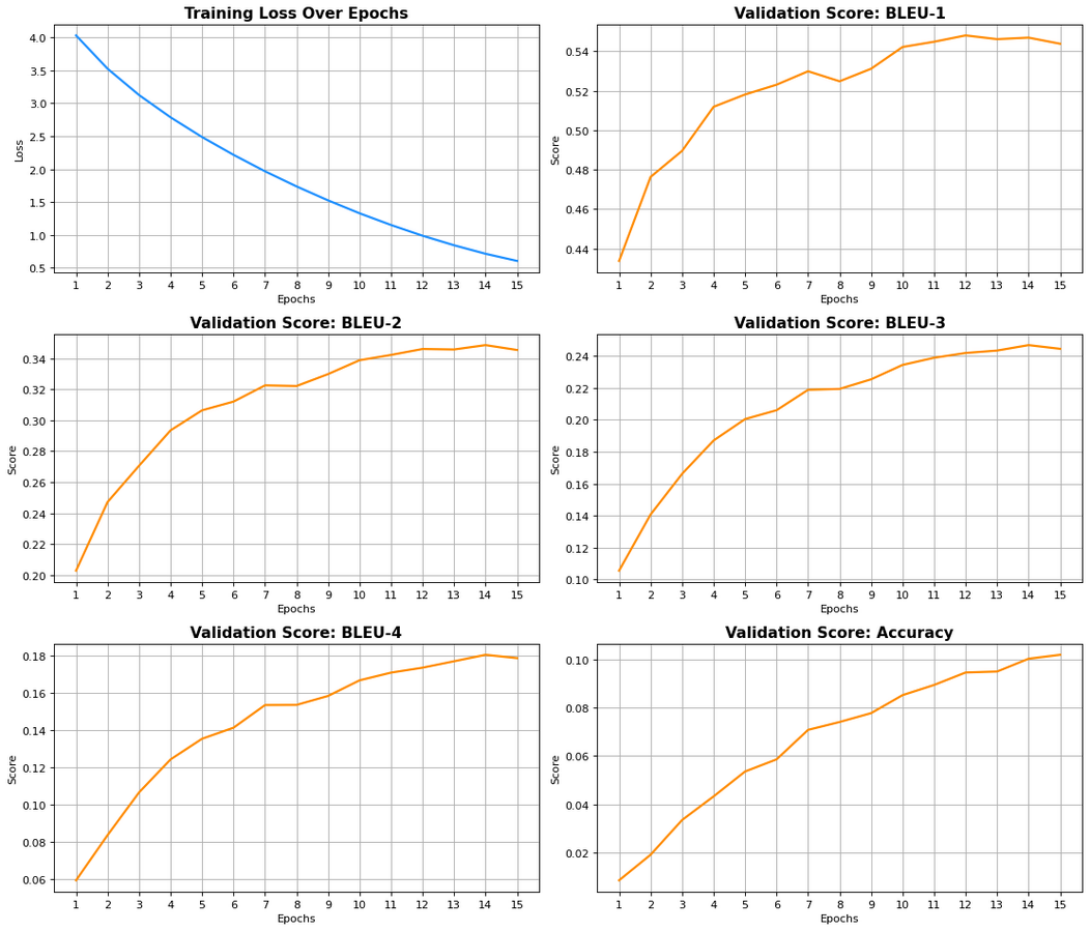
    # ---- Plot BLEU & Accuracy Scores ----
    for idx, name in enumerate(score_functions.keys(), start=1):
        score_history = [s[name] for s in scores_history]

        axes[idx].plot(epochs, score_history, color='darkorange', linewidth=2)
        axes[idx].set_title(f"Validation Score: {name}", fontsize=14, fontweight='bold')
        axes[idx].set_xlabel("Epochs")
        axes[idx].set_xticks(epochs_ticks)
        axes[idx].set_ylabel("Score")
        axes[idx].grid(True)

    # Hide unused subplots (if any)
    for idx in range(total_plots, len(axes)):
        fig.delaxes(axes[idx])

    plt.tight_layout()
    plt.show()
```

```
plot_training_progress(loss_history, scores_history, score_functions, epoch)
```



21.20. Testing and Evaluating our Model

```python
def test_model(model, x_test, y_test, target_vocab, save_path="best_model.pth"):
    """
    Evaluates the trained model on the test set and computes BLEU & Accuracy.

    - Loads the best saved model.
    - Runs inference on `x_test`.
    - Computes BLEU and Accuracy scores.
    - Prints the results.

    Args:
        model (nn.Module): The trained seq2seq model.
        x_test (torch.Tensor): Tensor containing test source sentences.
        y_test (list of lists): List where each element is a list of reference translations.
        target_vocab (Vocab): Vocabulary object to decode predictions.
        save_path (str): File path to save/load the best model.

    Returns:
        dict: Dictionary containing BLEU and Accuracy scores.
    """
    if os.path.exists(save_path):
        print(f"Loading best model from {save_path}...")
        model.load_state_dict(torch.load(save_path, map_location=device, weights_only=True))
```

```
    else:
        print("No saved model found! Using the last trained model.")

    model.eval()
    list_of_hypothesis = []
    list_of_references = []

    print("Running inference on the test set...")

    with torch.no_grad():
        predicted_sequences = model.forward(x_test.to(device))

        for pred, refs in zip(predicted_sequences, y_test):
            pred_clean = [
                target_vocab.index2word.get(idx, '<unk>')
                for idx in pred if target_vocab.index2word.get(idx, '<unk>') not in {'<sos>', '<eos>', '<pad>'}
            ]
            refs_clean = [
                [
                    target_vocab.index2word.get(idx.item(), '<unk>')
                    for idx in ref if target_vocab.index2word.get(idx.item(), '<unk>') not in {'<sos>',
  '<eos>', '<pad>'}
                ]
                for ref in refs
            ]

            list_of_hypothesis.append(pred_clean)
            list_of_references.append(refs_clean)

    scores = {name: func(list_of_references, list_of_hypothesis) for name, func in score_functions.items()}

    for i in range(min(15, len(list_of_hypothesis))):
        print(f"Predicted: {' '.join(list_of_hypothesis[i])}")
        print(f"References: {[' '.join(ref) for ref in list_of_references[i]]}\n")

    print("\nTest Set Evaluation:")
    for name, score in scores.items():
        print(f"{name}: {score:.4f}")

    return scores
```

```
save_path = "best_model.pth"
print(f"Saving best model to {save_path}...")
torch.save(best_model_wts, save_path)
```

```
Saving best model to best_model.pth...
```

```
test_scores = test_model(model, x_test, y_test, target_vocab, save_path)
```

```
Loading best model from best_model.pth...
Running inference on the test set...
Predicted: gyere velünk velünk !
References: ['gyere , egyél velünk .']

Predicted: segítenem kellett tominak .
References: ['segítenem kellett tominak .']

Predicted: nem nehéz , ha nehéz .
```

```
References: ['ez nem nehéz , ha tanulsz .']

Predicted: én csak számítottam .
References: ['egész idáig gondolkodtam .']

Predicted: tomnak nagyon ló a menedzser .
References: ['tominak eklektikus a zenei ízlése .']

Predicted: azt hiszem , hogy ellopta az ellopta a ruháimat .
References: ['azt hiszem , valaki ellopta a bőröndömet .']

[... 24 lines omitted ...]
Predicted: szerintem úgy vélem , meg vannak tenni .
References: ['szerintem szándékosan csinálták .', 'szerintem szánt szándékkel
tették .']


Test Set Evaluation:
BLEU-1: 0.5455
BLEU-2: 0.3483
BLEU-3: 0.2461
BLEU-4: 0.1792
Accuracy: 0.0996
```

21.21. Homework (Optional)

1. **Attention.** Replace the plain GRU decoder with a Bahdanau-attention decoder that conditions each output on a weighted sum of encoder hidden states. Report BLEU before and after.
2. **Scheduled sampling.** Introduce a linearly decaying teacher-forcing probability (starts at 1.0, ends at 0.5) over the training run. Report BLEU and qualitative translation quality on five test sentences.
3. **Beam search.** Implement a beam-search decoder with beam sizes 1 (greedy), 4 and 8. Report BLEU for each and explain the trade-off between beam size and latency.
4. **Longer sequences.** The current notebook clips sequences at 10 tokens. Retrain with a maximum length of 25 and report whether BLEU improves (and how training time changes).
5. **Subword tokenisation.** Replace the word-level vocabulary with Byte-Pair Encoding (BPE) or SentencePiece. Measure BLEU and the reduction in out-of-vocabulary tokens on the validation set.
6. **Different language pair.** Retrain the pipeline with a different source / target pair from Tatoeba (e.g. Hungarian -> English). Report BLEU and comment on any language-specific preprocessing needed (accented characters, word order).

Chapter 22

The Attention Mechanism

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1AjoDpt2NkPbMRNO36g2diEYQGwhkDqcG?usp=sharing

The **attention mechanism** is a fundamental component in modern deep learning models, enabling them to selectively focus on relevant parts of the input when making predictions. Originally introduced in sequence-to-sequence (`seq2seq`) models for **machine translation**, attention has since become a core feature of architectures like Transformers, significantly improving performance in tasks such as **language modeling, text generation, and even computer vision**.

In this notebook, we will explore the **power of attention** through a simple Transformer-based sequence-to-sequence (`seq2seq`) model, where the goal is to **predict the reversed version of an input sequence**. This task, while straightforward, highlights how attention allows the model to efficiently track dependencies across positions in a sequence, something that traditional recurrent networks often struggle with.

22.1. Necessary Imports

```
!pip install --quiet pytorch-lightning
```

```
import os
import numpy as np
import random
import math
import matplotlib
import gdown
import seaborn as sns
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim
import torchvision
import pytorch_lightning as pl
from torchvision import transforms
from IPython.display import Image as IMG
from pytorch_lightning.callbacks import ModelCheckpoint
from functools import partial
from tqdm.notebook import tqdm

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")

matplotlib.rcParams['lines.linewidth'] = 2.0  # Set the default line width for plots to 2.0 for better
   visibility
```

```
sns.reset_orig()                              # Reset Seaborn's settings to the original Matplotlib defaults
plt.set_cmap('cividis')                       # Set the default colormap for plots to 'cividis', which is
   colorblind-friendly
pl.seed_everything(42)                        # Set a global random seed (42) for reproducibility across
   different libraries
torch.backends.cudnn.deterministic = True     # Ensure deterministic behavior for CuDNN operations (useful for
   reproducibility)
torch.backends.cudnn.benchmark = False        # Disable CuDNN benchmarking to prevent adaptive optimizations
   that may affect reproducibility
```

```
INFO:lightning_fabric.utilities.seed:Seed set to 42
```

```
<Figure size 640x480 with 0 Axes>
```



22.2. Downloading Model Weights

Our pre-trained model is downloaded below. Ensure that your `MODEL_PATH` is correctly set before executing the code. This pre-trained model will be loaded into our Transformer model to bypass the training process and save time.

```
MODEL_PATH = "/content/saved_models/"
os.makedirs(MODEL_PATH, exist_ok=True)

GDRIVE_FILE_ID = "14l6Np8mfvxUPCUC_iROaiwZOX74vZNY9"
MODEL_FILENAME = "ReverseTask.ckpt"
DEST_PATH = os.path.join(MODEL_PATH, MODEL_FILENAME)

gdown.download(f"https://drive.google.com/uc?id={GDRIVE_FILE_ID}", MODEL_FILENAME, quiet=False)
os.rename(MODEL_FILENAME, DEST_PATH)

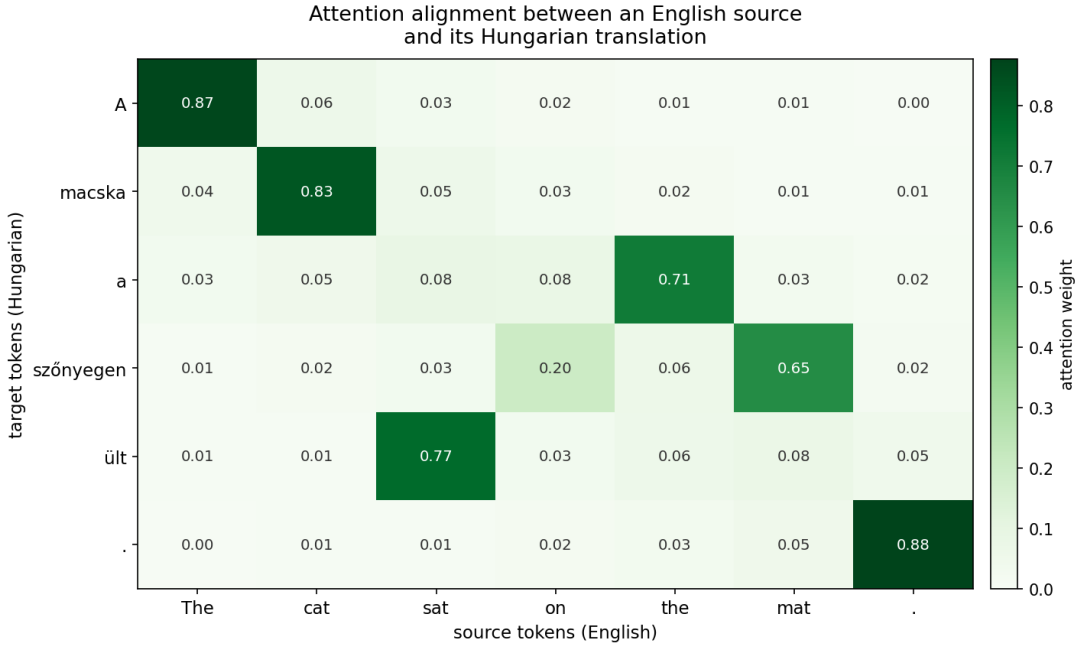
print(f"Model downloaded and saved to: {DEST_PATH}")
```

```
Downloading...
From: https://drive.google.com/uc?id=14l6Np8mfvxUPCUC_iROaiwZOX74vZNY9
To: /content/ReverseTask.ckpt
100%|##########| 45.8k/45.8k [00:00<00:00, 53.4MB/s]
```

```
Model downloaded and saved to: /content/saved_models/ReverseTask.ckpt
```

22.2.1. What attention looks like in practice

On a translation task, attention weights end up roughly aligning each target token with the source tokens that carry its meaning:



"macska" attends to "cat", "ült" attends to "sat", and the locative "szőnyegen" attends to both "mat" and "on" because Hungarian fuses them into a single inflected word. The diagonal-ish pattern is typical but not strict.

22.3. The Attention Mechanism

The most important part of a transformer is the **attention mechanism**.

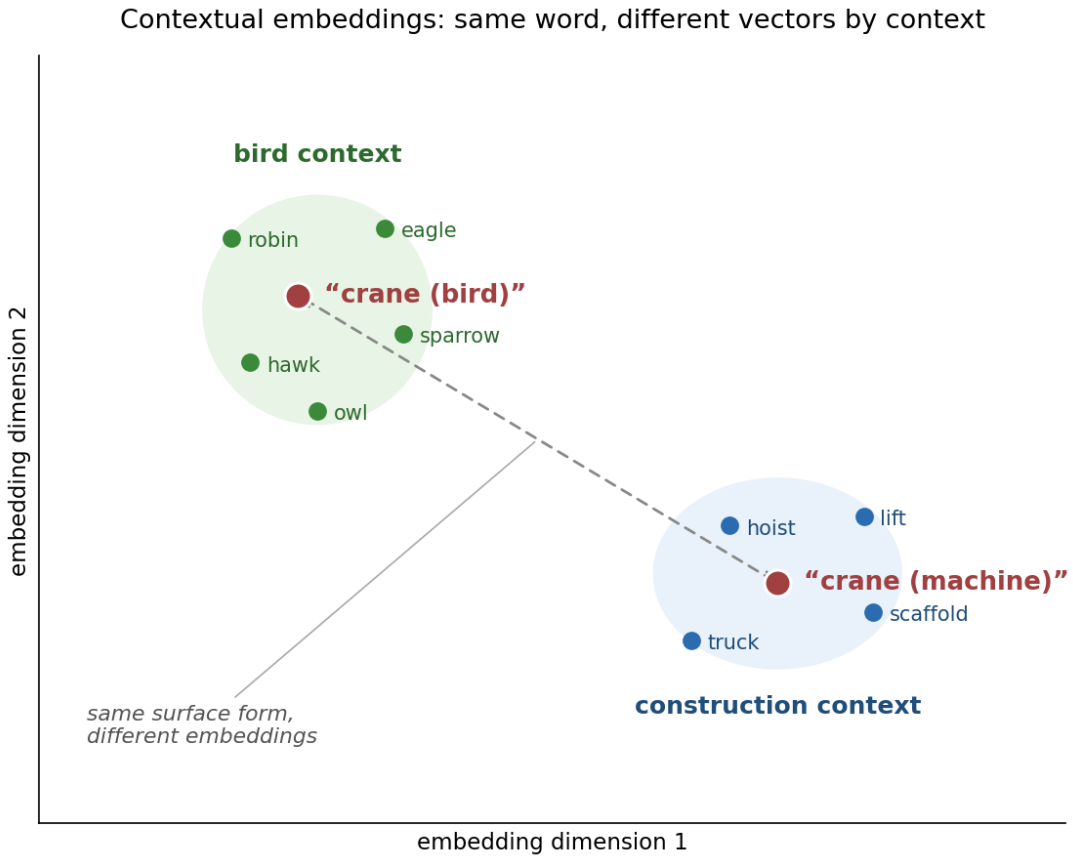
The **attention mechanism** describes a group of layers in neural networks that dynamically decide on which inputs we want to "attend" more than others.

To understand what is the attention, we will go through a simple example, and see what a good attention block shoud do. Let's take a look at these two sentences: 1. "The **crane** lifted the steel beam high above the construction site." 2. "A majestic **crane** stood gracefully by the edge of the lake."

When we tokenize a sentence and convert each token into a vector (**embedding**), a word like "crane" will initially get the same embedding in both sentences, something like `[0.2412, 0.44642, ..., 0.1235]`. This is because the embedding is **static**. However, in the first sentence, "crane" refers to a machine, and in the second, it refers to a bird.

A good attention mechanism helps resolve this ambiguity. It examines the context - the other words in the sentence - and adjusts the representation of "crane" accordingly. In the first sentence, **attention might shift the vector closer to embeddings** like construction, steel, or lift. In the second, it might lean toward bird, lake, or graceful.

In essence, **attention computes how much each surrounding word contributes to the final meaning of "crane"**, allowing the model to build context-aware representations.



In particular, an attention mechanism has usually **four parts** we need to specify:

- **Query**: The query is a feature vector that describes what we are looking for in the sequence, i.e. what would we maybe want to pay attention to.
- **Keys**: For each input element, we have a key which is again a feature vector. This feature vector roughly describes what the element is `offering`, or when it might be important. The keys should be designed such that we can identify the elements we want to pay attention to based on the query.
- **Values**: For each input element, we also have a value vector. This feature vector is the one we want to average over.
- **Score function**: To rate which elements we want to pay attention to, we need to specify a score function f_{attn}. The score function takes the query and a key as input, and output the score/attention weight of the query-key pair. It is usually implemented by simple similarity metrics like a dot product, or a small MLP.

22.4. Why scaled dot-product attention divides by $\sqrt{d_k}$

The scaled dot-product formula $\text{softmax}(QK^\top/\sqrt{d_k})V$ has a $\sqrt{d_k}$ denominator that is easy to skip past. It matters because of how softmax behaves in the high-dimensional regime.

If the entries of Q and K are drawn from a distribution with zero mean and unit variance, the dot product $q \cdot k$ of two such vectors of length d_k has zero mean and variance d_k. Its standard deviation is therefore $\sqrt{d_k}$. As d_k grows, individual scores in $QK^\top$ grow in magnitude. The softmax becomes sharp, putting

almost all its mass on a single position, and the gradient through the other positions becomes vanishingly small.

Dividing by $\sqrt{d_k}$ renormalises the scores to unit variance regardless of d_k, keeping the softmax in a useful regime where gradients flow through multiple positions. Without the scaling, training a transformer with large per-head dimensions is possible only with much smaller learning rates, and often not at all.

Variants that replace softmax entirely (linear attention, kernel attention) either do not need this scaling or fold an analogous normalisation into the kernel. Scaled dot-product is the idiom for the softmax-based case that dominates modern implementations.

22.4.1. Scaled Dot-Product Attention

The **core principle** behind self-attention is the **scaled dot-product attention**, which allows **each element in a sequence** to attend to every other element while maintaining **computational efficiency**.

The **attention mechanism** operates on three key matrices:
- **Queries**: $Q \in \mathbb{R}^{T \times d_k}$
- **Keys**: $K \in \mathbb{R}^{T \times d_k}$
- **Values**: $V \in \mathbb{R}^{T \times d_v}$

where:
- T is the **sequence length**,
- d_k is the **hidden dimensionality** for **queries and keys**,
- d_v is the **hidden dimensionality** for **values**.

For now, we omit the batch dimension for simplicity. The **attention score** for a given query-key pair is computed using the **dot product** as a similarity measure. The full **scaled dot-product attention** is expressed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

22.4.2. How Does It Work?

- The **matrix multiplication** QK^T computes the **dot product similarity** between **every query and key pair**, yielding a $T \times T$ **attention score matrix**.

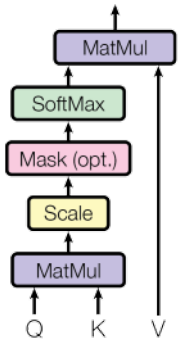
- Each row in this matrix represents the **attention logits** for a given query attending to all other elements.

- We **apply softmax** to normalize the scores, converting them into a **probability distribution**.

- The **final output** is obtained by computing a **weighted sum of the value vectors**, with weights determined by the **attention scores**.

A **computational visualization** is shown below (figure credit - Vaswani et al., 2017 [29]):

Scaled Dot-Product Attention



22.4.3. Why Do We Scale by $\frac{1}{\sqrt{d_k}}$?

The **scaling factor** $\frac{1}{\sqrt{d_k}}$ ensures that the variance of the attention values remains stable, preventing vanishing gradients and ensuring effective learning.

Consider that we initialize **queries** (Q) **and keys** (K) with a variance σ^2. When we take the **dot product** over two vectors, the result scales by d_k:

$$q_i \sim \mathcal{N}(0, \sigma^2), \quad k_i \sim \mathcal{N}(0, \sigma^2) \quad \Rightarrow \quad \text{Var}\left(\sum_{i=1}^{d_k} q_i \cdot k_i\right) = \sigma^4 \cdot d_k$$

Without **scaling**, the attention logits would have excessively **high variance**, causing the **softmax output to saturate**. This would lead to:
- A single token receiving **almost all** the attention weight.
- **Extremely small gradients**, making learning unstable.

By **scaling down** the dot-product values, we ensure that:
- The **softmax distribution** remains well-balanced.
- The **gradients remain non-zero**, enabling effective learning.

22.4.4. Masking in Attention

The **masking mechanism** (padding) is crucial when processing **variable-length sequences**. Since **batches** contain **sentences of different lengths**, we pad shorter sentences to a fixed size.

To **ignore padding tokens** during attention computation, we:
- **Apply a mask** to the attention matrix.
- **Set attention scores** for padded positions to a **large negative value** (e.g., -10^9).
- This ensures that **softmax assigns near-zero probability** to padded positions.

22.4.5. Causal (Look-Ahead) Masking

Causal masking is used to prevent a token from "seeing the future" during attention computation. It is essential in autoregressive tasks like language generation.

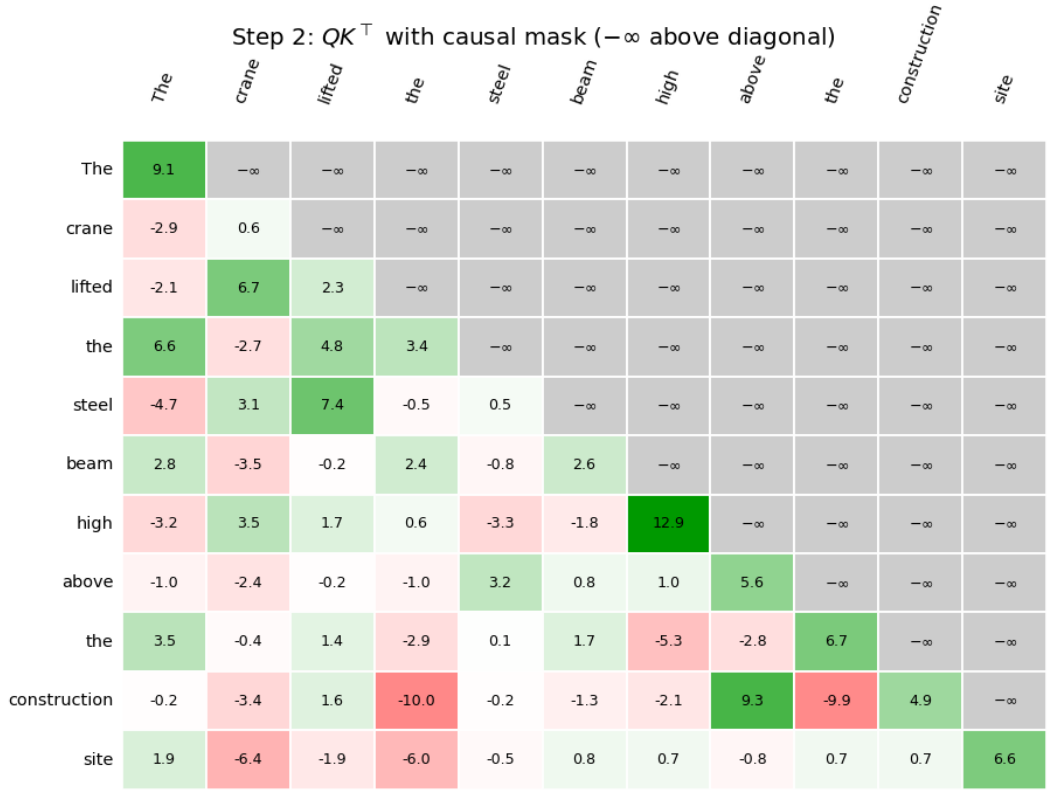
- Ensures that each position in the sequence can only attend to **previous or current tokens**, **not future tokens**.
- Implemented by setting attention scores for future positions to a **large negative value** (e.g., `-1e9` or $-\infty$).
- This causes the softmax to assign **near-zero probability** to those positions.
- Helps maintain the **left-to-right generation constraint** during training and inference.

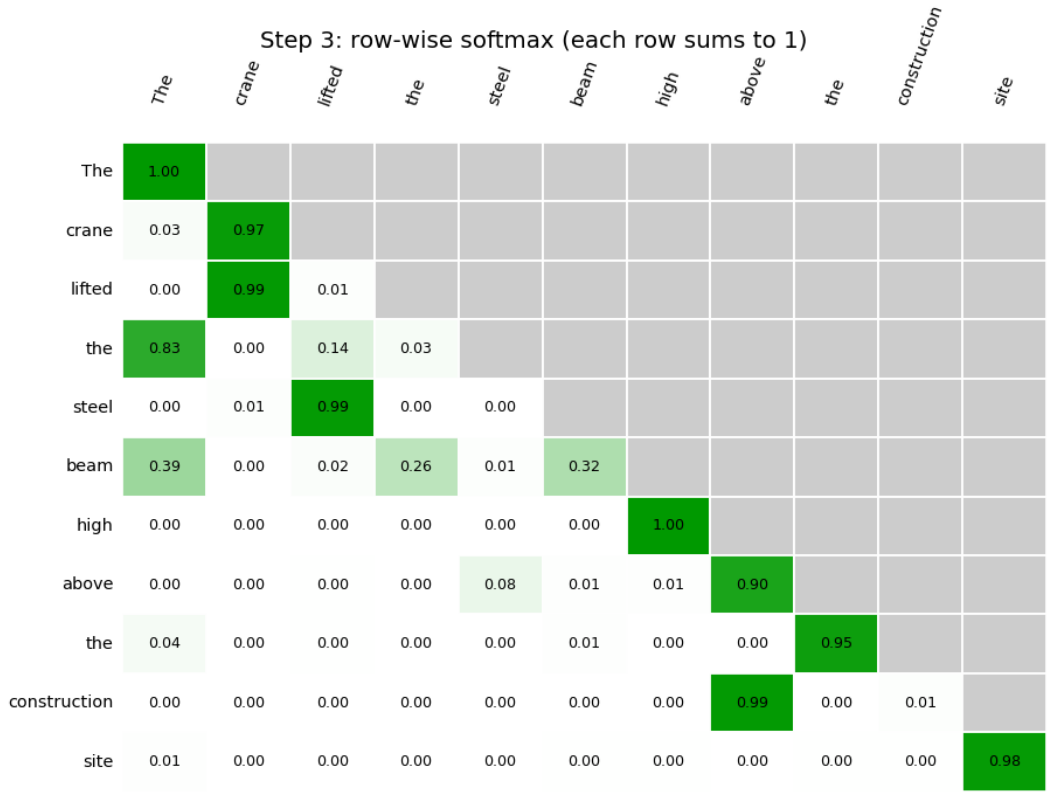
Example:
While generating the token at position `t`, the model can attend to tokens `[0, 1, ..., t]`, but **not** to `[t+1, t+2, ...]`.

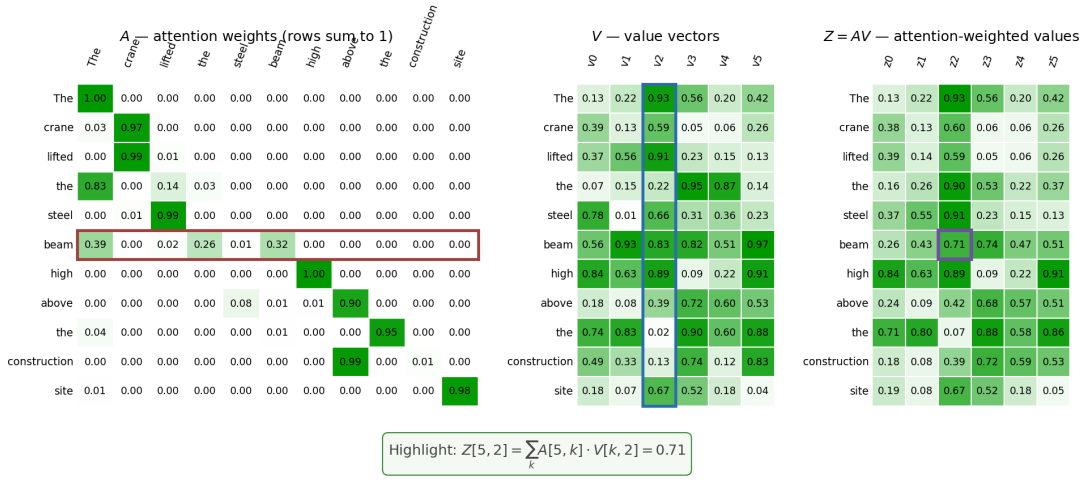
Now that we understand **scaled dot-product attention**, we can **implement** the function that computes the output features given the **query, key, and value matrices**.

Step 1: $QK^\top$ (raw attention scores)

	The	crane	lifted	the	steel	beam	high	above	the	construction	site
The	9.1	6.6	4.6	-3.9	-5.6	0.3	3.4	2.0	7.2	3.0	2.6
crane	-2.9	0.6	5.9	0.2	11.2	0.5	-1.7	-5.2	-3.1	8.6	-5.9
lifted	-2.1	6.7	2.3	-1.0	-0.9	1.7	-1.7	1.1	0.2	1.7	0.9
the	6.6	-2.7	4.8	3.4	-3.8	4.8	-1.8	-1.6	-5.6	-8.4	2.5
steel	-4.7	3.1	7.4	-0.5	0.5	1.6	3.0	-1.0	0.1	5.3	5.1
beam	2.8	-3.5	-0.2	2.4	-0.8	2.6	-3.1	-2.5	-2.7	-1.8	4.6
high	-3.2	3.5	1.7	0.6	-3.3	-1.8	12.9	0.4	2.2	2.7	4.2
above	-1.0	-2.4	-0.2	-1.0	3.2	0.8	1.0	5.6	4.9	-2.2	-1.9
the	3.5	-0.4	1.4	-2.9	0.1	1.7	-5.3	-2.8	6.7	9.0	1.8
construction	-0.2	-3.4	1.6	-10.0	-0.2	-1.3	-2.1	9.3	-9.9	4.9	0.3
site	1.9	-6.4	-1.9	-6.0	-0.5	0.8	0.7	-0.8	0.7	0.7	6.6


Step 2: $QK^\top$ with causal mask ($-\infty$ above diagonal)
The crane lifted the steel beam high above the construction site
The 9.1 −∞ −∞ −∞ −∞ −∞ −∞ −∞ −∞ −∞ −∞
crane -2.9 0.6 −∞ −∞ −∞ −∞ −∞ −∞ −∞ −∞ −∞
lifted -2.1 6.7 2.3 −∞ −∞ −∞ −∞ −∞ −∞ −∞ −∞
the 6.6 -2.7 4.8 3.4 −∞ −∞ −∞ −∞ −∞ −∞ −∞
steel -4.7 3.1 7.4 -0.5 0.5 −∞ −∞ −∞ −∞ −∞ −∞
beam 2.8 -3.5 -0.2 2.4 -0.8 2.6 −∞ −∞ −∞ −∞ −∞
high -3.2 3.5 1.7 0.6 -3.3 -1.8 12.9 −∞ −∞ −∞ −∞
above -1.0 -2.4 -0.2 -1.0 3.2 0.8 1.0 5.6 −∞ −∞ −∞
the 3.5 -0.4 1.4 -2.9 0.1 1.7 -5.3 -2.8 6.7 −∞ −∞
construction -0.2 -3.4 1.6 -10.0 -0.2 -1.3 -2.1 9.3 -9.9 4.9 −∞
site 1.9 -6.4 -1.9 -6.0 -0.5 0.8 0.7 -0.8 0.7 0.7 6.6


Step 3: row-wise softmax (each row sums to 1)
The crane lifted the steel beam high above the construction site
The 1.00
crane 0.03 0.97
lifted 0.00 0.99 0.01
the 0.83 0.00 0.14 0.03
steel 0.00 0.01 0.99 0.00 0.00
beam 0.39 0.00 0.02 0.26 0.01 0.32
high 0.00 0.00 0.00 0.00 0.00 0.00 1.00
above 0.00 0.00 0.00 0.00 0.08 0.01 0.01 0.90
the 0.04 0.00 0.00 0.00 0.00 0.01 0.00 0.00 0.95
construction 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.99 0.00 0.01
site 0.01 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.98



In terms of the sizes of the $ W_q $, $ W_k $, and $ W_v $ matrices, the **Query** and **Key** matrices typically have dimensions:

$$W_q, W_k \in \mathbb{R}^{d_{\text{emb}} \times d_{\text{kq}}}$$

where: - d_{emb} is the embedding dimension, - d_{kq} is the key-query space size.

The **Value** matrix usually has dimensions:

$$W_v \in \mathbb{R}^{d_{\text{emb}} \times d_{\text{emb}}}$$

However, this would be computationally expensive due to the large embedding dimension. To mitigate this, the **Value matrix** is factorized into two smaller matrices:

$$W_v = W_{v1} W_{v2}$$

where:

$$W_{v1} \in \mathbb{R}^{d_{\text{emb}} \times d_{\text{kq}}}, \quad W_{v2} \in \mathbb{R}^{d_{\text{kq}} \times d_{\text{emb}}}$$

This factorization reduces the computational complexity while maintaining the necessary transformations in self-attention.

```
def scaled_dot_product(q, k, v, mask=None):
    # q, k, v: Query, Key, and Value tensors respectively
    # mask: Optional tensor to mask certain positions (e.g., for padding or autoregressive tasks)

    # Get the dimensionality of the query vectors (last dimension of q)
    d_k = q.size()[-1]

    # Compute the dot product between queries (q) and transposed keys (k)
    # Resulting shape: [batch_size, num_heads, seq_length, seq_length]
    # Each entry represents the compatibility (attention score) between two tokens
    attn_logits = torch.matmul(q, k.transpose(-2, -1))
```

```
    # Scale the attention logits by the square root of d_k to prevent overly large values
    # This is important for stable gradient flow during backpropagation
    attn_logits = attn_logits / math.sqrt(d_k)

    if mask is not None:
        # If a mask is provided, use it to set positions that should be ignored to a very large negative value
        # This ensures these positions receive negligible attention after applying softmax
        attn_logits = attn_logits.masked_fill(mask == 0, -9e15)

    # Apply the softmax function along the last dimension (seq_length) to normalize attention scores
    # This converts logits into probabilities, where each row sums to 1
    attention = F.softmax(attn_logits, dim=-1)

    # Compute the weighted sum of values (v), where weights are the attention probabilities
    # Resulting shape: [batch_size, num_heads, seq_length, head_dim]
    values = torch.matmul(attention, v)

    # Return the weighted values (output of the attention mechanism) and the attention scores
    return values, attention
```

Our implementation is designed to handle **any additional dimensionality** preceding the sequence length, allowing it to support batched inputs seamlessly. However, to better illustrate the attention mechanism, let's generate a **few random query, key, and value vectors** and compute the corresponding attention outputs.

```
seq_len, d_k = 3, 2

q = torch.randn(seq_len, d_k)
k = torch.randn(seq_len, d_k)
v = torch.randn(seq_len, d_k)
values, attention = scaled_dot_product(q, k, v)
print("Q\n", q)
print("K\n", k)
print("V\n", v)
print("Values\n", values)
print("Attention\n", attention)
```

```
Q
 tensor([[ 0.3367,  0.1288],
        [ 0.2345,  0.2303],
        [-1.1229, -0.1863]])
K
 tensor([[ 2.2082, -0.6380],
        [ 0.4617,  0.2674],
        [ 0.5349,  0.8094]])
V
 tensor([[ 1.1103, -1.6898],
        [-0.9890,  0.9580],
        [ 1.3221,  0.8172]])
Values
 tensor([[ 0.5698, -0.1520],
        [ 0.5379, -0.0265],
        [ 0.2246,  0.5556]])
Attention
 tensor([[0.4028, 0.2886, 0.3086],
        [0.3538, 0.3069, 0.3393],
        [0.1303, 0.4630, 0.4067]])
```

Before continuing, make sure **you can follow the calculation of the specific values here**, and also check it by hand. It is important to fully understand how the scaled dot product attention is calculated.

22.4.6. Multi-Head Attention

The **scaled dot-product attention** enables a network to **attend** over a sequence, but in many cases, a single attention mechanism is insufficient. Different elements in a sequence may need to focus on **multiple aspects** of the input, and a single weighted average may not effectively capture all the necessary relationships.

To address this, **multi-head attention** extends the **attention mechanism** by applying **multiple independent attention heads**, each with its own query-key-value transformation. Specifically, we:

1. Transform the **query (Q), key (K), and value (V) matrices** into h **independent** subspaces.
2. Apply the **scaled dot-product attention** separately for each head.
3. **Concatenate** the outputs of all heads.
4. Pass the concatenated result through a **final linear projection** to obtain the final representation.

Mathematically, this can be expressed as:

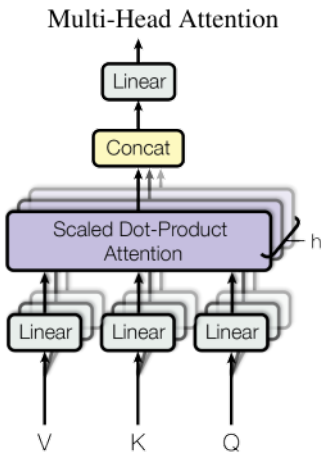
$$\begin{aligned}\text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, ..., \text{head}_h)W^O \\ \text{where head}_i &= \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)\end{aligned}$$

where the **learnable parameters** are:

- $W_{1...h}^Q \in \mathbb{R}^{D \times d_k}$ – Transforming the query vector.
- $W_{1...h}^K \in \mathbb{R}^{D \times d_k}$ – Transforming the key vector.
- $W_{1...h}^V \in \mathbb{R}^{D \times d_v}$ – Transforming the value vector.
- $W^O \in \mathbb{R}^{h \cdot d_v \times d_{\text{out}}}$ – Final output projection.

Here, D is the **input dimensionality**, while d_k and d_v represent the **key-query space size** and **value space size**, respectively.

Below is a **computational visualization** of multi-head attention (figure credit - Vaswani et al., 2017):



22.4.7. How is Multi-Head Attention Applied in Neural Networks?

In a practical **neural network**, we may not always have explicit **query, key, and value vectors** as input. Instead, we typically **derive them from feature representations** in the network.

A simple but effective approach is to use the **current feature map** of a neural network:

- Let $X \in \mathbb{R}^{B \times T \times d_{\text{model}}}$ be the **feature representation**,
  - where B is the **batch size**,
  - T is the **sequence length**,
  - d_{model} is the **hidden dimensionality** of X.
- We set X as the **query (Q), key (K), and value (V)** inputs.
- We apply **separate weight matrices** (W^Q, W^K, W^V) to transform X into query, key, and value vectors.

This formulation allows us to **parallelize attention computation** efficiently while maintaining **different contextual interpretations** across multiple heads.

Multi-head attention extends the standard attention mechanism by **stacking multiple attention layers**, where each head operates with its own **separate query, key, and value matrices**. This allows the model to **capture different contextual relationships** within the input sequence.

At the end, each head **proposes a vector for every word**, which is then **integrated into the original embedding**. Instead of summing these proposals explicitly, they are **concatenated for efficiency** before being transformed back into the original embedding space through a final projection layer.

```
# Helper function to support different mask shapes.
# Output shape supports (batch_size, number of heads, seq length, seq length)
# If 2D: broadcasted over batch size and number of heads
# If 3D: broadcasted over number of heads
```

```
# If 4D: leave as is
def expand_mask(mask):
    assert mask.ndim >= 2, "Mask must be at least 2-dimensional with seq_length x seq_length"
    if mask.ndim == 3:
        mask = mask.unsqueeze(1)
    while mask.ndim < 4:
        mask = mask.unsqueeze(0)
    return mask
```

```
class MultiheadAttention(nn.Module):
    def __init__(self, input_dim, embed_dim, num_heads):
        super().__init__()  # Initializes the parent class (nn.Module)
        # Ensure the embedding dimension is divisible by the number of heads
        assert embed_dim % num_heads == 0, "Embedding dimension must be 0 modulo number of heads."

        self.embed_dim = embed_dim  # Dimensionality of input and output tensors
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads  # Dimension of each individual head (embed_dim divided by
  num_heads)

        # Linear layer to project input into queries, keys, and values
        self.qkv_proj = nn.Linear(input_dim, 3 * embed_dim)
        # Linear layer to project concatenated output of all heads back to the embedding dimension
        self.o_proj = nn.Linear(embed_dim, embed_dim)

        self._reset_parameters()  # Initialize weights of the linear layers

    def _reset_parameters(self):
        # Initialize weights using Xavier uniform distribution (commonly used in Transformer models)
        nn.init.xavier_uniform_(self.qkv_proj.weight)
        self.qkv_proj.bias.data.fill_(0)  # Initialize biases to zero
        nn.init.xavier_uniform_(self.o_proj.weight)
        self.o_proj.bias.data.fill_(0)  # Initialize biases to zero

    def forward(self, x, mask=None, return_attention=False):
        # Input x: [batch_size, seq_length, input_dim]
        batch_size, seq_length, _ = x.size()  # Extract batch size and sequence length from input

        if mask is not None:
            mask = expand_mask(mask)  # Optionally expand mask for attention mechanism compatibility

        # Project input x into queries, keys, and values (Q, K, V) using the linear layer
        qkv = self.qkv_proj(x)

        # Reshape to separate heads: [batch_size, seq_length, num_heads, 3 * head_dim]
        qkv = qkv.reshape(batch_size, seq_length, self.num_heads, 3 * self.head_dim)
        # Permute dimensions for attention calculation: [batch_size, num_heads, seq_length, 3 * head_dim]
        qkv = qkv.permute(0, 2, 1, 3)
        # Split into three tensors: queries (q), keys (k), and values (v)
        q, k, v = qkv.chunk(3, dim=-1)  # Divide the last dimension into three equal parts

        # Compute scaled dot-product attention
        values, attention = scaled_dot_product(q, k, v, mask=mask)
        # values: [batch_size, num_heads, seq_length, head_dim]
        # attention: attention scores (optional)

        # Rearrange values to merge heads back: [batch_size, seq_length, num_heads, head_dim]
        values = values.permute(0, 2, 1, 3)
        # Reshape to combine heads into a single embedding dimension: [batch_size, seq_length, embed_dim]
        values = values.reshape(batch_size, seq_length, self.embed_dim)
```

```
        # Project the combined values back to the embedding space
        o = self.o_proj(values)

        if return_attention:
            return o, attention  # Optionally return attention scores along with the output
        else:
            return o  # Return only the output
```

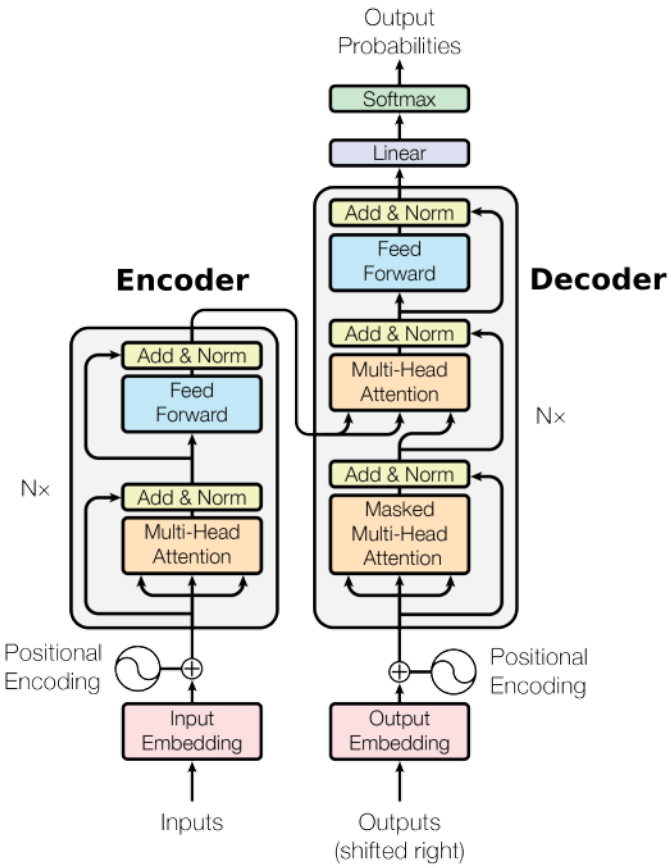
22.4.8. Transformer Encoder

Next, we will explore how **multi-head attention** is integrated into the **Transformer encoder architecture**. Originally, the Transformer model was designed for **machine translation**, following an **encoder-decoder** structure. The **encoder** processes the input sentence in the original language, generating an attention-based representation, while the **decoder** attends over this encoded representation and generates the translated sentence **autoregressively**, similar to an **RNN-based model**.

Although this structure is crucial for **sequence-to-sequence tasks**, in this tutorial, we will **focus primarily on the encoder**. Many advances in NLP have been made using **encoder-only** Transformer models, including:

- **BERT** – widely used for text classification, question answering, and other NLP tasks.
- **Vision Transformer (ViT)** – applies Transformer architecture to image processing.
- **Other transformer variants** optimized for specific domains.

If you **understand the encoder**, implementing the decoder is a small additional step. Below is the full Transformer architecture (figure credit - Vaswani et al., 2017):



22.4.9. Structure of the Transformer Encoder

The **encoder** consists of **N identical blocks**, applied sequentially. Given an input x, it is first processed through a **Multi-Head Attention** block, followed by:

1. **Residual Connection:** Adds the original input back to the attention output.

2. **Layer Normalization:** Normalizes the sum to stabilize training.

The overall computation is expressed as:

$$\text{LayerNorm}(x + \text{MultiHead}(x, x, x))$$

where x serves as the **query (Q), key (K), and value (V)** inputs to the attention layer.

22.4.10. Why Residual Connections?

Residual connections are crucial in the Transformer architecture for two key reasons:

1. **Enabling Deep Networks** – Similar to **ResNets**, Transformers can contain **24+ encoder layers**, and residual connections allow gradients to flow smoothly through these deep networks.

2. **Preserving Input Information** – Multi-head attention **ignores positional information**, relying only on learned embeddings. Without residual connections, information about the original sequence would be **lost** after the first attention layer.

   Alternative Approach? One alternative would be to **dedicate a head** in the attention mechanism to preserve the original input, but this would be inefficient and **would not improve gradient flow**.

22.4.11. Importance of Layer Normalization

Layer Normalization (LayerNorm) plays an essential role in Transformers by:

- **Improving training stability** – speeds up convergence.
- **Providing slight regularization** – prevents overfitting.
- **Ensuring feature consistency** – keeps activations at a similar magnitude.

Unlike **Batch Normalization (BatchNorm)**, **LayerNorm** is preferred in NLP because:

1. **Batch sizes in Transformers are often small** due to high memory requirements.
2. **BatchNorm struggles with high-variance features**, especially in text, where **rare words** can distort normalization statistics.

22.4.12. Feed-Forward Network (FFN) in Transformers

In addition to **Multi-Head Attention**, each encoder block contains a **fully connected feed-forward network (FFN)**, which is **applied independently to each position** in the sequence. The FFN consists of:

- A **Linear layer** -> Expands feature dimensions.
- A **ReLU activation** -> Introduces non-linearity.
- Another **Linear layer** -> Projects features back to original dimensions.

The **transformation formula** with **residual connection** is:

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$
$$x = \text{LayerNorm}(x + \text{FFN}(x))$$

The **MLP (FFN) expands features 2-8× larger** than the original input x, allowing the model to **process information more effectively**.

Why not a deeper MLP? Instead of multiple small layers, **wider layers** enable faster parallel execution on modern hardware.

22.4.13. Implementation Considerations

Now that we understand the **components of the Transformer encoder**, we can start implementing it. The **key architectural choices** include:

- **Dropout layers** – Applied to both Multi-Head Attention and Feed-Forward Network to improve generalization.

- **Stacking multiple encoder blocks** – Enhances the ability to **capture complex relationships** in the input sequence.

Let's proceed with implementing a **single encoder block**, followed by **stacking multiple blocks to form the full encoder**.

```
class EncoderBlock(nn.Module):

    def __init__(self, input_dim, num_heads, dim_feedforward, dropout=0.0):
        """
        Initializes the EncoderBlock.

        Parameters:
        - input_dim: The dimensionality of the input embeddings.
        - num_heads: The number of attention heads in the multi-head attention mechanism.
        - dim_feedforward: The dimensionality of the feedforward network's hidden layer.
        - dropout: Dropout probability to use for regularization.
        """
        super().__init__()  # Initializes the parent nn.Module

        # Multi-head self-attention mechanism
        self.self_attn = MultiheadAttention(input_dim, input_dim, num_heads)

        # Two-layer feedforward network (MLP with ReLU activation)
        self.linear_net = nn.Sequential(
            nn.Linear(input_dim, dim_feedforward),  # First linear transformation
            nn.Dropout(dropout),  # Apply dropout for regularization
            nn.ReLU(inplace=True),  # Apply ReLU activation
            nn.Linear(dim_feedforward, input_dim)  # Project back to the input dimensionality
        )

        # Layer normalization for stability and better gradient flow
        self.norm1 = nn.LayerNorm(input_dim)  # Normalization after the attention mechanism
        self.norm2 = nn.LayerNorm(input_dim)  # Normalization after the feedforward network

        # Dropout layer to apply dropout between sub-layers
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        """
        Forward pass through the encoder block.

        Parameters:
        - x: Input tensor of shape [batch_size, seq_length, input_dim].
        - mask: Optional attention mask to ignore certain positions.

        Returns:
```

```
        - x: The output tensor of shape [batch_size, seq_length, input_dim].
        """
        # Apply the self-attention mechanism
        attn_out = self.self_attn(x, mask=mask)  # Self-attention with optional masking
        # Add residual connection (skip connection) and apply dropout
        x = x + self.dropout(attn_out)
        # Normalize the result
        x = self.norm1(x)

        # Apply the feedforward network (MLP)
        linear_out = self.linear_net(x)  # Pass through the MLP
        # Add residual connection (skip connection) and apply dropout
        x = x + self.dropout(linear_out)
        # Normalize the result
        x = self.norm2(x)

        # Return the final output
        return x
```

Based on the implementation of the **Transformer block**, we can now construct the **full Transformer encoder module**. This module includes:

1. A **forward function** that **iterates through the sequence of encoder blocks**.

2. A **get_attention_maps function** that **returns the attention probabilities** from all **Multi-Head Attention layers** in the encoder.

The **get_attention_maps** function is particularly useful for **visualizing attention weights** and gaining insight into how the model processes input sequences. However, it is important to note that attention maps should be interpreted **cautiously**.

Several studies have highlighted that **attention weights do not always offer a faithful interpretation of model decisions**. For example:
- **Attention is not Explanation** argues that attention does not provide a reliable justification for model behavior.
- **Attention is not not Explanation** suggests that while attention alone is not a perfect explanation, it can still offer useful insights under certain conditions.

Thus, while attention maps can help **understand** the model, they should **not be solely relied upon for interpretability**.

```
class TransformerEncoder(nn.Module):

    def __init__(self, num_layers, **block_args):
        """
        Initializes the TransformerEncoder.

        Parameters:
        - num_layers: Number of encoder blocks to stack in the encoder.
        - block_args: Additional arguments to configure each EncoderBlock, such as input_dim, num_heads, etc.
        """
        super().__init__()  # Initializes the parent nn.Module

        # Create a stack of `num_layers` EncoderBlock layers
        self.layers = nn.ModuleList([EncoderBlock(**block_args) for _ in range(num_layers)])
```

```
    def forward(self, x, mask=None):
        """
        Forward pass through the TransformerEncoder.

        Parameters:
        - x: Input tensor of shape [batch_size, seq_length, input_dim].
        - mask: Optional attention mask to ignore certain positions (e.g., padding).

        Returns:
        - x: Output tensor of shape [batch_size, seq_length, input_dim] after processing through all layers.
        """
        for l in self.layers:  # Iterate over each encoder block
            x = l(x, mask=mask)  # Pass the input through the current block
        return x  # Return the final output after all layers

    def get_attention_maps(self, x, mask=None):
        """
        Extracts the attention maps from all encoder layers.

        Parameters:
        - x: Input tensor of shape [batch_size, seq_length, input_dim].
        - mask: Optional attention mask.

        Returns:
        - attention_maps: A list of attention maps from each encoder layer.
        """
        attention_maps = []  # Initialize a list to store attention maps
        for l in self.layers:  # Iterate over each encoder block
            # Extract the attention map from the current layer's self-attention mechanism
            _, attn_map = l.self_attn(x, mask=mask, return_attention=True)
            attention_maps.append(attn_map)  # Store the attention map
            # Pass the input through the current encoder block
            x = l(x)
        return attention_maps  # Return all attention maps
```

22.4.14. Positional Encoding

The **Multi-Head Attention block** is **permutation-equivariant**, meaning it does not inherently recognize the **order of elements** in a sequence. However, in tasks like **language processing**, word order is **crucial** for understanding meaning.

To incorporate **positional information**, we have two main options:
1. **Learn an embedding for each position** – but this would not generalize to variable sequence lengths.
2. **Use a deterministic function-based encoding** – which allows the model to generalize to **unseen sequence lengths**.

The Transformer model uses a **sinusoidal positional encoding**, defined as:

$$PE_{(pos,i)} = \begin{cases} \sin\left(\frac{pos}{10000^{i/d_{\text{model}}}}\right) & \text{if } i \bmod 2 = 0 \\ \cos\left(\frac{pos}{10000^{(i-1)/d_{\text{model}}}}\right) & \text{otherwise} \end{cases}$$

where:
- $PE_{(pos,i)}$ represents the positional encoding at position pos for dimension i.

- **Even indices** (i **mod** $2 = 0$) use a **sine function**.
- **Odd indices** (i **mod** $2 = 1$) use a **cosine function**.

These values are **added to the input embeddings**, providing the model with **positional context**. The encoding ensures that relative positions **can be inferred linearly**, enabling the model to generalize across different sequence lengths.

The wavelength of these sinusoidal functions **ranges from** 2π **to** $10000 \cdot 2\pi$, allowing the model to capture both **short-term and long-term dependencies**.

Below is the implementation of **positional encoding**, adapted from the official **PyTorch Transformer tutorial**.

```
class PositionalEncoding(nn.Module):

    def __init__(self, d_model, max_len=5000):
        """
        Initializes the PositionalEncoding module.

        Parameters:
        - d_model: Dimensionality of the input embeddings (e.g., embedding size).
        - max_len: Maximum length of input sequences this module can handle.
        """
        super().__init__()  # Initializes the parent nn.Module

        # Create a tensor to store positional encodings
        # Shape: [max_len, d_model], where max_len is the maximum sequence length
        pe = torch.zeros(max_len, d_model)

        # Compute the positions for each token in the sequence: [max_len, 1]
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)

        # Compute the scaling term for each dimension of the positional encoding
        # Shape: [d_model / 2], as we alternate between sine and cosine
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))

        # Assign sine values to even dimensions: [:, 0, 2, 4, ...]
        pe[:, 0::2] = torch.sin(position * div_term)
        # Assign cosine values to odd dimensions: [:, 1, 3, 5, ...]
        pe[:, 1::2] = torch.cos(position * div_term)

        # Add a batch dimension: Shape becomes [1, max_len, d_model]
        pe = pe.unsqueeze(0)

        # Register the positional encoding tensor as a buffer
        # Buffers are not trainable but are part of the model state (e.g., moved to the appropriate device)
        self.register_buffer('pe', pe, persistent=False)

    def forward(self, x):
        """
        Adds positional encoding to the input tensor.

        Parameters:
        - x: Input tensor of shape [batch_size, seq_length, d_model].

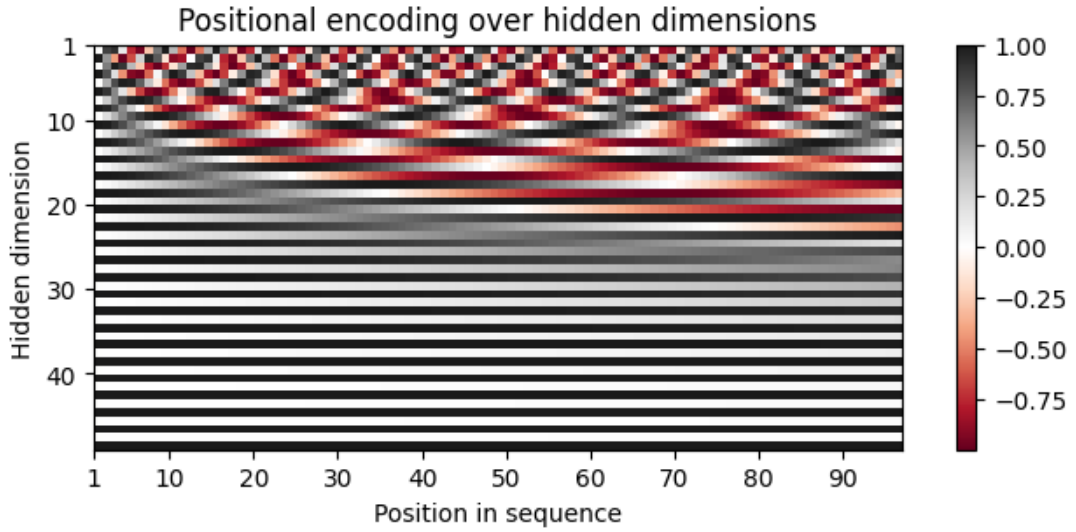
        Returns:
        - x: Output tensor with positional encodings added, same shape as input.
        """
        # Add the positional encoding (self.pe) to the input tensor
```

```
        # Note: self.pe is indexed to match the sequence length (x.size(1))
        x = x + self.pe[:, :x.size(1)]
        return x
```

To better understand **positional encoding**, we can visualize how it varies across **hidden dimensions** and **sequence positions**. Each pixel in the visualization represents how the **input features** are modified to encode **positional information**. Let's generate this visualization below.

```
encod_block = PositionalEncoding(d_model=48, max_len=96)
pe = encod_block.pe.squeeze().T.cpu().numpy()

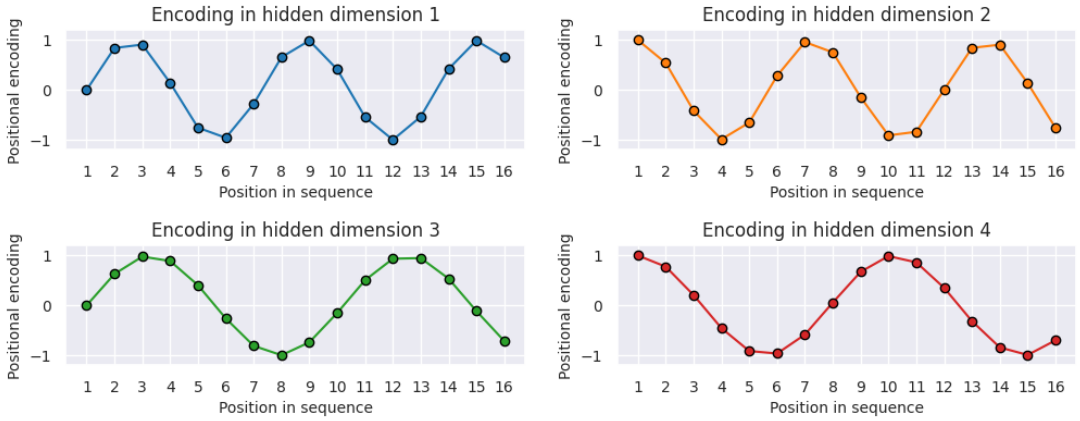
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(8,3))
pos = ax.imshow(pe, cmap="RdGy", extent=(1,pe.shape[1]+1,pe.shape[0]+1,1))
fig.colorbar(pos, ax=ax)
ax.set_xlabel("Position in sequence")
ax.set_ylabel("Hidden dimension")
ax.set_title("Positional encoding over hidden dimensions")
ax.set_xticks([1]+[i*10 for i in range(1,1+pe.shape[1]//10)])
ax.set_yticks([1]+[i*10 for i in range(1,1+pe.shape[0]//10)])
plt.show()
```



The **sine and cosine waves** with varying wavelengths are clearly visible, encoding positional information across different **hidden dimensions**. To better understand this pattern, we can examine the positional encoding for specific hidden dimensions. Below, we visualize the encoding for **hidden dimensions 1, 2, 3, and 4**.

```
sns.set_theme()
fig, ax = plt.subplots(2, 2, figsize=(12,4))
ax = [a for a_list in ax for a in a_list]
for i in range(len(ax)):
    ax[i].plot(np.arange(1,17), pe[i,:16], color=f'C{i}', marker="o", markersize=6, markeredgecolor="black")
    ax[i].set_title(f"Encoding in hidden dimension {i+1}")
    ax[i].set_xlabel("Position in sequence", fontsize=10)
    ax[i].set_ylabel("Positional encoding", fontsize=10)
    ax[i].set_xticks(np.arange(1,17))
    ax[i].tick_params(axis='both', which='major', labelsize=10)
    ax[i].tick_params(axis='both', which='minor', labelsize=8)
```

```
    ax[i].set_ylim(-1.2, 1.2)
fig.subplots_adjust(hspace=0.8)
sns.reset_orig()
plt.show()
```



We observe that the patterns for **hidden dimensions 1 and 2** differ only in their **starting phase**, while both share a wavelength of 2π, repeating approximately every **6 positions**. In contrast, **hidden dimensions 3 and 4** exhibit a wavelength that is roughly **twice as long**, demonstrating how different frequencies encode positional information.

```
class CosineWarmupScheduler(optim.lr_scheduler._LRScheduler):

    def __init__(self, optimizer, warmup, max_iters):
        self.warmup = warmup
        self.max_num_iters = max_iters
        super().__init__(optimizer)

    def get_lr(self):
        lr_factor = self.get_lr_factor(epoch=self.last_epoch)
        return [base_lr * lr_factor for base_lr in self.base_lrs]

    def get_lr_factor(self, epoch):
        lr_factor = 0.5 * (1 + np.cos(np.pi * epoch / self.max_num_iters))
        if epoch <= self.warmup:
            lr_factor *= epoch * 1.0 / self.warmup
        return lr_factor
```

22.4.15. PyTorch Lightning Module

To integrate the **Transformer architecture** into a structured training framework, we will implement it within a **PyTorch Lightning** module. **PyTorch Lightning** simplifies training, validation, and testing by organizing code into well-structured functions.

We will design a **template for a Transformer-based classifier** that provides an **output per sequence element**. If classification over the entire sequence is required, a common approach is to introduce a `[CLS]` token at the beginning, which serves as a dedicated classification representation. However, in this implementation, we focus on tasks where predictions are made for each individual sequence element.

Beyond the **Transformer encoder**, our module includes: - **An input network**: Maps input dimensions to the model's hidden dimensions. - **Positional encoding**: Injects positional information into the sequence. - **An output network**: Converts encoded representations into predictions.

Additionally, we integrate a **learning rate scheduler** that updates the learning rate at each iteration rather than once per epoch. This is essential for implementing **learning rate warmup** followed by **cosine decay**, ensuring smooth and stable optimization.

The **training, validation, and testing steps** will be defined based on specific tasks and datasets.

```
class TransformerPredictor(pl.LightningModule):

    def __init__(self, input_dim, model_dim, num_classes, num_heads, num_layers, lr, warmup, max_iters,
  dropout=0.0, input_dropout=0.0):
        """
        Inputs:
            input_dim - Hidden dimensionality of the input (or number of input features)
            model_dim - Hidden dimensionality to use inside the Transformer
            num_classes - Number of classes to predict per sequence element
            num_heads - Number of heads to use in the Multi-Head Attention blocks
            num_layers - Number of encoder blocks to use.
            lr - Learning rate for the optimizer
            warmup - Number of warmup steps (e.g., 50 to 500)
            max_iters - Total number of iterations to train for (needed for the cosine warmup scheduler)
            dropout - Dropout probability inside the Transformer and feedforward layers
            input_dropout - Dropout probability to apply on the input features
        """
        super().__init__()
        self.save_hyperparameters()
        self._create_model()

    def _create_model(self):
        # Map input to model dimensionality (acts similar to an embedding layer)
        self.input_net = nn.Sequential(
            nn.Dropout(self.hparams.input_dropout),
            nn.Linear(self.hparams.input_dim, self.hparams.model_dim)
        )
        # Add positional encoding to inject token order information
        self.positional_encoding = PositionalEncoding(d_model=self.hparams.model_dim)
        # Build the Transformer encoder using our EncoderBlock(s)
        self.transformer = TransformerEncoder(
            num_layers=self.hparams.num_layers,
            input_dim=self.hparams.model_dim,
            dim_feedforward=2 * self.hparams.model_dim,
            num_heads=self.hparams.num_heads,
            dropout=self.hparams.dropout
        )
        # Output network to map from model_dim to the number of classes
        self.output_net = nn.Sequential(
            nn.Linear(self.hparams.model_dim, self.hparams.model_dim),
            nn.LayerNorm(self.hparams.model_dim),
            nn.ReLU(inplace=True),
            nn.Dropout(self.hparams.dropout),
            nn.Linear(self.hparams.model_dim, self.hparams.num_classes)
        )

    def forward(self, x, mask=None, add_positional_encoding=True):
        """
        Inputs:
```

```
            x: Tensor of shape [batch_size, seq_length, input_dim]
            mask: Optional mask tensor (e.g., to mask padding tokens)
            add_positional_encoding: If True, add the sinusoidal positional encodings
        Returns:
            Output logits of shape [batch_size, seq_length, num_classes]
        """
        # Map raw input to model_dim (if using one-hot vectors, make sure x is float)
        x = self.input_net(x)
        if add_positional_encoding:
            x = self.positional_encoding(x)
        # Pass through the stacked Transformer encoder blocks
        x = self.transformer(x, mask=mask)
        # Compute per-token predictions
        x = self.output_net(x)
        return x

    @torch.no_grad()
    def get_attention_maps(self, x, mask=None, add_positional_encoding=True):
        """
        Extracts attention maps from each encoder block.
        Useful for visualizing what the model attends to.
        """
        x = self.input_net(x)
        if add_positional_encoding:
            x = self.positional_encoding(x)
        attention_maps = self.transformer.get_attention_maps(x, mask=mask)
        return attention_maps

    def configure_optimizers(self):
        """
        Configures the optimizer and learning rate scheduler.

        Returns:
            A tuple containing the optimizer list and a list of scheduler dictionaries,
            with the scheduler set to update at every optimizer step.
        """
        optimizer = optim.Adam(self.parameters(), lr=self.hparams.lr)
        # The cosine warmup scheduler updates the learning rate at each optimizer step.
        lr_scheduler = CosineWarmupScheduler(
            optimizer,
            warmup=self.hparams.warmup,
            max_iters=self.hparams.max_iters
        )
        return [optimizer], [{'scheduler': lr_scheduler, 'interval': 'step'}]

    def training_step(self, batch, batch_idx):
        """
        Performs a single training step.

        Args:
            batch: A tuple (x, y) where x has shape [B, T, input_dim] and y has shape [B, T].
            batch_idx: Index of the batch.

        Returns:
            The computed training loss.
        """
        # Expect batch to be a tuple: (input, target)
        x, y = batch  # x: [B, T, input_dim], y: [B, T] (target class indices)
        logits = self.forward(x, add_positional_encoding=True)  # [B, T, num_classes]
        # Flatten logits and targets for cross-entropy loss
```

```
        loss = F.cross_entropy(logits.view(-1, self.hparams.num_classes), y.view(-1))
        preds = logits.argmax(dim=-1)
        acc = (preds == y).float().mean()
        self.log("train_loss", loss, prog_bar=True, on_step=True, on_epoch=True)
        self.log("train_acc", acc, prog_bar=True, on_step=True, on_epoch=True)
        return loss

    def validation_step(self, batch, batch_idx):
        """
        Performs a single validation step.

        Args:
            batch: A tuple (x, y) where x has shape [B, T, input_dim] and y has shape [B, T].
            batch_idx: Index of the batch.

        Returns:
            A dictionary with validation loss and accuracy.
        """
        x, y = batch
        logits = self.forward(x, add_positional_encoding=True)
        loss = F.cross_entropy(logits.view(-1, self.hparams.num_classes), y.view(-1))
        preds = logits.argmax(dim=-1)
        acc = (preds == y).float().mean()
        self.log("val_loss", loss, prog_bar=True, on_epoch=True, sync_dist=True)
        self.log("val_acc", acc, prog_bar=True, on_epoch=True, sync_dist=True)
        return {"val_loss": loss, "val_acc": acc}

    def test_step(self, batch, batch_idx):
        """
        Performs a single test step.

        Args:
            batch: A tuple (x, y) where x has shape [B, T, input_dim] and y has shape [B, T].
            batch_idx: Index of the batch.

        Returns:
            A dictionary with test loss and accuracy.
        """
        x, y = batch
        logits = self.forward(x, add_positional_encoding=True)
        loss = F.cross_entropy(logits.view(-1, self.hparams.num_classes), y.view(-1))
        preds = logits.argmax(dim=-1)
        acc = (preds == y).float().mean()
        self.log("test_loss", loss, prog_bar=True, on_epoch=True)
        self.log("test_acc", acc, prog_bar=True, on_epoch=True)
        return {"test_loss": loss, "test_acc": acc}
```

22.5. Experiments

With the **Transformer architecture** fully implemented, we can now explore its application to various tasks. In this notebook, we will focus on a **parallel Sequence-to-Sequence task**, demonstrating how Transformers handle structured input-output relationships.

22.5.1. Sequence-to-Sequence Learning

A **Sequence-to-Sequence (Seq2Seq)** task involves both the **input and output** being sequences, which may have different lengths. Common Seq2Seq applications include: - **Machine Translation** (e.g., English -> French) - **Summarization** (e.g., compressing long articles into key points)

Typically, a **Transformer-based Seq2Seq model** consists of: - An **encoder** that processes the input sequence. - A **decoder** that generates the output in an **autoregressive manner**.

For this experiment, we simplify the task by using **only the encoder** and focusing on a **basic sequence-reversal problem**. Given a sequence of N numbers ranging from 0 to M, the goal is to **reverse** the sequence. In NumPy notation, if the input is x, the expected output is $x[::-1]$.

Although this task seems trivial, **RNNs often struggle** with it due to the requirement for **long-term dependencies**. Transformers, designed to handle such dependencies effectively, are expected to **perform well**.

We begin by defining a dataset class for this task.

```
class ReverseDataset(data.Dataset):
    """
    A dataset for a sequence-to-sequence reverse task.

    This dataset generates random sequences of integers (categories) of a fixed length.
    The target for each sequence is simply the input sequence reversed.
    """

    def __init__(self, num_categories, seq_len, size):
        """
        Initializes the ReverseDataset.

        Args:
            num_categories (int): Number of unique categories (values from 0 to num_categories-1).
            seq_len (int): The length of each sequence.
            size (int): The number of sequences in the dataset.
        """
        super().__init__()
        self.num_categories = num_categories
        self.seq_len = seq_len
        self.size = size

        # Generate a tensor of random integers with shape [size, seq_len]
        self.data = torch.randint(self.num_categories, size=(self.size, self.seq_len))

    def __len__(self):
        """
        Returns:
            int: The total number of sequences in the dataset.
        """
        return self.size

    def __getitem__(self, idx):
        """
        Retrieves the input sequence and its reversed version as the label.

        Args:
            idx (int): The index of the sample to retrieve.
```

```
        Returns:
            tuple: A tuple (inp_data, labels) where:
                - inp_data (Tensor): The input sequence of shape [seq_len].
                - labels (Tensor): The target sequence (reversed input) of shape [seq_len].
        """
        inp_data = self.data[idx]
        labels = torch.flip(inp_data, dims=(0,))
        return inp_data, labels
```

We generate a set of random sequences, where each element is a number between **0** and **`num_categories - 1`**. The **labels** for these sequences are simply the **reversed versions** of the original sequences. This ensures that the model learns to map an input sequence to its flipped counterpart.

Next, we define the **data loaders**, which will efficiently batch and shuffle the sequences for training.

```
dataset = partial(ReverseDataset, 10, 16)
train_loader = data.DataLoader(dataset(50000), batch_size=128, shuffle=True, drop_last=True, pin_memory=True)
val_loader   = data.DataLoader(dataset(1000), batch_size=128)
test_loader  = data.DataLoader(dataset(10000), batch_size=128)
```

```
inp_data, labels = train_loader.dataset[0]
print("Input data:", inp_data)
print("Labels:    ", labels)
```

```
Input data: tensor([8, 7, 9, 2, 7, 9, 1, 8, 4, 6, 0, 3, 6, 0, 2, 8])
Labels:     tensor([8, 2, 0, 6, 3, 0, 6, 4, 8, 1, 9, 7, 2, 9, 7, 8])
```

During training, we feed the input sequence into the **Transformer encoder**, predicting an output for each input token. We use the **Cross-Entropy loss** to optimize the predictions. Each number in the sequence is represented as a **one-hot vector**, ensuring that all categories are treated equidistantly in feature space.

Using scalar values (e.g., `0, 1, 2, ...`) instead of one-hot vectors would limit the model's expressiveness since numerical proximity does not imply semantic similarity (e.g., `0` and `1` are no closer than `0` and `9`). An alternative approach is to use **learned embeddings**, as provided by PyTorch's `nn.Embedding`. However, in our case, we achieve the same effect by applying a **linear transformation (`self.input_net`)** to the one-hot vectors, mapping them to dense feature representations.

To implement the **training loop**, we define a new class that inherits from `TransformerPredictor`, overriding the **training, validation, and test step functions** to manage data flow and loss computation.

```
class ReversePredictor(TransformerPredictor):
    """
    A Transformer-based predictor for the reverse sequence task.

    This model inherits from TransformerPredictor and is tailored for tasks
    where the target sequence is the reverse of the input sequence.
    """

    def _calculate_loss(self, batch, mode="train"):
        """
        Computes the cross-entropy loss and accuracy for a given batch.

        Args:
            batch (tuple): A tuple (inp_data, labels) where:
                - inp_data (Tensor): Input tensor of shape [B, T] containing category indices.
                - labels (Tensor): Target tensor of shape [B, T] containing the reversed indices.
            mode (str, optional): Mode string ("train", "val", or "test") used for logging.
```

```
        Returns:
            tuple: A tuple (loss, acc) where:
                - loss (Tensor): Cross-entropy loss computed over the batch.
                - acc (Tensor): Mean accuracy over the batch.
        """
        inp_data, labels = batch
        inp_data = F.one_hot(inp_data, num_classes=self.hparams.num_classes).float()
        preds = self.forward(inp_data, add_positional_encoding=True)  # [B, T, num_classes]
        loss = F.cross_entropy(preds.view(-1, self.hparams.num_classes), labels.view(-1))
        preds_idx = preds.argmax(dim=-1)
        acc = (preds_idx == labels).float().mean()

        self.log(f"{mode}_loss", loss)
        self.log(f"{mode}_acc", acc)
        return loss, acc

    def training_step(self, batch, batch_idx):
        """
        Performs a single training step.

        Args:
            batch (tuple): A tuple (inp_data, labels) for the batch.
            batch_idx (int): Index of the current batch.

        Returns:
            Tensor: The training loss for the current batch.
        """
        loss, _ = self._calculate_loss(batch, mode="train")
        return loss

    def validation_step(self, batch, batch_idx):
        """
        Performs a single validation step.

        Args:
            batch (tuple): A tuple (inp_data, labels) for the batch.
            batch_idx (int): Index of the current batch.

        Returns:
            dict: A dictionary containing validation loss and accuracy.
        """
        loss, acc = self._calculate_loss(batch, mode="val")
        return {"val_loss": loss, "val_acc": acc}

    def test_step(self, batch, batch_idx):
        """
        Performs a single test step.

        Args:
            batch (tuple): A tuple (inp_data, labels) for the batch.
            batch_idx (int): Index of the current batch.

        Returns:
            dict: A dictionary containing test loss and accuracy.
        """
        loss, acc = self._calculate_loss(batch, mode="test")
        return {"test_loss": loss, "test_acc": acc}
```

Finally, we define a **training function**. We create a `pl.Trainer` object that runs for N **epochs**, logs results in **TensorBoard**, and saves the **best model** based on validation performance.

A key parameter passed to the trainer is **`gradient_clip_val`**, which **clips the norm of gradients** for all parameters before an optimizer step. This helps prevent the model from diverging due to extremely high gradients, which can occur near sharp loss surfaces (for more details, see DeepAI's glossary). Gradient clipping is particularly useful for **Transformer models**, stabilizing training in the **first few iterations** and beyond.

In **plain PyTorch**, gradient clipping can be applied using:

```
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=clip_value)
```

(See the PyTorch documentation for more details.)

The **clip value** is typically set between **0.5 and 10**, depending on the severity of clipping required for large gradients.

With this understanding, we now implement the training function.

22.6. Training our Model

```python
def train_reverse(**kwargs):
    """
    Trains or loads a ReversePredictor model for the reverse sequence task.

    Sets up a PyTorch Lightning trainer with a ModelCheckpoint callback (monitoring validation accuracy).
    If a pretrained model exists, it is loaded; otherwise, a new model is trained using the global
    train and validation loaders. The model is then tested on both validation and test sets.

    Args:
        **kwargs: Keyword arguments for initializing ReversePredictor, such as:
                  - input_dim (int): Number of input categories.
                  - model_dim (int): Transformer hidden dimension.
                  - num_heads (int): Number of attention heads.
                  - num_classes (int): Number of output classes.
                  - num_layers (int): Number of encoder layers.
                  - dropout (float): Dropout rate.
                  - lr (float): Learning rate.
                  - warmup (int): Warmup steps.

    Returns:
        tuple: (model, result) where result is a dict with "test_acc" and "val_acc".
    """
    root_dir = os.path.join(MODEL_PATH, "ReverseTask")
    os.makedirs(root_dir, exist_ok=True)
    trainer = pl.Trainer(
        default_root_dir=root_dir,
        callbacks=[ModelCheckpoint(save_weights_only=True, mode="max", monitor="val_acc")],
        accelerator="gpu" if str(device).startswith("cuda") else "cpu",
        devices=1,
        max_epochs=10,
        gradient_clip_val=5
    )
    trainer.logger._default_hp_metric = None  # Disable default hyperparameter metric logging

    pretrained_filename = os.path.join(MODEL_PATH, "ReverseTask.ckpt")
```

```
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model, loading...")
        model = ReversePredictor.load_from_checkpoint(pretrained_filename)
    else:
        model = ReversePredictor(max_iters=trainer.max_epochs * len(train_loader), **kwargs)
        trainer.fit(model, train_loader, val_loader)

    val_result = trainer.test(model, val_loader, verbose=False)
    test_result = trainer.test(model, test_loader, verbose=False)
    result = {
        "test_acc": test_result[0]["test_acc"],
        "val_acc": val_result[0]["test_acc"]
    }

    model = model.to(device)
    return model, result
```

Finally, we proceed to **train the model**. For this setup, we use a **single encoder block** and **one attention head** in the **Multi-Head Attention** mechanism. This configuration is chosen due to the **simplicity of the task**, making the attention mechanism more interpretable.

In this specific case, the attention weights can **serve as an explanation** for the model's predictions. This contrasts with deep Transformer models, where **interpretability becomes more complex**, as discussed in prior research on attention mechanisms.

```
reverse_model, reverse_result = train_reverse(input_dim=train_loader.dataset.num_categories,
                                              model_dim=32,
                                              num_heads=1,
                                              num_classes=train_loader.dataset.num_categories,
                                              num_layers=1,
                                              dropout=0.0,
                                              lr=5e-4,
                                              warmup=50)
```

```
INFO:pytorch_lightning.utilities.rank_zero:GPU available: True (cuda), used:
True
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU
cores
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPUs
```

```
Found pretrained model, loading...
```

```
INFO:pytorch_lightning.utilities.migration.utils:Lightning automatically
upgraded your loaded checkpoint from v1.0.2 to v2.5.0.post0. To apply the
upgrade to your files permanently ... [94 chars truncated]
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES:
[0]
```

```
Testing: |          | 0/? [00:00<?, ?it/s]
```

```
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES:
[0]
```

```
Testing: |          | 0/? [00:00<?, ?it/s]
```

The **PyTorch Lightning warning** regarding the **number of workers** can be ignored for now. Given the **simplicity of our dataset**, the `__getitem__` method completes in **negligible time**, meaning that **subprocesses for data loading are unnecessary**.

In fact, **increasing the number of workers** could **slow down training** due to **communication overhead** between processes and threads. With this in mind, let's proceed to print the results:

```
print(f"Val accuracy:  {(100.0 * reverse_result['val_acc']):4.2f}%")
print(f"Test accuracy: {(100.0 * reverse_result['test_acc']):4.2f}%")
```

```
Val accuracy:  100.00%
Test accuracy: 100.00%
```

As expected, the **Transformer successfully solves the task**. However, an interesting question arises: **how does the attention mechanism behave** in the **Multi-Head Attention block** for a given input?

To better understand this, let's visualize the attention patterns below.

22.7. The Attention Mask

```
data_input, labels = next(iter(val_loader))
inp_data = F.one_hot(data_input, num_classes=reverse_model.hparams.num_classes).float()
inp_data = inp_data.to(device)
attention_maps = reverse_model.get_attention_maps(inp_data)
```

The object `attention_maps` is a **list of length** N, where N **represents the number of layers** in the Transformer model. Each element in this list is a **tensor** with the shape **[Batch, Heads, SeqLen, SeqLen]**. This structure allows us to examine how attention is distributed across different layers, heads, and sequence positions.

We can verify this by inspecting the tensor shapes below.

```
attention_maps[0].shape   # display value
```

```
torch.Size([128, 1, 16, 16])
```

Next, we will write a **plotting function** that takes as input the **sequences, attention maps, and a batch index** specifying which element to visualize. The function will generate a plot where:

- **Rows** represent different **layers** of the Transformer.
- **Columns** represent different **attention heads**.

Since **softmax** has been applied along each row separately, the values in each row indicate the distribution of attention over the input sequence for a given token. This visualization helps in understanding how different parts of the input sequence attend to each other across layers and attention heads.

```
def plot_attention_maps(input_data, attn_maps, idx=0):
    """
    Plots attention maps for a given sample from a batch.

    Args:
        input_data (Tensor or None): Tensor of shape [B, T, ...] representing the input sequence.
            If provided, its values (converted to numpy) are used as tick labels.
            If None, integer positions are used.
        attn_maps (list of Tensors): List of attention map tensors from each layer.
```

```
            Each tensor should have shape [B, num_heads, T, T].
        idx (int): Index of the sample in the batch to visualize.

    Returns:
        None. Displays a matplotlib figure with attention maps.
    """
    # If input_data is provided, extract the sample and convert to numpy.
    # Otherwise, create a default sequence [0, 1, ..., T-1].
    if input_data is not None:
        input_data = input_data[idx].detach().cpu().numpy()
    else:
        seq_len = attn_maps[0][idx].shape[-1]
        input_data = np.arange(seq_len)
    attn_maps = [m[idx].detach().cpu().numpy() for m in attn_maps]

    num_layers = len(attn_maps)
    num_heads = attn_maps[0].shape[0]
    seq_len = input_data.shape[0]
    fig_size = 4 if num_heads == 1 else 3

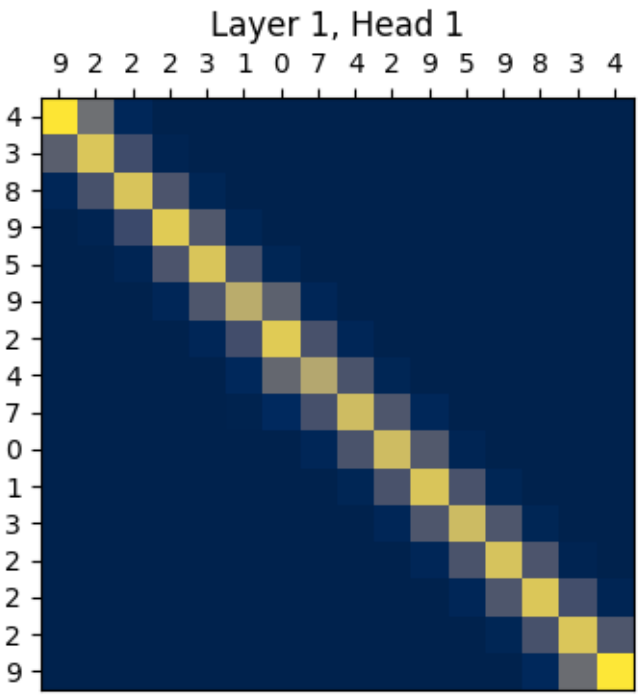
    # Create subplots: rows = layers, columns = heads.
    fig, ax = plt.subplots(num_layers, num_heads, figsize=(num_heads * fig_size, num_layers * fig_size))
    # Ensure ax is a 2D array.
    if num_layers == 1:
        ax = [ax]
    if num_heads == 1:
        ax = [[a] for a in ax]

    # Plot each attention map.
    for row in range(num_layers):
        for col in range(num_heads):
            ax[row][col].imshow(attn_maps[row][col], origin='lower', vmin=0)
            ax[row][col].set_xticks(range(seq_len))
            ax[row][col].set_xticklabels(input_data.tolist())
            ax[row][col].set_yticks(range(seq_len))
            ax[row][col].set_yticklabels(input_data.tolist())
            ax[row][col].xaxis.set_ticks_position('top')
            ax[row][col].xaxis.set_label_position('top')
            ax[row][col].set_title(f"Layer {row + 1}, Head {col + 1}")

    fig.subplots_adjust(hspace=0.5)
    plt.show()
```

Finally, we can plot the attention map of our **trained Transformer** on the reverse task:

```
plot_attention_maps(data_input, attention_maps, idx=0)
```



The model has successfully learned to **attend to the token at the flipped index** of itself, effectively solving the task as intended. However, we observe that it also distributes some attention to **neighboring indices** near the flipped position. This behavior arises because the model does not require a **perfect, hard attention mechanism** to solve the problem, it can rely on an **approximate, noisier attention map**. The slight attention to nearby positions is influenced by **positional encoding similarities**, which is an expected effect of using **continuous position representations** in the Transformer architecture.

22.8. Homework (Optional)

1. **Without the $\sqrt{d_k}$.** Train the reverse-sequence model with and without the scaling factor. Plot validation accuracy vs. epoch for both and explain the difference in terms of softmax saturation.
2. **Number of heads.** Train the same model with 1, 4 and 8 attention heads, keeping the total model dimension fixed. Report validation accuracy and training time for each configuration.
3. **Attention map inspection.** Visualise the attention weights for the last encoder layer on three test sequences. For each, describe what the attention pattern tells you about how the model solves the reverse task.
4. **Causal vs. bidirectional.** Add causal masking to the encoder (turning it into a decoder-style block) and retrain. Report the accuracy drop on the reverse task and explain why.
5. **Relative position encoding.** Replace the sinusoidal absolute position encoding with a learned relative position embedding (one scalar per pairwise distance). Retrain and report the difference.

6. **Longer sequences.** Increase the sequence length from 16 to 64 and retrain. Report accuracy and memory usage. Discuss why attention's $O(n^2)$ memory scaling becomes a concern at longer lengths.

Chapter 23

Transformer from Scratch

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1o0W6u6AQ01VkLh3qXsAx58o7JWkf32D8?usp=sharing

In this notebook, we will implement a **Transformer model from scratch**, based on the influential paper "Attention is All You Need [29]" by Vaswani et al. This model has become a foundational architecture for many state-of-the-art models in natural language processing (NLP), and understanding its mechanics is essential for anyone diving into deep learning and sequence modeling.

23.0.1. Notebook Overview

We will implement the **Transformer** model step-by-step, focusing on the following components:

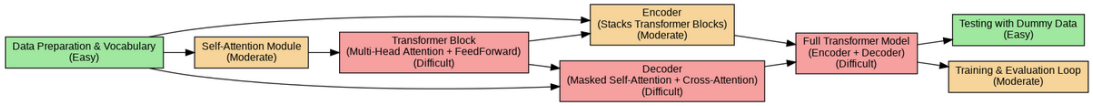
1. **Self-Attention Mechanism**: The core mechanism enabling the Transformer to capture relationships between all words in a sequence simultaneously.
2. **Multi-Head Attention**: An extension of self-attention that allows the model to jointly attend to information from different representation subspaces.
3. **Positional Encoding**: Transformers lack inherent order-awareness, so positional encodings are added to input embeddings to preserve word order.
4. **Transformer Blocks**: Each block contains layers of multi-head self-attention, normalization, and feed-forward networks.
5. **Encoder and Decoder Structure**: The Transformer architecture is based on stacks of encoder and decoder layers that interact through attention mechanisms.
6. **Full Transformer Model**: We will combine the encoder and decoder to create the complete Transformer model, capable of handling machine translation or other sequence-to-sequence tasks.

Let's dive into the implementation!

23.1. Necessary Imports

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
from IPython.display import Image as IMG

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```



23.2. Masking in self-attention

A transformer uses three distinct masks, each serving a different purpose. Getting them right is the difference between a model that learns and a model that silently cheats.

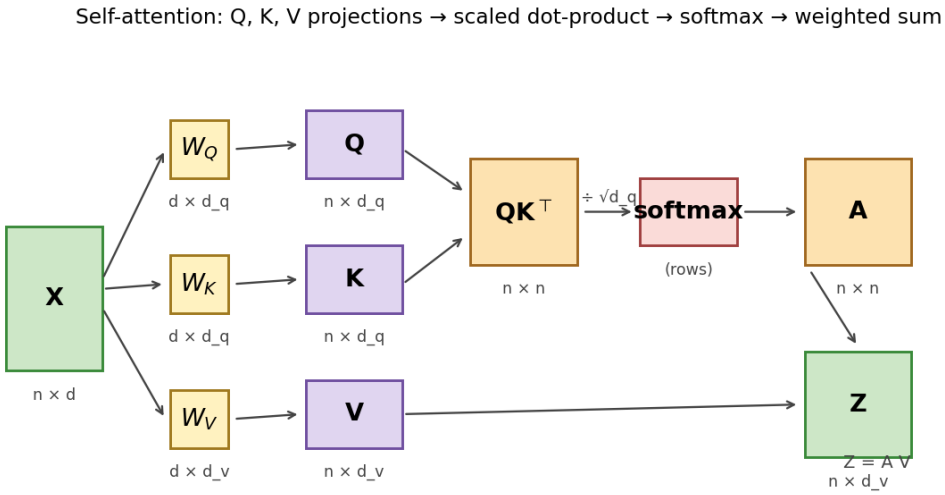
Padding mask hides the padding tokens that were added to fit every sequence into a fixed-length batch tensor. Without it, attention weights leak into the padded positions and the softmax normalisation puts spurious mass on tokens that carry no information. The padding mask is applied to **keys** in every attention block (encoder self-attention, decoder self-attention, decoder cross-attention).

Causal mask is the triangular mask used in decoder self-attention. It blocks each position from attending to any later position, enforcing the autoregressive property that token `t` is predicted only from tokens `1..t-1`. Implementation is an upper-triangular mask of $-\infty$ added to the pre-softmax scores. Without the causal mask, the decoder trivially copies the next token from its input and training never teaches it to predict.

Cross-attention key mask applies the *source* sequence's padding mask to cross-attention keys in the decoder. The decoder attends over the encoder output. Positions corresponding to source padding must be masked out so the decoder does not align target tokens with non-tokens.

The practical rule: build one mask tensor per sequence (source and target) and combine them at the attention level via element-wise or between the causal mask and the padding masks where applicable.

23.3. The Self-Attention Module



The **self-attention** mechanism is a key component of transformer models that allows each token in an input sequence to attend to all other tokens, **dynamically computing contextualized representations**. It achieves this by computing `query`, `key`, and `value` matrices, applying scaled dot-product attention, and

weighting the contribution of each token based on its relevance, enabling the model to capture long-range dependencies and contextual meaning effectively.

```
class SelfAttention(nn.Module):
    def __init__(self, embed_size, heads):
        super(SelfAttention, self).__init__()
        # Dimensionality of input embeddings
        self.embed_size = embed_size
        # Number of attention heads
        self.heads = heads
        # Dimensionality of each head
        self.head_dim = embed_size // heads

        # Ensure the embedding size is divisible by the number of heads
        assert (
            self.head_dim * heads == embed_size
        ), "Embedding size needs to be divisible by heads"

        # Linear layers for transforming inputs to query, key, and value vectors
        self.values = nn.Linear(embed_size, embed_size)
        self.keys = nn.Linear(embed_size, embed_size)
        self.queries = nn.Linear(embed_size, embed_size)

        # Linear layer for output transformation after attention calculation
        self.fc_out = nn.Linear(embed_size, embed_size)

    def forward(self, values, keys, query, mask):
        N = query.shape[0]  # Number of samples in the batch

        # Get the sequence length for each input (value, key, query)
        value_len, key_len, query_len = values.shape[1], keys.shape[1], query.shape[1]

        # Transform values, keys, and queries using the linear layers
        # Resulting shapes: (N, sequence_length, embed_size)
        values = self.values(values)
        keys = self.keys(keys)
        queries = self.queries(query)

        # Reshape values, keys, and queries for multi-head attention
        # New shape: (N, sequence_length, heads, head_dim)
        values = values.reshape(N, value_len, self.heads, self.head_dim)
        keys = keys.reshape(N, key_len, self.heads, self.head_dim)
        queries = queries.reshape(N, query_len, self.heads, self.head_dim)

        # Calculate the energy scores using dot-product attention
        # einsum notation helps with the multi-dimensional matrix multiplication:
        # "nqhd,nkhd->nhqk" means:
        # - N: batch size
        # - q, k: query and key positions
        # - h: number of heads
        # - d: head dimension
        # This results in a tensor of shape (N, heads, query_len, key_len)
        energy = torch.einsum("nqhd,nkhd->nhqk", [queries, keys])

        # If a mask is provided, apply it to the energy scores by setting masked positions to a large negative
  value
        # This prevents the model from attending to certain positions
        if mask is not None:
            energy = energy.masked_fill(mask == 0, float("-1e20"))
```

```
        # Normalize energy values across the last dimension (key dimension) to obtain attention weights
        # The division by sqrt(embed_size) helps stabilize gradients
        attention = torch.softmax(energy / (self.embed_size ** (1 / 2)), dim=3)

        # Weighted sum of the values, using the attention scores as weights
        # einsum notation: "nhql,nlhd->nqhd" to match dimensions for summation:
        # Resulting shape is (N, query_len, heads, head_dim)
        out = torch.einsum("nhql,nlhd->nqhd", [attention, values]).reshape(
            N, query_len, self.heads * self.head_dim
        )

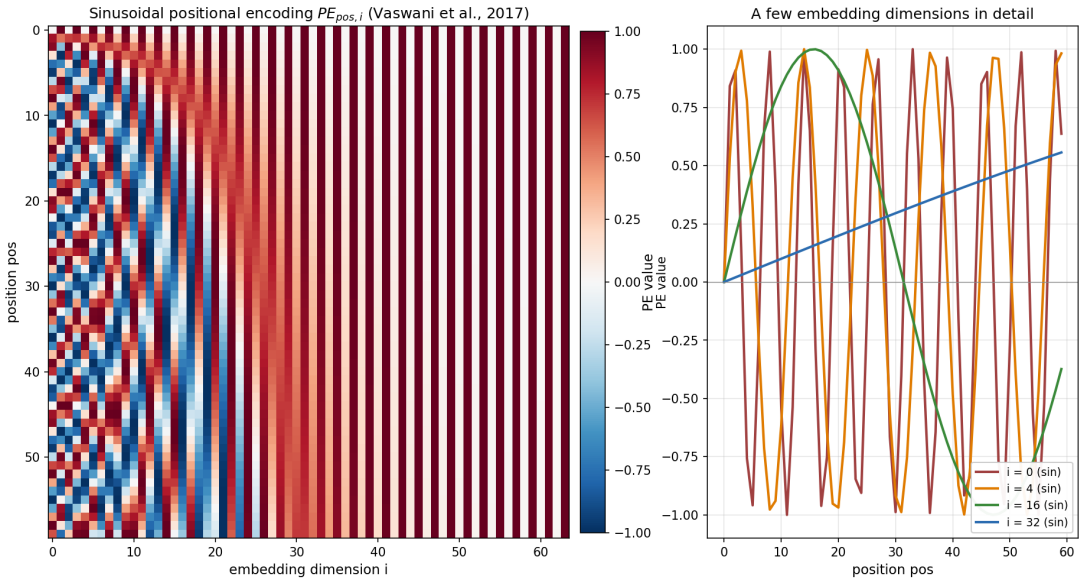
        # Final linear transformation to produce the output of the same shape as input
        # (N, query_len, embed_size), combining all heads' outputs
        out = self.fc_out(out)

        return out
```

23.4. Sinusoidal positional encoding

Self-attention is permutation-equivariant by itself, so we need to inject position information explicitly. The original Transformer adds a fixed **sinusoidal positional encoding** to the input embeddings:

$$PE_{pos,2i} = \sin\left(pos/10000^{2i/d}\right), \quad PE_{pos,2i+1} = \cos\left(pos/10000^{2i/d}\right)$$



Low dimensions oscillate quickly with position, high dimensions slowly. Each position therefore has a unique signature, and relative positions are easy for the model to recover via linear combinations.

23.5. The Encoder

A **Transformer block** consists of two key components: `multi-head self-attention` and a `position-wise feedforward network`, both followed by layer normalization and residual connections. The multi-head self-attention captures contextual relationships between tokens, while the feedforward network applies non-linear transformations independently to each token.

The **Transformer block** is represented by the gray section, while the entire **encoder** also includes tokenization, embedding, and positional encoding components, which prepare the input before it passes through the self-attention and feedforward layers.

```
class TransformerBlock(nn.Module):
    def __init__(self, embed_size, heads, dropout, forward_expansion):
        super(TransformerBlock, self).__init__()

        # Define the self-attention layer using the previously defined SelfAttention class
        self.attention = SelfAttention(embed_size, heads)

        # Define layer normalization layers for stabilization and improved gradient flow
        self.norm1 = nn.LayerNorm(embed_size)  # Normalization after attention
        self.norm2 = nn.LayerNorm(embed_size)  # Normalization after feed-forward network

        # Define the feed-forward network
        # Sequential model with two linear layers separated by ReLU activation:
        # - Expands the dimensionality to `forward_expansion * embed_size` in the hidden layer
        # - Reduces it back to `embed_size` in the output layer
        self.feed_forward = nn.Sequential(
            nn.Linear(embed_size, forward_expansion * embed_size),  # First linear layer
            nn.ReLU(),  # Activation function introducing non-linearity
            nn.Linear(forward_expansion * embed_size, embed_size)  # Second linear layer
        )

        # Define dropout layer to help prevent overfitting by randomly setting some activations to zero
        self.dropout = nn.Dropout(dropout)

    def forward(self, value, key, query, mask):

        # Compute self-attention output using provided value, key, and query inputs
        # Shape after self-attention should be (N, sequence_length, embed_size)
        attention = self.attention(value, key, query, mask)

        # Apply first residual connection (attention + query), followed by normalization and dropout
        # - Adds the query as a residual (skip) connection to help retain input information
        # - Normalizes the sum for smoother training
        # - Applies dropout to prevent overfitting
        x = self.dropout(self.norm1(attention + query))

        # Pass through the feed-forward network
        # - The input `x` is fed into the feed-forward layers defined above
        forward = self.feed_forward(x)

        # Apply second residual connection (feed-forward output + x), normalization, and dropout
        # - Adds the original `x` as a skip connection to retain information
        # - Normalizes the output to keep activations stable
        # - Applies dropout again to prevent overfitting
        out = self.dropout(self.norm2(forward + x))

        return out  # Returns the final output with the same shape as the input
```

```
class Encoder(nn.Module):
    def __init__(self, src_vocab_size, embed_size, num_layers, heads, device, forward_expansion, dropout,
  max_length):
        super(Encoder, self).__init__()

        # Save embedding size and device for later use
        self.embed_size = embed_size
```

```
        self.device = device

        # Define word embedding layer: maps vocabulary tokens to embedding vectors
        self.word_embedding = nn.Embedding(src_vocab_size, embed_size)

        # Define positional embedding layer: adds positional context to embeddings
        self.position_embedding = nn.Embedding(max_length, embed_size)

        # Define multiple Transformer blocks (num_layers indicates how many)
        # nn.ModuleList allows stacking Transformer blocks in a list format, making them iterable in the
    forward pass
        self.layers = nn.ModuleList(
            [
                TransformerBlock(embed_size, heads, dropout, forward_expansion)
                for _ in range(num_layers)
            ]
        )

        # Dropout layer to apply dropout to the embedding sum
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask):
        # Get batch size (N) and sequence length
        N, seq_length = x.shape

        # Generate position indices for the sequence
        # `torch.arange(0, seq_length)` creates a sequence from 0 to seq_length-1
        # `expand(N, seq_length)` repeats this for each sample in the batch
        # `.to(self.device)` moves it to the appropriate device (CPU/GPU)
        positions = torch.arange(0, seq_length).expand(N, seq_length).to(self.device)

        # Apply word and positional embeddings, then sum them
        # Shape after word embedding: (N, seq_length, embed_size)
        # Shape after position embedding: (N, seq_length, embed_size)
        # Summing both gives a combined embedding that contains token identity and position information
        out = self.dropout(
            (self.word_embedding(x) + self.position_embedding(positions))
        )

        # Pass the embedding through each Transformer block in the stack
        # Each layer refines the representation of `out` based on self-attention and feed-forward networks
        for layer in self.layers:
            out = layer(out, out, out, mask)

        # Return the final output after all Transformer blocks
        # Shape remains (N, seq_length, embed_size), with enhanced context and representation
        return out
```

23.6. The Decoder

The **decoder block** in a Transformer consists of masked multi-head self-attention, encoder-decoder attention, and a position-wise feedforward network, enabling it to generate output tokens while attending to both previous outputs and the encoded input. The full decoder stacks multiple such blocks, incorporating causal masking to prevent future information leakage, ensuring that predictions are generated sequentially in an **autoregressive manner**.

```
class DecoderBlock(nn.Module):
    def __init__(self, embed_size, heads, forward_expansion, dropout, device):
        super(DecoderBlock, self).__init__()

        # Layer normalization for the output of self-attention layer
        self.norm = nn.LayerNorm(embed_size)

        # Self-attention layer for the target sequence (decoder input) with multiple heads
        self.attention = SelfAttention(embed_size, heads=heads)

        # Transformer block that includes cross-attention (attending to encoder output) and a feed-forward
  layer
        self.transformer_block = TransformerBlock(embed_size, heads, dropout, forward_expansion)

        # Dropout layer for regularization
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, value, key, src_mask, trg_mask):

        # Perform self-attention on the decoder input `x` (target sequence) using the target mask `trg_mask`
        # Allows each position in the target to attend to previous positions
        attention = self.attention(x, x, x, trg_mask)

        # Add a residual connection (x + attention), followed by normalization and dropout
        # `query` now contains self-attended target sequence information
        query = self.dropout(self.norm(attention + x))

        # Pass the `query` (self-attended target sequence), encoder outputs `value` and `key`, and source mask
  `src_mask`
        # through the Transformer block for cross-attention and feed-forward processing
        # Cross-attention layer allows each position in the decoder to attend to relevant parts of the encoder
  output
        out = self.transformer_block(value, key, query, src_mask)

        return out  # Returns the final processed output of the decoder block
```

```
class Decoder(nn.Module):
    def __init__(self, trg_vocab_size, embed_size, num_layers, heads, forward_expansion, dropout, device,
  max_length):
        super(Decoder, self).__init__()

        # Save device information for later use
        self.device = device

        # Embedding layer for target vocabulary tokens
        self.word_embedding = nn.Embedding(trg_vocab_size, embed_size)

        # Positional embedding layer to add positional information to each token in the target sequence
        self.position_embedding = nn.Embedding(max_length, embed_size)

        # Stack multiple Decoder blocks, each containing self-attention, cross-attention, and feed-forward
  layers
        self.layers = nn.ModuleList(
            [
                DecoderBlock(embed_size, heads, forward_expansion, dropout, device)
                for _ in range(num_layers)
            ]
        )
```

```
        # Final output layer that projects the decoder output embeddings to the target vocabulary size
        # This layer is used to produce the final output predictions
        self.fc_out = nn.Linear(embed_size, trg_vocab_size)

        # Dropout layer for regularization
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, enc_out, src_mask, trg_mask):
        # Get the batch size and sequence length from the input `x`
        N, seq_length = x.shape

        # Create position indices for each token position in the sequence
        # `torch.arange(0, seq_length)` creates indices from 0 to seq_length - 1
        # `expand(N, seq_length)` repeats these for each sample in the batch
        # `.to(self.device)` moves the indices to the specified device (CPU/GPU)
        positions = torch.arange(0, seq_length).expand(N, seq_length).to(self.device)

        # Add word embeddings and positional embeddings together
        # This combined embedding includes both the identity of each token and its position
        x = self.dropout((self.word_embedding(x) + self.position_embedding(positions)))

        # Pass the combined embedding through each Decoder block
        # `enc_out`, `src_mask`, and `trg_mask` are passed to each block for cross-attention
        for layer in self.layers:
            x = layer(x, enc_out, enc_out, src_mask, trg_mask)

        # Apply the final linear layer to map the decoder output to target vocabulary logits
        # This output will be used for predictions (e.g., through a softmax layer)
        out = self.fc_out(x)

        return out  # Returns logits for each token in the target vocabulary
```

23.7. Putting it All Together

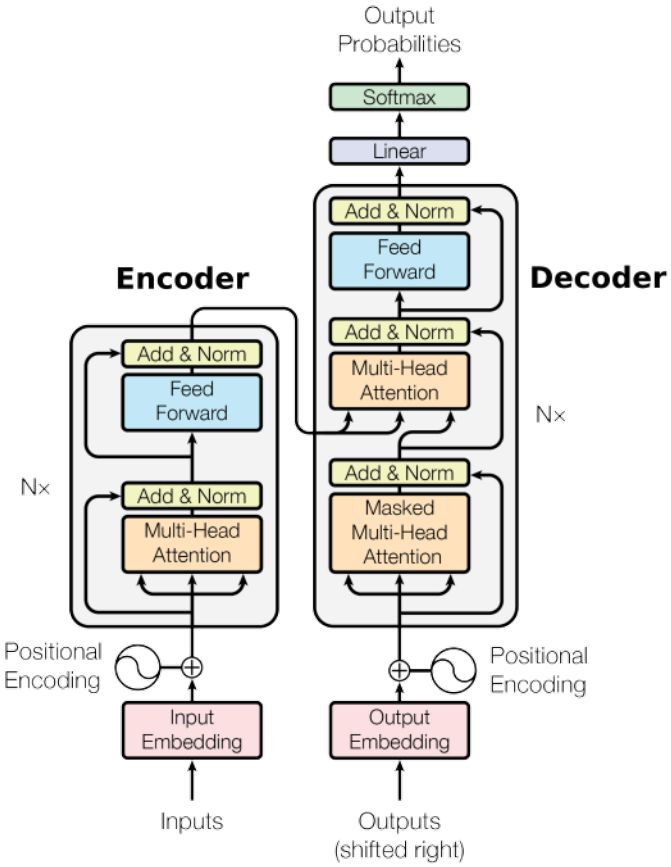


Figure credit: Vaswani et al., 2017.

```python
class Transformer(nn.Module):
    def __init__(self, src_vocab_size, trg_vocab_size, src_pad_idx, trg_pad_idx, embed_size=512, num_layers=6,
  forward_expansion=4, heads=8, dropout=0, device="cpu", max_length=100):
        super(Transformer, self).__init__()

        # Initialize the Encoder and Decoder with specified parameters
        self.encoder = Encoder(
            src_vocab_size, embed_size, num_layers, heads, device, forward_expansion, dropout, max_length
        )
        self.decoder = Decoder(
            trg_vocab_size, embed_size, num_layers, heads, forward_expansion, dropout, device, max_length
        )

        # Padding indices to identify padding tokens in the source and target sequences
        self.src_pad_idx = src_pad_idx  # Index for padding tokens in the source vocabulary
        self.trg_pad_idx = trg_pad_idx  # Index for padding tokens in the target vocabulary
        self.device = device  # Device (CPU or GPU) for model operations

    def make_src_mask(self, src):
        # Create a mask for the source sequence to ignore padding tokens
        # The mask has the shape (N, 1, 1, src_len), where:
        # - N is the batch size
        # - `src_len` is the length of the source sequence
        # This mask will have `True` for positions that are not padding tokens and `False` for padding tokens
        src_mask = (src != self.src_pad_idx).unsqueeze(1).unsqueeze(2)
        return src_mask.to(self.device)  # Move mask to the appropriate device
```

```
    def make_trg_mask(self, trg):
        # Create a mask for the target sequence to prevent attending to future tokens
        # Shape of `trg_mask`: (N, 1, trg_len, trg_len), where:
        # - N is the batch size
        # - `trg_len` is the length of the target sequence
        # Uses `torch.tril` to create a lower triangular matrix, ensuring each token only attends to previous
  tokens
        N, trg_len = trg.shape
        trg_mask = torch.tril(torch.ones((trg_len, trg_len))).expand(N, 1, trg_len, trg_len)
        return trg_mask.to(self.device)  # Move mask to the appropriate device

    def forward(self, src, trg):
        # Generate masks for the source and target sequences
        src_mask = self.make_src_mask(src)  # Mask for padding in source sequence
        trg_mask = self.make_trg_mask(trg)  # Mask for preventing future token access in target sequence

        # Pass the source sequence and its mask to the encoder
        enc_src = self.encoder(src, src_mask)

        # Pass the target sequence, encoded source output, and both masks to the decoder
        out = self.decoder(trg, enc_src, src_mask, trg_mask)

        # Return the final output logits from the decoder
        return out
```

23.7.1. Testing on Dummy Data

```
# Create a sample source input tensor and move it to the chosen device
x = torch.tensor([[1, 5, 6, 4, 3, 9, 5, 2, 0], [1, 8, 7, 3, 4, 5, 6, 7, 2]]).to(device)
# Shape of `x` is (2, 9), representing two sequences (batch size of 2) with 9 tokens each

# Create a sample target tensor and move it to the chosen device
trg = torch.tensor([[1, 7, 4, 3, 5, 9, 2, 0], [1, 5, 6, 2, 4, 7, 6, 2]]).to(device)
# Shape of `trg` is (2, 8), representing two sequences (batch size of 2) with 8 tokens each

# Define padding indices for source and target sequences
src_pad_idx = 0  # Padding index for the source vocabulary
trg_pad_idx = 0  # Padding index for the target vocabulary

# Define source and target vocabulary sizes
src_vocab_size = 10  # Number of unique tokens in the source vocabulary
trg_vocab_size = 10  # Number of unique tokens in the target vocabulary

# Initialize the Transformer model and move it to the device
model = Transformer(src_vocab_size, trg_vocab_size, src_pad_idx, trg_pad_idx, device=device).to(device)

# Pass the source tensor `x` and the target tensor `trg` (excluding the last token for teacher forcing)
out = model(x, trg[:, :-1])
# `trg[:, :-1]` means we use all tokens except the last one as input to the decoder during training

# Print the shape of the output
print(out.shape)
# Expected output shape: (2, 7, trg_vocab_size)
# Explanation:
# - Batch size: 2 (from input batch)
# - Sequence length: 7 (target length - 1 due to teacher forcing)
# - trg_vocab_size: 10 (output is logits over target vocabulary)
```

```
torch.Size([2, 7, 10])
```

23.8. Training the Transformer Model

We will generate a **mini-dataset** consisting of dummy **English-French** sentence pairs for training our Transformer model. From these sentence pairs, we will construct a basic vocabulary and define two dictionaries: `word_to_idx` (mapping words to indices) and `idx_to_word` (mapping indices back to words), which will serve as our tokenization scheme. Additionally, we will pad sentences with zeros to ensure uniform input length.

```
sentence_pairs = [
    # Greetings and questions
    (["how", "are", "you"], ["comment", "ca", "va"]),
    (["hello", "world"], ["bonjour", "monde"]),
    (["good", "morning"], ["bonjour", "matin"]),
    (["hello"], ["bonjour"]),

    # Common phrases
    (["you", "are", "good"], ["tu", "es", "bien"]),
    (["I", "am", "fine"], ["je", "vais", "bien"]),
    (["thank", "you"], ["merci"]),
    (["my", "friend"], ["mon", "ami"]),

    # Requests
    (["please", "help", "me"], ["s'il", "te", "plaît", "aide", "moi"]),
    (["help", "me"], ["aide", "moi"]),

    # Expanded questions
    (["are", "you", "my", "friend"], ["es", "tu", "mon", "ami"]),
    (["how", "is", "your", "day"], ["comment", "est", "ta", "journée"]),
    (["I", "am", "good"], ["je", "suis", "bien"]),
    (["good", "evening"], ["bonsoir"]),

    # Additional polite expressions
    (["please"], ["s'il", "te", "plaît"]),
    (["thank", "you", "my", "friend"], ["merci", "mon", "ami"]),
]

word_set = set()
for src, trg in sentence_pairs:
    word_set.update(src)
    word_set.update(trg)

# Define special tokens
special_tokens = ['<pad>', '<sos>', '<eos>']

# Create vocabulary mapping, ensuring special tokens come first.
word_to_idx = {token: idx for idx, token in enumerate(special_tokens)}
for word in word_set:
    if word not in word_to_idx:
        word_to_idx[word] = len(word_to_idx)

# Create reverse dictionary
idx_to_word = {idx: word for word, idx in word_to_idx.items()}

# Padding index (same for source and target)
PAD_IDX = word_to_idx["<pad>"]
```

```
# Define maximum sequence length (you might want to choose a fixed max length based on your data)
# Here, as an example, we set it to 10. Note that for target sentences, this length should include <sos> and
   <eos>.
MAX_LENGTH = 10
```

23.8.1. Utility Functions

```
def encode_src_sentence(sentence, word_to_idx, max_length):
    """
    Encode a sentence into a fixed-length sequence of token indices.

    Args:
        sentence (list of str): The input sentence as a list of words.
        word_to_idx (dict): Dictionary mapping words to their corresponding indices.
        max_length (int): The fixed length for encoding, applying padding or truncation if needed.

    Returns:
        list of int: The encoded sentence as a list of token indices with padding or truncation.
    """
    encoded = [word_to_idx[word] for word in sentence]
    # Pad if shorter than max_length
    if len(encoded) < max_length:
        encoded += [PAD_IDX] * (max_length - len(encoded))
    return encoded[:max_length]  # Truncate if necessary

def encode_trg_sentence(sentence, word_to_idx, max_length):
    """
    Encode a target sentence by adding <sos> at the beginning and <eos> at the end.
    Then pad/truncate to a fixed length.
    Args:
        sentence (list of str): The target sentence as a list of words.
        word_to_idx (dict): Dictionary mapping words to their corresponding indices.
        max_length (int): The fixed length for encoding, applying padding or truncation if needed.

    Returns:
        list of int: The encoded sentence as a list of token indices with padding or truncation.
    """
    encoded = [word_to_idx["<sos>"]] + [word_to_idx[word] for word in sentence] + [word_to_idx["<eos>"]]
    if len(encoded) < max_length:
        encoded += [PAD_IDX] * (max_length - len(encoded))
    return encoded[:max_length]
```

```
def translate_sentence(sentence):
    """
    Translate an input sentence using a trained Transformer model.

    Args:
        sentence (list of str): The input sentence as a list of words.

    Returns:
        list of str: The translated sentence as a list of words.
    """
    model.eval()
    # Encode the source sentence
    tokens = encode_src_sentence(sentence, word_to_idx, MAX_LENGTH)
    src_tensor = torch.tensor(tokens).unsqueeze(0).to(device)  # Shape: (1, MAX_LENGTH)

    # Initialize the target sequence with <sos>
```

```
    trg_indices = [word_to_idx["<sos>"]]

    for _ in range(MAX_LENGTH - 1):
        trg_tensor = torch.tensor(trg_indices).unsqueeze(0).to(device)
        with torch.no_grad():
            output = model(src_tensor, trg_tensor)
        # Get the most probable token from the last time step
        next_token = output.argmax(dim=2)[:, -1].item()
        trg_indices.append(next_token)

        # Stop if <eos> is generated
        if next_token == word_to_idx["<eos>"]:
            break

    # Convert indices to words and filter out special tokens
    translated_sentence = [
        idx_to_word[idx] for idx in trg_indices
        if idx not in {PAD_IDX, word_to_idx["<sos>"], word_to_idx["<eos>"]}
    ]
    return translated_sentence
```

23.9. Creating our PyTorch Dataset and DataLoader

```
src_sentences = torch.tensor([encode_src_sentence(pair[0], word_to_idx, MAX_LENGTH) for pair in
  sentence_pairs])
trg_sentences = torch.tensor([encode_trg_sentence(pair[1], word_to_idx, MAX_LENGTH) for pair in
  sentence_pairs])

# Create DataLoader
BATCH_SIZE = 4
dataset = TensorDataset(src_sentences, trg_sentences)
data_loader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)
```

23.9.1. Model Hyperparameters

```
NUM_EPOCHS = 50
LEARNING_RATE = 0.0001
EMBED_SIZE = 512            # Dimensionality of embeddings and hidden layers
NUM_LAYERS = 6              # Number of Transformer layers
FORWARD_EXPANSION = 4       # Feedforward layer expansion factor
HEADS = 8                   # Number of attention heads
DROPOUT = 0.1
```

```
src_vocab_size = len(word_to_idx)
trg_vocab_size = len(word_to_idx)
model = Transformer(
    src_vocab_size=src_vocab_size,
    trg_vocab_size=trg_vocab_size,
    src_pad_idx=PAD_IDX,
    trg_pad_idx=PAD_IDX,
    embed_size=EMBED_SIZE,
    num_layers=NUM_LAYERS,
    forward_expansion=FORWARD_EXPANSION,
    heads=HEADS,
    dropout=DROPOUT,
    device=device,
    max_length=MAX_LENGTH,
).to(device)
```

```
# Initialize optimizer and loss function
optimizer = optim.AdamW(model.parameters(), lr=LEARNING_RATE)
criterion = nn.CrossEntropyLoss(ignore_index=PAD_IDX)
```

23.9.2. Testing our Model with a Dummy Tensor

```
test_sentence = ["good", "morning", "my", "friend"]
print("Input sentence:", " ".join(test_sentence))
print("Translated sentence:", " ".join(translate_sentence(test_sentence)))
```

```
Input sentence: good morning my friend
Translated sentence: world good journée good suis suis plaît good
```

23.10. Training Loop

```
for epoch in range(NUM_EPOCHS):
    model.train()
    total_loss = 0.0
    valid_batches = 0

    for src, trg in data_loader:
        src, trg = src.to(device), trg.to(device)

        # For the decoder, the input sequence is the target sequence without its last token.
        # The expected output (for loss calculation) is the target sequence without its first token.
        trg_input = trg[:, :-1]      # Decoder input (starts with <sos>)
        targets = trg[:, 1:].reshape(-1)  # Shifted target (ends with <eos>)

        optimizer.zero_grad()
        output = model(src, trg_input)
        # output shape: (batch, seq_len, vocab_size) -> reshape to (batch * seq_len, vocab_size)
        output = output.reshape(-1, output.shape[2])

        loss = criterion(output, targets)
        if torch.isnan(loss):
            continue
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        valid_batches += 1

    avg_loss = total_loss / valid_batches if valid_batches > 0 else float('inf')
    print(f"Epoch [{epoch + 1}/{NUM_EPOCHS}], Loss: {avg_loss:.4f}")

torch.save(model.state_dict(), "transformer_model.pth")
print("Model trained and saved as transformer_model.pth")
```

```
Epoch [1/50], Loss: 3.7638
Epoch [2/50], Loss: 3.0053
Epoch [3/50], Loss: 2.8880
Epoch [4/50], Loss: 2.6639
Epoch [5/50], Loss: 2.4474
Epoch [6/50], Loss: 2.2590
Epoch [7/50], Loss: 1.8693
Epoch [8/50], Loss: 1.6046
Epoch [9/50], Loss: 1.4797
Epoch [10/50], Loss: 1.2580
```

```
Epoch [11/50], Loss: 1.1304
Epoch [12/50], Loss: 0.9592
Epoch [13/50], Loss: 0.9344
Epoch [14/50], Loss: 0.8424
Epoch [15/50], Loss: 0.7005
Epoch [16/50], Loss: 0.6327
Epoch [17/50], Loss: 0.5851
Epoch [18/50], Loss: 0.4989
Epoch [19/50], Loss: 0.4842
Epoch [20/50], Loss: 0.4912
[... 21 lines omitted ...]
Epoch [42/50], Loss: 0.1041
Epoch [43/50], Loss: 0.1247
Epoch [44/50], Loss: 0.1170
Epoch [45/50], Loss: 0.0817
Epoch [46/50], Loss: 0.0975
Epoch [47/50], Loss: 0.0798
Epoch [48/50], Loss: 0.0727
Epoch [49/50], Loss: 0.0688
Epoch [50/50], Loss: 0.0735
Model trained and saved as transformer_model.pth
```

23.11. Testing our Model

Since this is a simple example without a dedicated validation or test set, we will generate variations of the input sentences for evaluation. Let's load the trained model and run some test translations.

```
model.load_state_dict(torch.load("transformer_model.pth", map_location=device, weights_only=True))
model.eval()

test_sentence = ["my", "friend"]
print("Input sentence:", " ".join(test_sentence))
print("Translated sentence:", " ".join(translate_sentence(test_sentence)))
```

```
Input sentence: my friend
Translated sentence: mon ami
```

23.12. Homework (Optional)

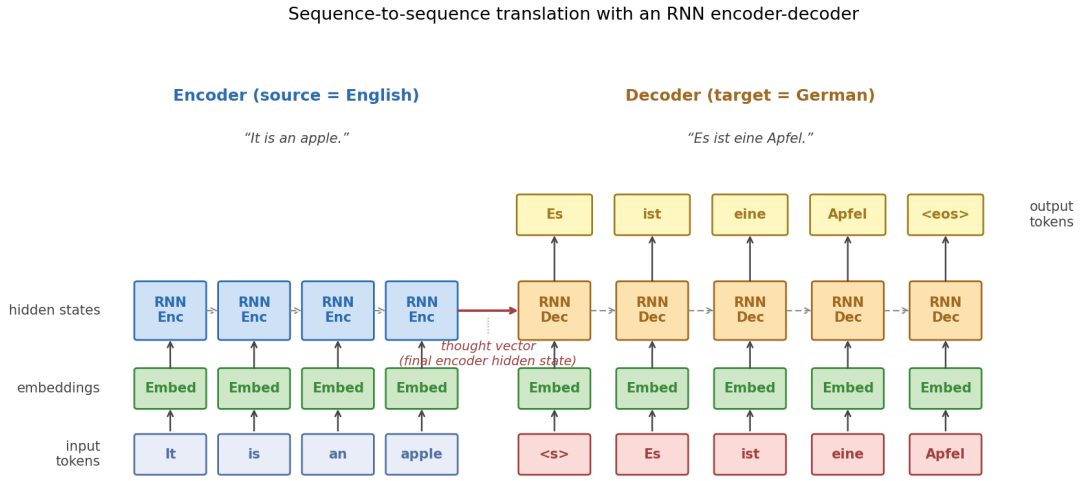
1. **Missing causal mask.** Train the model without the causal mask in decoder self-attention. Report training loss and translation accuracy, and explain why the training accuracy can reach nearly 100 percent while inference fails.
2. **Layer depth.** Train transformers with `num_layers = 2, 4, 8`. Report validation loss, parameters, and epoch time. Pick the best depth for this tiny translation dataset and justify.
3. **Label smoothing.** Add `label_smoothing=0.1` to the cross-entropy loss. Report the change in validation loss and qualitative translation quality on five test pairs.
4. **Warmup schedule.** Implement the original transformer warmup `lr = d_model^{-0.5} * min(step^{-0.5}, step * warmup^{-1.5})`. Compare with a cosine schedule at matched compute.
5. **Bigger vocabulary.** Retrain on a bilingual corpus with 2000+ distinct words (e.g. a slice of OPUS or Tatoeba). Report BLEU and the impact of vocabulary size on memory.

6. **Weight sharing.** The original transformer shares weights between the target-embedding, the source-embedding and the output projection. Implement this tie and report the parameter-count and accuracy impact.

Chapter 24

NMT with Transformers

Authors: Máté Badó Miklós
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1kv4v4N0dNUF06KgCYaaKsdqPTJbJ_eUa?usp=sharing



Neural machine translation utilizes end-to-end deep learning models, typically based on encoder-decoder architectures enhanced by attention mechanisms, to map sequences from a source language directly to a target language. This approach enables the model to capture long-range dependencies and contextual subtleties within the text, thereby producing translations that are both fluent and semantically accurate, surpassing the performance of traditional statistical or rule-based translation systems.

24.1. Necessary Imports

```
import subprocess
import os
import pandas as pd
import nltk
import math
import copy
import numpy as np
import torch
import torch.nn as nn
import torch.nn.init as init
import matplotlib.pyplot as plt
from torch.optim import Adam, AdamW
from IPython.display import Image as IMG
from nltk.translate.bleu_score import sentence_bleu, corpus_bleu
from torch.utils.data import Dataset, DataLoader
from tqdm.notebook import tqdm
```

```
from nltk.tokenize import WordPunctTokenizer
from collections import Counter

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```



24.2. Data

For this practice, we will use a **bilingual dataset** containing sentences in multiple languages, available for download from Tatoeba and ManyThings. We will then filter this dataset to extract only the **Hungarian** and **English** sentence pairs, allowing us to focus on building and evaluating a neural machine translation model for these two languages.

```
%%bash
export PYTHONWARNINGS="ignore"
if [ -d "data" ]; then
    echo "data directory already exists. Skipping download."
else
    echo "data directory does not exist. Downloading..."
    mkdir -p data
    wget https://downloads.tatoeba.org/exports/sentences.tar.bz2 --quiet -P data
    tar xvjf data/sentences.tar.bz2 -C data
    wget https://downloads.tatoeba.org/exports/links.tar.bz2 --quiet -P data
    tar xvjf data/links.tar.bz2 -C data
    rm -rf data/sentences.tar.bz2
    rm -rf data/links.tar.bz2
fi
```

```
data directory does not exist. Downloading...
sentences.csv
links.csv
```

```
#ISO 639-3 code for a source language
source_lang = 'eng'
target_lang = 'hun'
data_dir = './'

# read all data
sentences = pd.read_csv('data/sentences.csv', names=['id', 'lang', 'text'], header=None, delimiter='\t')
links = pd.read_csv('data/links.csv', names=['sent_id', 'tran_id'], header=None, delimiter='\t')

# extract source - target connected
source_sentences = sentences[sentences.lang == source_lang]
target_sentences = sentences[sentences.lang == target_lang]
```

```
sentences.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 13291595 entries, 0 to 13291594
Data columns (total 3 columns):
 #   Column  Dtype
---  ------  -----
 0   id      int64
 1   lang    object
 2   text    object
```

```
dtypes: int64(1), object(2)
memory usage: 304.2+ MB
```

24.3. Data Exploration

Let's find out how each of the english sentences are linked to their hungarian variants through the `links` dataset.

```
source_sentences.head(5)
```

```
id lang                                          text
1232  1276  eng                            Let's try something.
1233  1277  eng                           I have to go to sleep.
1235  1280  eng  Today is June 18th and it is Muiriel's birthday!
1236  1282  eng                              Muiriel is 20 now.
1237  1283  eng                     The password is "Muiriel".
```

```
source_sentences.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 2013074 entries, 1232 to 13291593
Data columns (total 3 columns):
 #   Column  Dtype
---  ------  -----
 0   id      int64
 1   lang    object
 2   text    object
dtypes: int64(1), object(2)
memory usage: 61.4+ MB
```

```
target_sentences.head(5)
```

```
id lang                                                text
313802  331032  hun                                    Ez nem lep meg.
363632  382214  hun  Minden emberi lény szabadon születik és egyenl...
381965  400937  hun                                          Hogy vagy?
382119  401101  hun                                         Köszönöm!
382122  401104  hun                                 Köszönöm szépen!
```

```
target_sentences.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Index: 429436 entries, 313802 to 13291286
Data columns (total 3 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   id      429436 non-null  int64
 1   lang    429436 non-null  object
 2   text    429436 non-null  object
dtypes: int64(1), object(2)
memory usage: 13.1+ MB
```

```
links.head(5)
```

```
sent_id  tran_id
0        1     2481
1        1     5350
```

```
2        1   180624
3        1   344899
4        1   380381
```

```
ID = 1276

# We have around 108 translations for a given sentence
print(links[links['sent_id'] == ID])
```

```
sent_id   tran_id
15712     1276        77
15713     1276      2481
15714     1276      3091
15715     1276      4702
15716     1276      5350
...        ...       ...
15820     1276  13037538
15821     1276  13096615
15822     1276  13096621
15823     1276  13498738
15824     1276  13632536

[113 rows x 2 columns]
```

```
# Filter out the english sentences with ID of 1276
english_sentence = source_sentences[source_sentences['id'] == ID]
# Find all translations for sentence with ID of 1276 (108 items)
link_rows = links[links['sent_id'] == ID]
# Filter out all the translation IDs from the `link_rows` DataFrame
target_ids = link_rows['tran_id'].unique()
# Find the hungarian sentences whos ID is in the list of `target_ids`
hungarian_sentence = target_sentences[target_sentences['id'].isin(target_ids)]
```

```
print(english_sentence)
print()
print(hungarian_sentence)
```

```
id lang                 text
1232  1276  eng  Let's try something.

              id lang                    text
3524818  3753714  hun  Próbáljunk ki valamit.
```

24.4. Creating a Joint Dataset of Translations

```
source_sentences = source_sentences.merge(links, left_on='id', right_on='sent_id')
```

```
source_sentences.head(5)
```

```
id lang                 text  sent_id  tran_id
0  1276  eng  Let's try something.     1276       77
1  1276  eng  Let's try something.     1276     2481
2  1276  eng  Let's try something.     1276     3091
3  1276  eng  Let's try something.     1276     4702
4  1276  eng  Let's try something.     1276     5350
```

```
source_sentences.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7563791 entries, 0 to 7563790
Data columns (total 5 columns):
 #   Column   Dtype
---  ------   -----
 0   id       int64
 1   lang     object
 2   text     object
 3   sent_id  int64
 4   tran_id  int64
dtypes: int64(3), object(2)
memory usage: 288.5+ MB
```

```
bilang_sentences = source_sentences.merge(target_sentences, left_on='tran_id', right_on='id',
   suffixes=[source_lang, target_lang])
```

```
bilang_sentences.head(5)
```

```
ideng langeng                                        texteng  sent_id  \
0   1276     eng                             Let's try something.     1276
1   1277     eng                           I have to go to sleep.     1277
2   1277     eng                           I have to go to sleep.     1277
3   1277     eng                           I have to go to sleep.     1277
4   1280     eng  Today is June 18th and it is Muiriel's birthday!     1280

   tran_id    idhun langhun                                          texthun
0  3753714  3753714     hun                            Próbáljunk ki valamit.
1   414529   414529     hun                               Mennem kell aludni.
2   522344   522344     hun                               Aludni kell mennem.
3   534629   534629     hun                                 Le kell feküdnöm.
4  7902836  7902836     hun  Ma június 18-a van, és ez Muiriel születésnapja!
```

```
bilang_sentences.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 179471 entries, 0 to 179470
Data columns (total 8 columns):
 #   Column   Non-Null Count   Dtype
---  ------   --------------   -----
 0   ideng    179471 non-null  int64
 1   langeng  179471 non-null  object
 2   texteng  179471 non-null  object
 3   sent_id  179471 non-null  int64
 4   tran_id  179471 non-null  int64
 5   idhun    179471 non-null  int64
 6   langhun  179471 non-null  object
 7   texthun  179471 non-null  object
dtypes: int64(4), object(4)
memory usage: 11.0+ MB
```

```
bilang_sentences = bilang_sentences[['text' + source_lang, 'text' + target_lang]]
```

```
bilang_sentences.head(5)
```

```
texteng  \
0                             Let's try something.
1                           I have to go to sleep.
```

```
2                            I have to go to sleep.
3                            I have to go to sleep.
4  Today is June 18th and it is Muiriel's birthday!

                                          texthun
0                            Próbáljunk ki valamit.
1                               Mennem kell aludni.
2                               Aludni kell mennem.
3                                 Le kell feküdnöm.
4  Ma június 18-a van, és ez Muiriel születésnapja!
```

24.5. Save the New Subset

```
file_name = os.path.join(data_dir, '{source}-{target}.csv'.format(source=source_lang, target=target_lang))
bilang_sentences.to_csv(file_name, index=False, sep='\t')
```

```
!rm -rf data
```

24.6. Let's Load it Back in

```
corpus = pd.read_csv(os.path.join(data_dir, '{}-{}.csv'.format(source_lang, target_lang)), delimiter='\t')
print("Number of sentence pairs in the corpus:", len(corpus))
```

```
Number of sentence pairs in the corpus: 179471
```

```
corpus.head(5)
```

```
texteng  \
0                              Let's try something.
1                            I have to go to sleep.
2                            I have to go to sleep.
3                            I have to go to sleep.
4  Today is June 18th and it is Muiriel's birthday!

                                          texthun
0                            Próbáljunk ki valamit.
1                               Mennem kell aludni.
2                               Aludni kell mennem.
3                                 Le kell feküdnöm.
4  Ma június 18-a van, és ez Muiriel születésnapja!
```

24.7. Creating our Vocabulary

Creating a **vocabulary** is a crucial aspect of **natural language processing**, as it enables the efficient mapping between textual tokens and their corresponding numerical representations. Special tokens such as `SOS` (start-of-sentence), `EOS` (end-of-sentence), `UNK` (unknown), and `PAD` (padding) are essential for managing sequence boundaries, handling words that are not present in the vocabulary, and maintaining consistent input lengths for model training and text generation.

```
class Vocab:
    """
    A simple vocabulary class for mapping words to indices and vice versa.

    Attributes:
```

```
        index2word (dict): Maps indices to words.
        word2index (dict): Maps words to indices.
        Special tokens:
            SOS_token (str): Start-of-sentence token.
            EOS_token (str): End-of-sentence token.
            UNK_token (str): Unknown token for out-of-vocabulary words.
            PAD_token (str): Padding token.
    """
    def __init__(self):
        # Define special tokens and their fixed indices
        self.SOS_token = '<sos>'
        self.EOS_token = '<eos>'
        self.UNK_token = '<unk>'
        self.PAD_token = '<pad>'

        self.PAD_idx = 0
        self.SOS_idx = 1
        self.EOS_idx = 2
        self.UNK_idx = 3

        self.index2word = {
            self.SOS_idx: self.SOS_token,
            self.EOS_idx: self.EOS_token,
            self.UNK_idx: self.UNK_token,
            self.PAD_idx: self.PAD_token
        }
        self.word2index = {word: idx for idx, word in self.index2word.items()}

    def index_words(self, words):
        """
        Adds a list of words to the vocabulary.

        Args:
            words (iterable): An iterable of words to be added.
        """
        for word in words:
            self.index_word(word)

    def index_word(self, word):
        """
        Adds a single word to the vocabulary if it's not already present.

        Args:
            word (str): The word to add.
        """
        if word not in self.word2index:
            idx = len(self.index2word)
            self.word2index[word] = idx
            self.index2word[idx] = word

    def __len__(self):
        """
        Returns:
            int: Total number of words in the vocabulary.
        """
        assert len(self.index2word) == len(self.word2index), "Mappings are inconsistent."
        return len(self.index2word)

    def decode_words(self, indices):
        """
```

```
        Converts a list of indices into their corresponding words.

        Args:
            indices (list or iterable): List of word indices.

        Returns:
            list: List of words corresponding to the indices. Unknown indices are replaced with UNK_token.
        """
        return [self.index2word.get(i, self.UNK_token) for i in indices]

    def to_file(self, filename):
        """
        Saves the vocabulary (excluding special tokens) to a file with one word per line.

        Args:
            filename (str): Path to the output file.
        """
        # Exclude the first four special tokens
        words = [word for idx, word in sorted(self.index2word.items()) if idx >= 4]
        try:
            with open(filename, 'w', encoding='utf-8') as f:
                for word in words:
                    f.write(word + '\n')
        except IOError as e:
            print(f"Error saving vocabulary to {filename}: {e}")

    @classmethod
    def from_file(cls, filename):
        """
        Loads a vocabulary from a file where each line contains one word.

        Args:
            filename (str): Path to the file containing vocabulary words.

        Returns:
            Vocab: A new Vocab instance with the words loaded.
        """
        vocab = cls()  # Initialize with special tokens
        try:
            with open(filename, 'r', encoding='utf-8') as f:
                words = [line.strip() for line in f if line.strip()]
                vocab.index_words(words)
        except IOError as e:
            print(f"Error loading vocabulary from {filename}: {e}")
        return vocab

    def __repr__(self):
        """
        Returns a string representation of the vocabulary.
        """
        return f"Vocab(size={len(self)}, special_tokens={list(self.index2word.values())[:4]})"
```

24.8. Tokenization

Tokenization is a critical step in NLP that involves splitting raw text into smaller, meaningful units (`tokens`) such as words. This process allows us to convert text into numerical representations that can be fed into machine learning models. Moreover, thorough text preprocessing, including cleaning (e.g.,

removing HTML tags, punctuation, digits, and stopwords), helps reduce noise and improves model performance.

```
def preprocess_corpus(sents, tokenizer, min_word_count):
    """
    Preprocesses a corpus by tokenizing sentences, converting them to lowercase,
    and replacing words that occur less frequently than 'min_word_count' with the UNK token.

    Args:
        sents (iterable of str): Iterable of sentence strings.
        tokenizer (callable): A function that splits a sentence into tokens.
        min_word_count (int): Minimum frequency a word must have to be retained.

    Returns:
        list of list of str: A list where each element is a tokenized sentence.
    """
    # Tokenize and lowercase each sentence
    sents_tokenized = [[w.lower() for w in tokenizer(sent)] for sent in sents]

    # Count word frequencies across the entire corpus
    word_counts = Counter(token for sent in sents_tokenized for token in sent)

    # Replace infrequent words with the UNK token
    processed = []
    for tokens in sents_tokenized:
        new_tokens = [token if word_counts[token] >= min_word_count else '<unk>' for token in tokens]
        processed.append(new_tokens)

    return processed

def read_vocab(sents):
    """
    Builds a vocabulary from a list of tokenized sentences.

    Args:
        sents (iterable of list of str): Iterable of tokenized sentences.

    Returns:
        Vocab: A Vocab object populated with words from the sentences.
    """
    vocab = Vocab()
    for sent in sents:
        vocab.index_words(sent)
    return vocab
```

The `WordPunctTokenizer` is a simple yet effective tool that splits text into individual word and punctuation tokens. For this English-Hungarian machine translation task, it is particularly useful because it uniformly handles the tokenization of both languages.

```
max_length = 10
min_word_count = 1

tokenizers = {
    'eng': WordPunctTokenizer().tokenize,
    'hun': WordPunctTokenizer().tokenize
}

source_sents = preprocess_corpus(corpus['text' + source_lang], tokenizers[source_lang], min_word_count)
target_sents = preprocess_corpus(corpus['text' + target_lang], tokenizers[target_lang], min_word_count)
```

```
# Filter out sentence pairs where either sentence exceeds the maximum length.
filtered_pairs = sorted({
    (tuple(s), tuple(t))
    for s, t in zip(source_sents, target_sents)
    if len(s) <= max_length and len(t) <= max_length
})
source_sents, target_sents = zip(*filtered_pairs)

source_vocab = read_vocab(source_sents)
target_vocab = read_vocab(target_sents)

# Save the vocabularies to files.
source_vocab.to_file(os.path.join(data_dir, f'{source_lang}.vocab.txt'))
target_vocab.to_file(os.path.join(data_dir, f'{target_lang}.vocab.txt'))
```

```
# Print a summary of the processed corpus.
print('Corpus length: {}\nSource vocabulary size: {}\nTarget vocabulary size: {}'.format(
    len(source_sents), len(source_vocab.word2index), len(target_vocab.word2index)
))

examples = list(zip(source_sents, target_sents))[80:90]
for src, tgt in examples:
    print(f'Source: "{" ".join(src)}", target: "{" ".join(tgt)}"')
```

```
Corpus length: 152879
Source vocabulary size: 15780
Target vocabulary size: 57751
Source: "a cheetah is just a big cat .", target: "a gepárd csak egy nagy macska
."
Source: "a child is missing .", target: "egy gyerek hiányzik ."
Source: "a child is missing .", target: "eltűnt egy gyerek ."
Source: "a child needs a mother .", target: "a gyereknek anyára van szüksége ."
Source: "a child needs a mother .", target: "egy gyereknek szüksége van anyára
."
Source: "a chinch bug is climbing the window .", target: "egy poloska mászik az
ablakon ."
Source: "a clan of fairies lives under there .", target: "tündérek serege él
azalatt ."
Source: "a cloud passed the sun .", target: "egy felhő haladt el a nap előtt ."
Source: "a cobra bit tom .", target: "tomit megmarta egy kobra ."
Source: "a coconut fell on tom ' s head .", target: "a kókuszdió tamás fejére
zuhant ."
```

```
print("Number of source sentences (english):", len(source_sents))
print("Number of target sentences (hungarian):", len(target_sents))
```

```
Number of source sentences (english): 152879
Number of target sentences (hungarian): 152879
```

24.8.1. Eliminating Duplicate Entries

```
def aggregate_source_target_pairs(source_sents, target_sents):
    """
    Group target sentences by their corresponding unique source sentence.
```

```
    Args:
        source_sents (iterable): Source sentences.
        target_sents (iterable): Target sentences.

    Returns:
        tuple: (unique_sources, grouped_targets) where unique_sources is a tuple of unique source sentences,
               and grouped_targets is a tuple of lists of target sentences for each source.
    """
    source_to_target = {}
    for source, target in zip(source_sents, target_sents):
        source_to_target.setdefault(source, []).append(target)

    unique_sources = tuple(source_to_target.keys())
    grouped_targets = tuple(source_to_target.values())
    return unique_sources, grouped_targets

unique_source, unique_targets = aggregate_source_target_pairs(source_sents, target_sents)
print("Number of unique source sentences:", len(unique_source))
print("Number of unique target sentences:", len(unique_targets))
```

```
Number of unique source sentences: 125041
Number of unique target sentences: 125041
```

```
print(target_sents)
print(unique_targets)
```

```
(('a', 'radioaktivitás', 'kémiai', 'fogalom', '.'), ('a', '100', 'évet',
'évszázadnak', 'nevezik', '.'), ('1975', 'az', 'az', 'év', ',', 'amikor',
'születtem', '.'), ('kétezerhuszo ... [6853315 chars truncated]
([('a', 'radioaktivitás', 'kémiai', 'fogalom', '.')], [('a', '100', 'évet',
'évszázadnak', 'nevezik', '.')], [('1975', 'az', 'az', 'év', ',', 'amikor',
'születtem', '.')], [('kétez ... [7103397 chars truncated]
```

```
examples_filtered = list(zip(unique_source, unique_targets))[80:100]
for src, tgts in examples_filtered:
    src_text = " ".join(src)
    tgt_texts = [ " ".join(tgt) for tgt in tgts ]
    print(f'Source: "{src_text}", Targets: {tgt_texts}')
```

```
Source: "a cold wind blows .", Targets: ['hideg szél fúj .']
Source: "a cold wind was blowing .", Targets: ['hideg szél fújt .']
Source: "a comma isn ' t necessary here .", Targets: ['nem kell ide vessző .']
Source: "a commission !", Targets: ['jutalék !']
Source: "a compound word consist of two smaller words .", Targets: ['egy
összetett szó két szóból áll .']
Source: "a computer is a wonderful thing .", Targets: ['egy számítógép
csodálatos dolog .']
Source: "a convergent sequence is bounded .", Targets: ['egy konvergens sorozat
korlátos is .']
Source: "a cool wind is blowing from the sea .", Targets: ['hűvös szél fúj a
tenger felől .']
Source: "a corpse was found at the bayou just yesterday .", Targets: ['egy
holttestet találtak tegnap a mocsárban .']
Source: "a couple more !", Targets: ['még egy párat !']
Source: "a cow has a long tail .", Targets: ['hosszú farka van a teheneknek .']
Source: "a crocodile ate thomas .", Targets: ['egy krokodil megette tamást .']
```

```
Source: "a crocodile ate tom .", Targets: ['egy krokodil megette tamást .']
Source: "a crocodile has eaten tom .", Targets: ['egy krokodil megette tamást
.']
Source: "a crow is as black as coal .", Targets: ['a varjú olyan fekete , mint a
szén .']
Source: "a crying girl opened the door .", Targets: ['egy síró lány nyitott
ajtót .', 'egy síró lány nyitotta ki az ajtót .', 'síró lany nyitott ajtót .']
Source: "a cube has six sides .", Targets: ['a kockának hat oldala van .']
Source: "a cunt remains a cunt !", Targets: ['a picsa picsa marad .']
Source: "a cup of coffee , please .", Targets: ['egy csésze kávét kérek .']
Source: "a cup of coffee costs a krone .", Targets: ['egy csésze kávé egy
koronába kerül .']
```

24.9. Splitting the Date

We partition our aggregated sentence pair corpus into three subsets: **80% for training**, which the model will learn from, **6% for validation**, used to fine-tune hyperparameters and monitor performance, and **14% for testing**, reserved for evaluating the final model's generalization ability.

To structure our training data effectively, we **unwrap multi-target sentence pairs**, ensuring that each unique source sentence is paired individually with each of its corresponding target translations. This means that if a source sentence has multiple valid translations, we create separate training instances for each one:

(Source) "The weather is nice." -> (Target) "Az idő szép."

(Source) "The weather is nice." -> (Target) "Szép az idő."

(Source) "The weather is nice." -> (Target) "Kellemes az időjárás."

However, for the **validation set**, we preserve the original multi-target structure. This allows us to evaluate model predictions against all valid translations, ensuring a fairer and more flexible assessment of translation quality:

(Source) "The weather is nice." -> (Targets) ["Az idő szép.", "Szép az idő.", "Kellemes az időjárás."]

By keeping multiple references intact in validation, we can measure how well the model produces **any of the correct translations** rather than being restricted to a single expected output.

```python
seed = 42
np.random.seed(seed)

source_length = len(unique_source)
indices = np.random.permutation(source_length)

train_split = int(source_length * 0.8)
valid_split = int(source_length * 0.86)

training_indices = indices[:train_split]
valid_indices = indices[train_split:valid_split]
test_indices = indices[valid_split:]
```

```
training_source = [unique_source[i] for i in training_indices]
valid_source = [unique_source[i] for i in valid_indices]
test_source = [unique_source[i] for i in test_indices]

training_target = [unique_targets[i] for i in training_indices]
valid_target = [unique_targets[i] for i in valid_indices]
test_target = [unique_targets[i] for i in test_indices]

# -------------------------
# 2. UNWRAP MULTI-TARGET TRAINING INSTANCES
# -------------------------
train_source_unwrapped = []
train_target_unwrapped = []

for src, tgt_list in zip(training_source, training_target):
    for tgt in tgt_list:
        train_source_unwrapped.append(src)
        train_target_unwrapped.append(tgt)

training_source = train_source_unwrapped
training_target = train_target_unwrapped
```

Visualizing Training Pairs

```
for i in range(10, 20):
    print(f"Training Source: {' '.join(training_source[i])}")
    print(f"Training Target: {' '.join(training_target[i])}\n")
```

```
Training Source: you ' ve got it in the wrong gear .
Training Target: rossz sebességbe kapcsoltál .

Training Source: you ' ve got it in the wrong gear .
Training Target: rossz sebességbe váltottál .

Training Source: tom doesn ' t like working here .
Training Target: tamás nem szeret itt dolgozni .

Training Source: tell tom to give you some scissors .
Training Target: mondd meg tomnak , hogy adjon neked egy ollót .

Training Source: the man ' s name was francisco pizarro .
Training Target: a férfi neve francisco pizarro volt .

Training Source: i need my glasses .
Training Target: szükségem van a szemüvegemre .

Training Source: mr . smith left japan this morning .
Training Target: smith úr ma reggel elhagyta japánt .

Training Source: everyone makes mistakes .
Training Target: emberből vagyunk , hibázhatunk .

Training Source: everyone makes mistakes .
Training Target: mindenki hibázik .

Training Source: everyone makes mistakes .
Training Target: mindenki követ el hibákat .
```

24.10. Padding Sequences and Converting them to Tensors

Before feeding our data into the model, we need to convert sentences into **fixed-length tensors of word indices**. This ensures that our model can process batches efficiently, as deep learning frameworks like PyTorch require **uniform input sizes** for batch operations.

Since sentences can have varying lengths, we use **padding** to bring them to a consistent length. We apply **end-padding** (rather than prefix-padding) so that shorter sentences do not affect the structure of the earlier words, which can be particularly beneficial for sequence-based models like LSTMs. Additionally, each sequence is **wrapped with special tokens**: `(start-of-sequence) at the beginning` and (end-of-sequence) at the end. This helps the decoder understand when a sentence starts and when it should stop generating words.

```
EOS_idx = target_vocab.word2index[target_vocab.EOS_token]
SOS_idx = target_vocab.word2index[target_vocab.SOS_token]
PAD_idx = target_vocab.word2index[target_vocab.PAD_token]

def indexes_from_sentence(vocab, sentence):
    """
    Converts a tokenized sentence into a list of corresponding indices from the vocabulary.

    Args:
        vocab (Vocab): Vocabulary object that maps words to indices.
        sentence (list of str): A tokenized sentence.

    Returns:
        list of int: List of word indices corresponding to the input sentence.
    """
    return [vocab.word2index[word] for word in sentence]

def tensor_from_sentence(vocab, sentence, max_seq_length):
    """
    Converts a tokenized sentence into a fixed-length tensor of token indices with padding.

    Args:
        vocab (Vocab): Vocabulary object containing word-index mappings.
        sentence (list of str): Tokenized sentence to be converted.
        max_seq_length (int): Maximum length for the sequence, including <SOS> and <EOS>.

    Returns:
        torch.Tensor: LongTensor of shape (max_seq_length,) containing token indices.
    """
    indexes = indexes_from_sentence(vocab, sentence)
    indexes.append(EOS_idx)  # Append end-of-sequence token
    indexes.insert(0, SOS_idx)  # Insert start-of-sequence token

    # Pad sequence to max length
    if len(indexes) < max_seq_length:
        indexes += [PAD_idx] * (max_seq_length - len(indexes))

    return torch.LongTensor(indexes).to(device)

def tensors_from_pair(source_sent, target_sent, max_seq_length):
    """
    Creates a tuple of (source_tensor, target_tensor) for a given source-target sentence pair.

    Args:
```

```
        source_sent (list of str): Tokenized source sentence.
        target_sent (list of str): Tokenized target sentence.
        max_seq_length (int): Maximum sequence length for both source and target.

    Returns:
        tuple: (source_tensor, target_tensor), each of shape (max_seq_length,).
    """
    source_tensor = tensor_from_sentence(source_vocab, source_sent, max_seq_length)
    target_tensor = tensor_from_sentence(target_vocab, target_sent, max_seq_length)
    return source_tensor, target_tensor

# Define sequence length including <SOS> and <EOS>
max_seq_length = max_length + 2

training = []
for source_sent, target_sent in zip(training_source, training_target):
    training.append(tensors_from_pair(source_sent, target_sent, max_seq_length))

x_training, y_training = zip(*training)
x_training = torch.stack(x_training).to(device)
y_training = torch.stack(y_training).to(device)

validation = []
for src, tgt_list in zip(valid_source, valid_target):
    source_tensor = tensor_from_sentence(source_vocab, src, max_seq_length)

    # Convert all target translations for this source into tensors
    target_tensors = [tensor_from_sentence(target_vocab, tgt, max_seq_length) for tgt in tgt_list]

    # Append a single (source_tensor, list_of_target_tensors) pair
    validation.append((source_tensor, target_tensors))

x_valid, y_valid = zip(*validation)
x_valid = torch.stack(x_valid).to(device)
y_valid = list(y_valid)

testing = []
for src, tgt_list in zip(test_source, test_target):
    source_tensor = tensor_from_sentence(source_vocab, src, max_seq_length)
    # Convert all target translations for this source into tensors
    target_tensors = [tensor_from_sentence(target_vocab, tgt, max_seq_length) for tgt in tgt_list]

    # Append a single (source_tensor, list_of_target_tensors) pair
    testing.append((source_tensor, target_tensors))

x_test, y_test = zip(*testing)
x_test = torch.stack(x_test).to(device)
y_test = list(y_test)
```

```
print("Training Sentence Shape: ", x_training.shape)
print("Training Target Shape: ", y_training.shape)
print("Validation Sentence Shape: ", x_valid.shape)
print("Validation Target Count: ", len(y_valid))
print("Test Sentence Shape: ", x_test.shape)
```

```
Training Sentence Shape:  torch.Size([122401, 12])
Training Target Shape:  torch.Size([122401, 12])
Validation Sentence Shape:  torch.Size([7503, 12])
Validation Target Count:  7503
Test Sentence Shape:  torch.Size([17506, 12])
```

24.11. Creating a Dataset and a Dataloader

To efficiently feed our data into the model, we use **PyTorch's Dataset and DataLoader classes**. Since our training and validation data have different structures, **one-to-one mappings in training** vs. **one-to-many mappings in validation**, we create a custom dataset to handle both cases.

The **`TranslationDataset`** class is designed to: - Load **preprocessed source and target tensors**. - Support both **training (single target per source)** and **validation (multiple targets per source).**

Additionally, we define a **custom `collate_fn` function**, which: - **Stacks** source sentences into a batch tensor. - **Handles validation targets properly**, preserving multiple references as lists instead of flattening them.

```
class TranslationDataset(Dataset):
    """
    PyTorch Dataset for loading preprocessed sentence tensor pairs.

    Supports:
    - Training: Each source sentence has one corresponding target sentence.
    - Validation: Each source sentence has multiple correct target sentences.

    Args:
        source_tensor (torch.Tensor): Tensor containing source sentences.
        target_tensor (torch.Tensor or list): Tensor of target sentences (for training) or list of multiple
   target tensors (for validation).
        is_validation (bool): If True, the dataset contains multiple target translations per source.
    """
    def __init__(self, source_tensor, target_tensor):
        self.source_tensor = source_tensor
        self.target_tensor = target_tensor

    def __len__(self):
        return len(self.source_tensor)

    def __getitem__(self, idx):
        return self.source_tensor[idx], self.target_tensor[idx]

def collate_fn(batch):
    """
    Collate function to handle batching for both training and validation datasets.

    Args:
        batch (list of tuples): Each tuple contains (source_tensor, target_tensor or list_of_target_tensors).

    Returns:
        - torch.Tensor: Batch of source sentences.
        - torch.Tensor (for training) or list of lists (for validation).
    """
    sources, targets = zip(*batch)

    # Stack sources as normal (tensor batch)
    sources = torch.stack(sources)

    # If targets are lists (validation case), keep them as lists
    if isinstance(targets[0], list):
        return sources, list(targets)  # List of target lists
```

```
    # Otherwise, stack targets normally (training case)
    return sources, torch.stack(targets)
```

```
# Create Dataset Instances
train_dataset = TranslationDataset(x_training, y_training)
valid_dataset = TranslationDataset(x_valid, y_valid)
test_dataset = TranslationDataset(x_test, y_test)
# Create DataLoaders
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True, collate_fn=collate_fn)
valid_loader = DataLoader(valid_dataset, batch_size=64, shuffle=True, collate_fn=collate_fn)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=True, collate_fn=collate_fn)
```

Visualizing our Training Dataset

```
for source_batch, target_batch in train_loader:
    print(f"Training Batch Shapes: Source {source_batch.shape}, Target {target_batch.shape}\n")

    # Convert first 3 examples into readable text
    for i in range(min(3, len(source_batch))):
        src_text = " ".join(
            [source_vocab.index2word[idx.item()] for idx in source_batch[i]]
        )
        tgt_text = " ".join(
            [target_vocab.index2word[idx.item()] for idx in target_batch[i]]
        )
        print(f"Source: {src_text}")
        print(f"Target: {tgt_text}\n")
    break
```

```
Training Batch Shapes: Source torch.Size([64, 12]), Target torch.Size([64, 12])

Source: <sos> everyone seems to be hungry but me . <eos> <pad> <pad>
Target: <sos> úgy néz ki , mindenki éhes engem kivéve . <eos> <pad>

Source: <sos> come if you can . <eos> <pad> <pad> <pad> <pad> <pad>
Target: <sos> gyere , ha tudsz . <eos> <pad> <pad> <pad> <pad> <pad>

Source: <sos> they will announce the winner tomorrow . <eos> <pad> <pad> <pad>
Target: <sos> holnap hirdetik ki a nyertest . <eos> <pad> <pad> <pad> <pad>
```

Visualizing our Validation Dataset

```
for source_batch, target_batch in valid_loader:
    print(f"Validation Batch Shapes: Source {source_batch.shape}, Targets per Source: {len(target_batch)}\n")

    # Process the first 10 examples for readability
    for i in range(min(10, len(source_batch))):
        src_text = " ".join(
            [source_vocab.index2word[idx.item()] for idx in source_batch[i]]
        )
        tgt_texts = [
            " ".join([target_vocab.index2word[idx.item()] for idx in tgt])
            for tgt in target_batch[i]
        ]

        print(f"Source:\n{src_text}")
        print(f"Possible Translations:")
        for j, tgt in enumerate(tgt_texts, start=1):
```

```
            print(f"{tgt}")
        print()
    break
```

```
Validation Batch Shapes: Source torch.Size([64, 12]), Targets per Source: 64

Source:
<sos> the rich make the laws . <eos> <pad> <pad> <pad> <pad>
Possible Translations:
<sos> a törvényeket a gazdagok hozzák . <eos> <pad> <pad> <pad> <pad>

Source:
<sos> there ' s nothing more important than friends . <eos> <pad>
Possible Translations:
<sos> semmi sem fontosabb , mint a barátok . <eos> <pad> <pad>

Source:
<sos> she used to love him . <eos> <pad> <pad> <pad> <pad>
Possible Translations:
<sos> régen szerette őt . <eos> <pad> <pad> <pad> <pad> <pad> <pad>

Source:
<sos> we aren ' t kids anymore . <eos> <pad> <pad> <pad>
Possible Translations:
[... 21 lines omitted ...]

Source:
<sos> she hates secrets . <eos> <pad> <pad> <pad> <pad> <pad> <pad>
Possible Translations:
<sos> utálja a titkolózást . <eos> <pad> <pad> <pad> <pad> <pad> <pad>

Source:
<sos> the car ' s price is very high . <eos> <pad>
Possible Translations:
<sos> borsos ára van az autónak . <eos> <pad> <pad> <pad> <pad>
```

24.12. From RNN seq2seq to transformer seq2seq

Swapping the RNN encoder and decoder from the previous chapter for transformer blocks changes four things.

No hidden state to pass step-by-step. RNNs process one token at a time and carry a recurrent hidden state. Transformers process the whole sequence in parallel through self-attention, so training and inference are much better parallelised on modern hardware.

Positional encoding becomes mandatory. Because attention is permutation-invariant over its input, the transformer must be told where each token sits. The original paper uses sinusoids of different frequencies. Learned embeddings work equally well in practice. Without positional encoding, a transformer treats "the cat sat on the mat" the same as "mat the on sat cat the".

Three distinct masks. Encoder self-attention only needs the source padding mask. Decoder self-attention also needs a causal (triangular) mask so the decoder cannot peek at future target tokens. Decoder cross-attention uses the source padding mask applied to the encoder output. Getting any of these wrong is silent: training still runs, but inference behaves strangely.

Faster convergence, bigger appetite. On the translation task below, the transformer typically matches the RNN baseline's validation loss in fewer epochs, then keeps improving. It also uses more parameters and more GPU memory per batch, so the lookback (sequence length), batch size, and hidden dimensions need to be picked together.

24.13. Model

The **encoder** in an Encoder-Decoder network is a **Recurrent Neural Network (RNN)**, designed to process sequential data such as sentences. Unlike standard feedforward networks, RNNs **maintain a hidden state** that allows them to capture dependencies between elements in a sequence by using the output from the previous timestep as additional input for the next timestep.

However, **vanilla RNNs struggle with long sequences** due to vanishing and exploding gradient issues, making it difficult to learn dependencies over extended text. To address this, we use **Long Short-Term Memory (LSTM) networks**, which include a gating mechanism to regulate information flow, enabling them to **preserve long-range dependencies** effectively. PyTorch provides an efficient implementation via `nn.LSTM`, which we leverage in this model.

To further improve the encoder's ability to **capture relationships in text**, we employ a **bidirectional LSTM**. Unlike standard LSTMs, which process input in a left-to-right manner, bidirectional LSTMs **process sequences in both directions**, allowing the network to **capture both past and future context** for each word.

Additionally, we incorporate an **embedding layer** (`nn.Embedding`) to convert word indices into dense vector representations. This allows words to be represented in a **continuous space**, where semantically similar words are positioned closer together. In this implementation, we train these embeddings from scratch using the machine translation supervision signal. However, one could also use **pre-trained word embeddings** (such as FastText or Word2Vec) for better initialization, particularly when working with low-resource languages or small datasets.

24.14. Encoder

```
class EncoderRNN(nn.Module):
    """
    Encoder RNN for sequence-to-sequence models using a bidirectional LSTM.

    Args:
        vocab_size (int): Size of the vocabulary.
        hidden_size (int): Dimensionality of the hidden state.
        n_layers (int, optional): Number of LSTM layers. Default is 1.
    """
    def __init__(self, vocab_size, hidden_size, n_layers=1):
        super(EncoderRNN, self).__init__()
        self.vocab_size = vocab_size
```

```
        self.hidden_size = hidden_size
        self.n_layers = n_layers

        self.embedding = nn.Embedding(vocab_size, hidden_size)
        init.normal_(self.embedding.weight, 0.0, 0.2)

        # Because we use a bidirectional LSTM, the hidden state output will be doubled.
        # Therefore, we reduce the LSTM's output dimensionality to half.
        self.lstm = nn.LSTM(
            input_size=hidden_size,
            hidden_size=hidden_size // 2,
            num_layers=n_layers,
            batch_first=True,
            bidirectional=True
        )

    def forward(self, word_inputs, hidden):
        """
        Forward pass through the encoder.

        Args:
            word_inputs (Tensor): Tensor of shape (batch_size, seq_length) containing word indices.
            hidden (tuple): Tuple of (h_0, c_0) initial hidden states.

        Returns:
            output (Tensor): Output tensor of shape (batch_size, seq_length, hidden_size).
            hidden (tuple): Tuple of hidden states after processing.
        """
        embedded = self.embedding(word_inputs)
        output, hidden = self.lstm(embedded, hidden)
        return output, hidden

    def init_hidden(self, batch_size, device):
        """
        Initialize the hidden and cell states with zeros.

        Args:
            batch_size (int): Batch size.
            device (torch.device): The device to initialize the hidden states on.

        Returns:
            tuple: (h_0, c_0) each of shape (n_layers * 2, batch_size, hidden_size // 2).
        """
        h_0 = torch.zeros(self.n_layers * 2, batch_size, self.hidden_size // 2, device=device)
        c_0 = torch.zeros(self.n_layers * 2, batch_size, self.hidden_size // 2, device=device)
        return (h_0, c_0)
```

24.15. Decoder

```
class DecoderRNN(nn.Module):
    """
    Decoder RNN for sequence-to-sequence models using an LSTM.

    Args:
        vocab_size (int): Size of the vocabulary.
        hidden_size (int): Dimensionality of the hidden state (should match the encoder output).
        n_layers (int, optional): Number of LSTM layers. Default is 1.
    """
    def __init__(self, vocab_size, hidden_size, n_layers=1):
```

```
        super(DecoderRNN, self).__init__()
        self.vocab_size = vocab_size
        self.hidden_size = hidden_size
        self.n_layers = n_layers

        self.embedding = nn.Embedding(vocab_size, hidden_size)
        init.normal_(self.embedding.weight, 0.0, 0.2)

        self.lstm = nn.LSTM(
            input_size=hidden_size,
            hidden_size=hidden_size,
            num_layers=n_layers,
            batch_first=True,
            bidirectional=False
        )

    def forward(self, word_inputs, hidden):
        """
        Forward pass through the decoder for one time step.

        Args:
            word_inputs (Tensor): Tensor of shape (batch_size,) containing word indices.
            hidden (tuple): Tuple of (h_0, c_0) hidden states.

        Returns:
            output (Tensor): Output tensor of shape (batch_size, 1, hidden_size).
            hidden (tuple): Updated hidden states.
        """
        # Unsqueeze to add the time dimension: (batch_size, 1)
        embedded = self.embedding(word_inputs).unsqueeze(1)
        output, hidden = self.lstm(embedded, hidden)
        return output, hidden
```

Testing our Model with Fake Data

```
vocab_size = 10
hidden_dim = 10
n_layers = 2

encoder_test = EncoderRNN(vocab_size, hidden_dim, n_layers).to(device)
print(encoder_test)

batch_size = 1
encoder_hidden = encoder_test.init_hidden(batch_size, device)

test_seq_length = 3
word_input = torch.LongTensor([[1, 2, 3]]).to(device)

# Run the forward pass
encoder_outputs, encoder_hidden = encoder_test(word_input, encoder_hidden)

print(f"Encoder Outputs Shape: {encoder_outputs.shape}  (Expected: [1, {test_seq_length}, {hidden_dim}])")
print(f"Hidden State (h) Shape: {encoder_hidden[0].shape}  (Expected: [{n_layers*2}, 1, {hidden_dim//2}])")
print(f"Cell State (c) Shape: {encoder_hidden[1].shape}  (Expected: [{n_layers*2}, 1, {hidden_dim//2}])")

assert encoder_outputs.shape == (batch_size, test_seq_length, hidden_dim), "Encoder output shape mismatch!"
assert encoder_hidden[0].shape == (n_layers*2, batch_size, hidden_dim//2), "Hidden state shape mismatch!"
assert encoder_hidden[1].shape == (n_layers*2, batch_size, hidden_dim//2), "Cell state shape mismatch!"
```

```
EncoderRNN(
  (embedding): Embedding(10, 10)
  (lstm): LSTM(10, 5, num_layers=2, batch_first=True, bidirectional=True)
)
Encoder Outputs Shape: torch.Size([1, 3, 10])  (Expected: [1, 3, 10])
Hidden State (h) Shape: torch.Size([4, 1, 5])  (Expected: [4, 1, 5])
Cell State (c) Shape: torch.Size([4, 1, 5])  (Expected: [4, 1, 5])
```

```
decoder_test = DecoderRNN(vocab_size, hidden_dim, n_layers).to(device)
print(decoder_test)

decoder_hidden_h = encoder_hidden[0].view(n_layers, 2, batch_size, hidden_dim // 2).permute(0, 2, 1, 3).
  reshape(n_layers, batch_size, hidden_dim)
decoder_hidden_c = encoder_hidden[1].view(n_layers, 2, batch_size, hidden_dim // 2).permute(0, 2, 1, 3).
  reshape(n_layers, batch_size, hidden_dim)

for i in range(test_seq_length):
    input_word = word_input[:, i]
    decoder_output, decoder_hidden = decoder_test(input_word, (decoder_hidden_h, decoder_hidden_c))
    decoder_hidden_h, decoder_hidden_c = decoder_hidden

    print(f"Step {i+1}")
    print(f"Decoder Output Shape: {decoder_output.shape}  (Expected: [1, 1, {hidden_dim}])")
    print(f"Hidden State Shape: {decoder_hidden_h.shape}  (Expected: [{n_layers}, 1, {hidden_dim}])")
    print(f"Cell State Shape: {decoder_hidden_c.shape}  (Expected: [{n_layers}, 1, {hidden_dim}])\n")

assert decoder_output.shape == (batch_size, 1, hidden_dim), "Decoder output shape mismatch!"
assert decoder_hidden_h.shape == (n_layers, batch_size, hidden_dim), "Decoder hidden state shape mismatch!"
assert decoder_hidden_c.shape == (n_layers, batch_size, hidden_dim), "Decoder cell state shape mismatch!"
```

```
DecoderRNN(
  (embedding): Embedding(10, 10)
  (lstm): LSTM(10, 10, num_layers=2, batch_first=True)
)
Step 1
Decoder Output Shape: torch.Size([1, 1, 10])  (Expected: [1, 1, 10])
Hidden State Shape: torch.Size([2, 1, 10])  (Expected: [2, 1, 10])
Cell State Shape: torch.Size([2, 1, 10])  (Expected: [2, 1, 10])

Step 2
Decoder Output Shape: torch.Size([1, 1, 10])  (Expected: [1, 1, 10])
Hidden State Shape: torch.Size([2, 1, 10])  (Expected: [2, 1, 10])
Cell State Shape: torch.Size([2, 1, 10])  (Expected: [2, 1, 10])

Step 3
Decoder Output Shape: torch.Size([1, 1, 10])  (Expected: [1, 1, 10])
Hidden State Shape: torch.Size([2, 1, 10])  (Expected: [2, 1, 10])
Cell State Shape: torch.Size([2, 1, 10])  (Expected: [2, 1, 10])
```

24.16. Seq2Seq

The **Seq2Seq** module manages the interaction between the **encoder** and **decoder**, ensuring that the decoder's outputs are transformed into a probability distribution over the vocabulary.

It implements two forward functions:

- **Training mode** uses **Teacher Forcing**, where the decoder receives the correct target words at each step

to speed up learning.

- **Inference mode** generates words step-by-step, using the model's own predictions as input, since ground-truth targets are unavailable.

```
class TransformerSeq2Seq(nn.Module):
    def __init__(self, input_vocab_size, output_vocab_size, hidden_size=512, n_layers=6, n_heads=8,
                 ff_dim=2048, dropout=0.1, max_len=100, device="cuda"):
        super(TransformerSeq2Seq, self).__init__()

        self.device = device
        self.hidden_size = hidden_size

        self.src_embedding = nn.Embedding(input_vocab_size, hidden_size)
        self.tgt_embedding = nn.Embedding(output_vocab_size, hidden_size)

        self.positional_encoding = PositionalEncoding(hidden_size, max_len, dropout)

        self.transformer = nn.Transformer(
            d_model=hidden_size,
            nhead=n_heads,
            num_encoder_layers=n_layers,
            num_decoder_layers=n_layers,
            dim_feedforward=ff_dim,
            dropout=dropout,
            batch_first=True
        )

        self.fc_out = nn.Linear(hidden_size, output_vocab_size)

    def _causal_mask(self, T, device):
        return torch.triu(torch.ones(T, T, device=device, dtype=torch.bool), diagonal=1)

    def forward(self, src, tgt):
        """
        Forward pass for training.
        Args:
            src: Source sequence tensor of shape (batch_size, src_len)
            tgt: Target sequence tensor of shape (batch_size, tgt_len)
        Returns:
            Output logits of shape (batch_size, tgt_len, output_vocab_size)
        """
        src = self.src_embedding(src) * math.sqrt(self.hidden_size)
        src = self.positional_encoding(src)

        tgt = self.tgt_embedding(tgt) * math.sqrt(self.hidden_size)
        tgt = self.positional_encoding(tgt)

        T = tgt.size(1)
        tgt_mask = self._causal_mask(T, tgt.device)

        out = self.transformer(src, tgt, tgt_mask=tgt_mask)
        return self.fc_out(out)

    def forward_with_masks(self, src, tgt_inp,
                           tgt_mask=None,
                           src_key_padding_mask=None,
                           tgt_key_padding_mask=None,
                           memory_key_padding_mask=None):
        # encoder
```

```
        src = self.src_embedding(src) * math.sqrt(self.hidden_size)
        src = self.positional_encoding(src)
        memory = self.transformer.encoder(
            src, src_key_padding_mask=src_key_padding_mask
        )
        # decoder
        tgt = self.tgt_embedding(tgt_inp) * math.sqrt(self.hidden_size)
        tgt = self.positional_encoding(tgt)
        out = self.transformer.decoder(
            tgt, memory,
            tgt_mask=tgt_mask,
            memory_key_padding_mask=memory_key_padding_mask,
            tgt_key_padding_mask=tgt_key_padding_mask,
        )
        return self.fc_out(out)

    def encode(self, src, src_key_padding_mask=None):
        src = self.src_embedding(src) * math.sqrt(self.hidden_size)
        src = self.positional_encoding(src)
        return self.transformer.encoder(
            src, src_key_padding_mask=src_key_padding_mask
        )

    def decode(self, memory, tgt, memory_key_padding_mask=None, tgt_key_padding_mask=None):
        tgt = self.tgt_embedding(tgt) * math.sqrt(self.hidden_size)
        tgt = self.positional_encoding(tgt)
        T = tgt.size(1)
        tgt_mask = torch.triu(torch.ones(T, T, device=tgt.device, dtype=torch.bool), 1)
        out = self.transformer.decoder(
            tgt, memory,
            tgt_mask=tgt_mask,
            memory_key_padding_mask=memory_key_padding_mask,
            tgt_key_padding_mask=tgt_key_padding_mask,
        )
        return self.fc_out(out)

    def greedy_decode(self, src, max_len=None, sos_token=1, eos_token=2, pad_token_src=0):
        src = src.to(self.device)
        src_kpm = (src == pad_token_src)
        memory  = self.encode(src, src_key_padding_mask=src_kpm)

        if max_len is None:
            max_len = self.positional_encoding.pe.size(1) - 1

        B = src.size(0)
        decoded  = torch.full((B, 1), sos_token, dtype=torch.long, device=self.device)
        finished = torch.zeros(B, dtype=torch.bool, device=self.device)

        for _ in range(max_len):
            logits = self.decode(memory, decoded, memory_key_padding_mask=src_kpm)[:, -1, :]
            next_t = logits.argmax(dim=-1)
            next_t = next_t.masked_fill(finished, eos_token)
            decoded = torch.cat([decoded, next_t.unsqueeze(1)], dim=1)
            finished |= (next_t == eos_token)
            if finished.all():
                break
        return decoded.tolist()

    def beam_search_decode(self, src, beam_size=4, max_len=32, sos_token=1, eos_token=2, pad_token_src=0,
alpha=0.6):
```

```
        device = src.device
        B = src.size(0)
        src_kpm = (src == pad_token_src)

        with torch.inference_mode(), torch.autocast(device_type=device.type, dtype=torch.bfloat16 if device.
type=="cuda" else torch.float32):
            memory = self.encode(src, src_key_padding_mask=src_kpm)                    # [B,S,H]
            mem_beam = memory.unsqueeze(1).expand(B, beam_size, *memory.shape[1:])     # view
            mask_beam = src_kpm.unsqueeze(1).expand(B, beam_size, *src_kpm.shape[1:])

            start = torch.full((B, 1), sos_token, dtype=torch.long, device=device)
            logits0 = self.decode(memory, start, memory_key_padding_mask=src_kpm)[:, -1, :]
            logp0 = F.log_softmax(logits0, dim=-1)
            topk_logp, topk_ids = logp0.topk(beam_size, dim=-1)                        # [B,K]

            seqs = torch.cat([start.unsqueeze(1).expand(B, beam_size, 1), topk_ids.unsqueeze(-1)], dim=-1)  #
[B,K,2]
            scores = topk_logp.clone()                                                 # [B,K]
            finished = topk_ids.eq(eos_token)                                          # [B,K]

            for t in range(2, max_len + 1):
                dec_in = seqs.reshape(B * beam_size, -1)
                logits = self.decode(
                    mem_beam.reshape(B * beam_size, *memory.shape[1:]),
                    dec_in,
                    memory_key_padding_mask=mask_beam.reshape(B * beam_size, *src_kpm.shape[1:])
                )[:, -1, :].reshape(B, beam_size, -1)                                  # [B,K,V]

                # lock finished beams
                logits = logits.masked_fill(finished.unsqueeze(-1), float("-inf"))
                logits[..., eos_token] = torch.where(finished, torch.zeros_like(scores), logits[...,
eos_token])

                logp = F.log_softmax(logits, dim=-1)
                total = scores.unsqueeze(-1) + logp                                     # [B,K,V]
                total_flat = total.reshape(B, -1)                                       # [B,K*V]
                top_total, top_idx = total_flat.topk(beam_size, dim=-1)                 # [B,K]
                parent = top_idx // logp.size(-1)                                       # [B,K]
                token  = top_idx %  logp.size(-1)                                       # [B,K]

                # gather beams without re-allocating memory
                seqs = torch.gather(seqs, 1, parent.unsqueeze(-1).expand(B, beam_size, seqs.size(-1)))
                seqs = torch.cat([seqs, token.unsqueeze(-1)], dim=-1)

                # gather finished and scores
                finished = torch.gather(finished, 1, parent) | token.eq(eos_token)

                # length penalty
                lp = ((5.0 + t) / 6.0) ** alpha
                scores = top_total / lp

                if finished.all(): break

            best = scores.argmax(dim=1)
            best_seq = seqs[torch.arange(B, device=device), best]
            return best_seq[0].tolist() if B == 1 else [best_seq[i].tolist() for i in range(B)]
```

24.17. Training

We initialize the **Seq2Seq model** with a hidden size of **300** and **2 LSTM layers**.

For optimization, we use **AdamW**, a variant of Adam that improves weight decay handling, along with **CrossEntropyLoss**, which ignores " tokens to prevent them from affecting training. A **learning rate scheduler (ReduceLROnPlateau)** is applied to adjust the learning rate dynamically when validation loss stops improving.

```
from torch.optim import lr_scheduler

class NoamScheduler(lr_scheduler._LRScheduler):
    """
    Noam LR: factor * d_model^{-0.5} * min(step^{-0.5}, step * warmup^{-1.5})

    Args:
        optimizer: torch.optim.Optimizer
        d_model: int
        warmup_steps: int
        factor: float overall multiplier
        scale_base_lr: bool. If True, multiply each base_lr by schedule. If False, ignore base_lr and use
   absolute LR.
        min_lr: float clamp
        max_lr: float clamp
        last_epoch: int PyTorch counter; leave -1 normally
    """
    def __init__(self, optimizer, d_model, warmup_steps=8000, factor=1.0,
                 scale_base_lr=False, min_lr=0.0, max_lr=float("inf"), last_epoch=-1):
        self.d_model = float(d_model)
        self.warmup_steps = int(warmup_steps)
        self.factor = float(factor)
        self.scale_base_lr = bool(scale_base_lr)
        self.min_lr = float(min_lr)
        self.max_lr = float(max_lr)
        super().__init__(optimizer, last_epoch=last_epoch)

    def get_lr(self):
        # PyTorch increments last_epoch before calling get_lr()
        step = max(1, self.last_epoch + 1)
        scale = (self.d_model ** -0.5) * min(step ** -0.5,
                                             step * (self.warmup_steps ** -1.5))
        val = self.factor * scale
        if self.scale_base_lr:
            lrs = [self._clamp(base_lr * val) for base_lr in self.base_lrs]
        else:
            lrs = [self._clamp(val) for _ in self.base_lrs]
        return lrs

    def _clamp(self, x):
        return max(self.min_lr, min(self.max_lr, x))
```

```
import torch
import torch.nn.functional as F

# IDs
SRC_PAD = source_vocab.word2index['<pad>']
TGT_PAD = target_vocab.word2index['<pad>']
```

```python
def causal_mask(T, device):
    return torch.triu(torch.ones(T, T, device=device, dtype=torch.bool), 1)

@torch.no_grad()
def test_no_future_leak(model, src, tgt, tol=1e-4):
    """Decoder must not see future tokens."""
    model.eval()
    src = src.to(model.device); tgt = tgt.to(model.device)

    y_in  = tgt[:, :-1]                        # [B, L-1]
    y_in2 = torch.cat([y_in, y_in[:, -1:].clone()], dim=1)  # [B, L]

    src_kpm = (src == SRC_PAD)
    tmask1  = causal_mask(y_in.size(1),  model.device)
    tmask2  = causal_mask(y_in2.size(1), model.device)
    tgt_kpm1= (y_in  == TGT_PAD)
    tgt_kpm2= (y_in2 == TGT_PAD)

    out1 = model.forward_with_masks(
        src, y_in,
        tgt_mask=tmask1,
        src_key_padding_mask=src_kpm,
        tgt_key_padding_mask=tgt_kpm1,
        memory_key_padding_mask=src_kpm,
    )                                   # [B, L-1, V]

    out2 = model.forward_with_masks(
        src, y_in2,
        tgt_mask=tmask2,
        src_key_padding_mask=src_kpm,
        tgt_key_padding_mask=tgt_kpm2,
        memory_key_padding_mask=src_kpm,
    )[:, :y_in.size(1), :]              # align first L-1 steps

    diff = (out1 - out2).abs().max().item()
    print("future-leak max diff:", diff)
    assert diff < tol, "Causal mask failure."

@torch.no_grad()
def test_src_padding_mask_encoder_and_decoder(model, src, tgt, tol=1e-4):
    """Pads in src must be ignored by encoder and by decoder cross-attn."""
    model.eval()
    src = src.to(model.device); tgt = tgt.to(model.device)
    half = src.size(1) // 2

    src_trunc = src[:, :half]
    src_pad   = torch.cat([src_trunc, torch.full_like(src[:, half:], SRC_PAD)], dim=1)

    # encoder invariance on the real region
    mem1 = model.encode(src_trunc, src_key_padding_mask=(src_trunc == SRC_PAD))
    mem2 = model.encode(src_pad,   src_key_padding_mask=(src_pad   == SRC_PAD))
    enc_diff = (mem1 - mem2[:, :mem1.size(1)]).abs().max().item()
    print("encoder src_kpm diff:", enc_diff)
    assert enc_diff < tol, "Encoder src_key_padding_mask not honored."

    # decoder invariance when using memory_key_padding_mask
    y_in = tgt[:, :-1]
    tgt_mask = causal_mask(y_in.size(1), model.device)
    tgt_kpm  = (y_in == TGT_PAD)
```

```
    out1 = model.forward_with_masks(
        src_trunc, y_in,
        tgt_mask=tgt_mask,
        src_key_padding_mask=(src_trunc == SRC_PAD),
        tgt_key_padding_mask=tgt_kpm,
        memory_key_padding_mask=(src_trunc == SRC_PAD),
    )
    out2 = model.forward_with_masks(
        src_pad, y_in,
        tgt_mask=tgt_mask,
        src_key_padding_mask=(src_pad == SRC_PAD),
        tgt_key_padding_mask=tgt_kpm,
        memory_key_padding_mask=(src_pad == SRC_PAD),
    )[:, :, :]  # same length

    dec_diff = (out1 - out2).abs().max().item()
    print("decoder mem_kpm diff:", dec_diff)
    assert dec_diff < tol, "Decoder memory_key_padding_mask not honored."

@torch.no_grad()
def test_tgt_padding_mask(model, src, tgt, tol=1e-4):
    """Extra PADs in y_in must not affect earlier steps."""
    model.eval()
    src = src.to(model.device); tgt = tgt.to(model.device)

    y_in = tgt[:, :-1]
    # append PADs
    pad_len = max(1, y_in.size(1) // 4)
    y_in_pad = F.pad(y_in, (0, pad_len), value=TGT_PAD)

    src_kpm = (src == SRC_PAD)
    tmask1  = causal_mask(y_in.size(1),      model.device)
    tmask2  = causal_mask(y_in_pad.size(1),  model.device)
    tgt_kpm1= (y_in     == TGT_PAD)
    tgt_kpm2= (y_in_pad == TGT_PAD)

    out1 = model.forward_with_masks(
        src, y_in,
        tgt_mask=tmask1,
        src_key_padding_mask=src_kpm,
        tgt_key_padding_mask=tgt_kpm1,
        memory_key_padding_mask=src_kpm,
    )                                   # [B, L-1, V]

    out2 = model.forward_with_masks(
        src, y_in_pad,
        tgt_mask=tmask2,
        src_key_padding_mask=src_kpm,
        tgt_key_padding_mask=tgt_kpm2,
        memory_key_padding_mask=src_kpm,
    )[:, :out1.size(1), :]              # compare valid prefix

    diff = (out1 - out2).abs().max().item()
    print("tgt pad-mask diff:", diff)
    assert diff < tol, "tgt_key_padding_mask not honored."

# ---- Run on a small training batch ----
x_batch, y_batch = next(iter(train_loader))
x_batch = x_batch[:16]; y_batch = y_batch[:16]
```

```
test_no_future_leak(model, x_batch, y_batch)
test_src_padding_mask_encoder_and_decoder(model, x_batch, y_batch)
test_tgt_padding_mask(model, x_batch, y_batch)
print("All mask tests passed.")
```

```
/usr/local/lib/python3.12/dist-packages/torch/nn/modules/transformer.py:531:
UserWarning: The PyTorch API of nested tensors is in prototype stage and will
change in the near future ... [259 chars truncated]
  output = torch._nested_tensor_from_mask(
```

```
future-leak max diff: 0.0
encoder src_kpm diff: 0.0
decoder mem_kpm diff: 3.2186508178710938e-06
tgt pad-mask diff: 9.506940841674805e-06
All mask tests passed.
```

Performance Metrics

24.18. Main Training Loop

```
SRC_PAD = source_vocab.word2index['<pad>']
TGT_PAD = target_vocab.word2index['<pad>']
SOS = target_vocab.word2index['<sos>']
EOS = target_vocab.word2index['<eos>']

def causal_mask(T, device):
    return torch.triu(torch.ones(T, T, device=device, dtype=torch.bool), 1)

num_epochs = 15
early_stop_after = 5
early_stop_counter = 0
best_score = 0.0
scores_history, loss_history = [], []

for epoch in range(num_epochs):
    model.train()
    total_loss = 0.0

    for x_batch, y_batch in tqdm(train_loader, desc=f"Epoch {epoch+1}/{num_epochs} Training", leave=False):
        x_batch = x_batch.to(device)      # [B,S]
        y_batch = y_batch.to(device)      # [B,T]  <sos> ... <eos> <pad>...

        y_in  = y_batch[:, :-1]           # decoder inputs
        y_out = y_batch[:,  1:]           # next-token targets

        src_kpm = (x_batch == SRC_PAD)           # [B,S] bool
        tgt_kpm = (y_in    == TGT_PAD)           # [B,T-1] bool
        tmask   = causal_mask(y_in.size(1), device)

        logits = model.forward_with_masks(
            x_batch, y_in,
            tgt_mask=tmask,
            src_key_padding_mask=src_kpm,
            tgt_key_padding_mask=tgt_kpm,
            memory_key_padding_mask=src_kpm,
        )                                        # [B,T-1,V]

        loss = criterion(logits.reshape(-1, logits.size(-1)),
                         y_out.reshape(-1))
```

```
        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

        optimizer.step()
        scheduler.step()

        total_loss += loss.item()

    avg_loss = total_loss / len(train_loader)
    loss_history.append(avg_loss)
    current_lr = optimizer.param_groups[0]['lr']
    print(f"Epoch {epoch+1}/{num_epochs} - Training Loss: {avg_loss:.4f} | lr={current_lr:.6e}")

    # Validation
    model.eval()
    val_scores = score(model, valid_loader, target_vocab, device=device,
                       desc=f"Validation Epoch {epoch+1}", src_pad_id=SRC_PAD)  # <-- pass src PAD
    scores_history.append(val_scores)

    print(f"Epoch {epoch+1} Validation Scores:")
    for name, sc in val_scores.items():
        print(f"{name}: {sc:.4f}")

    metric = val_scores['BLEU-4']
    if metric > best_score:
        best_score = metric
        best_model_wts = copy.deepcopy(model.state_dict())
        early_stop_counter = 0
        print("New best model.")
    else:
        early_stop_counter += 1
        print(f"No improvement for {early_stop_counter} epochs.")
        if early_stop_counter >= early_stop_after:
            print("Early stopping triggered.")
            break
```

```
Epoch 1/15 - Training Loss: 6.5634 | lr=2.364297e-05
```

```
Validation Epoch 1: 100%|##########| 118/118 [00:39<00:00,  3.00it/s]
```

```
Epoch 1 Validation Scores:
BLEU-1: 0.3389
BLEU-2: 0.1030
BLEU-3: 0.0260
BLEU-4: 0.0065
Accuracy: 0.0000
New best model.
```

```
Epoch 2/15 Training:  56%|######    | 1062/1913 [03:03<02:26,  5.82it/s]
```

24.19. Plotting our Results

```
def plot_training_progress(loss_history, scores_history, score_functions, epoch):
    """
    Plots training loss and evaluation metrics (BLEU, Accuracy) over epochs.
```

```
    Args:
        loss_history (list): List of training loss values per epoch.
        scores_history (list of dicts): List of evaluation scores per epoch (each dict contains score names as
    keys).
        score_functions (dict): Dictionary containing scoring function names.
        epoch (int): The last completed epoch.
    """
    num_metrics = len(score_functions)
    total_plots = num_metrics + 1
    ncols = 2
    nrows = int(np.ceil(total_plots / ncols))

    fig, axes = plt.subplots(nrows, ncols, figsize=(14, 4 * nrows), dpi=80)
    axes = axes.flatten()
    epochs = list(range(1, epoch + 2))

    epochs_ticks = list(range(1, epoch + 2, max(1, epoch // 10)))

    # ---- Plot Loss History ----
    axes[0].plot(epochs, loss_history, color='dodgerblue', linewidth=2)
    axes[0].set_title("Training Loss Over Epochs", fontsize=14, fontweight='bold')
    axes[0].set_xlabel("Epochs")
    axes[0].set_xticks(epochs_ticks)
    axes[0].set_ylabel("Loss")
    axes[0].grid(True)

    # ---- Plot BLEU & Accuracy Scores ----
    for idx, name in enumerate(score_functions.keys(), start=1):
        score_history = [s[name] for s in scores_history]

        axes[idx].plot(epochs, score_history, color='darkorange', linewidth=2)
        axes[idx].set_title(f"Validation Score: {name}", fontsize=14, fontweight='bold')
        axes[idx].set_xlabel("Epochs")
        axes[idx].set_xticks(epochs_ticks)
        axes[idx].set_ylabel("Score")
        axes[idx].grid(True)

    # Hide unused subplots (if any)
    for idx in range(total_plots, len(axes)):
        fig.delaxes(axes[idx])

    plt.tight_layout()
    plt.show()
```

```
plot_training_progress(loss_history, scores_history, score_functions, epoch)
```

24.20. Testing and Evaluating our Model

```
@torch.inference_mode()
def run_test(model, test_loader, target_vocab, src_pad_id, sos_id, eos_id, beam_size=4, max_len=32):
    hyps, refs = [], []
    if max_len is None:
        max_len = model.positional_encoding.pe.size(1) - 1

    for xb, yb in tqdm(test_loader, desc="Test"):
        xb = xb.to(model.device, non_blocking=True)
        preds = model.beam_search_decode(xb, beam_size=beam_size, max_len=max_len,
```

```
                                        sos_token=sos_id, eos_token=eos_id, pad_token_src=src_pad_id)
        # strip specials
        for p, refset in zip(preds, yb):
            hyp = [target_vocab.index2word.get(int(t), "<unk>") for t in p
                   if target_vocab.index2word.get(int(t), "<unk>") not in {"<sos>","<eos>","<pad>"}]
            ref_clean = []
            for r in refset:
                r = r.tolist() if torch.is_tensor(r) else r
                ref_clean.append([target_vocab.index2word.get(int(t), "<unk>") for t in r
                                  if target_vocab.index2word.get(int(t), "<unk>") not in
  {"<sos>","<eos>","<pad>"}])
            hyps.append(hyp)
            refs.append(ref_clean)
    return hyps, refs
```

```
save_path = "best_model.pth"
print(f"Saving best model to {save_path}...")
torch.save(best_model_wts, save_path)
```

```
test_scores = run_test(model, test_loader, target_vocab, src_pad_id=SRC_PAD, sos_id=SOS_idx, eos_id=EOS_idx,
  max_len=32)
```

```
import random
from nltk.translate.bleu_score import sentence_bleu, SmoothingFunction
import numpy as np

def compare_test_outputs(hyps, refs, n_samples=10, seed=0):
    """
    Compare model outputs vs references with sample inspection and simple stats.

    Args:
        hyps (list[list[str]]): predicted token sequences.
        refs (list[list[list[str]]]): list of lists of reference token lists.
        n_samples (int): number of random examples to print.
        seed (int): random seed for reproducibility.
    """
    assert len(hyps) == len(refs), f"Size mismatch: {len(hyps)} hyps vs {len(refs)} refs"

    smoothing = SmoothingFunction().method1

    lens_pred = [len(h) for h in hyps]
    lens_ref  = [np.mean([len(r) for r in rs]) for rs in refs]
    avg_bleu1 = []
    token_match = 0
    token_total = 0

    for h, rs in zip(hyps, refs):
        bleu1 = sentence_bleu(rs, h, weights=(1,0,0,0), smoothing_function=smoothing)
        avg_bleu1.append(bleu1)
        ref_best = max(rs, key=len)
        # token overlap
        token_match += sum(t in ref_best for t in h)
        token_total += len(h)

    print(f"Total samples: {len(hyps)}")
    print(f"Avg pred length: {np.mean(lens_pred):.2f}, avg ref length: {np.mean(lens_ref):.2f}")
    print(f"Token overlap ratio: {token_match/token_total:.3f}")
    print(f"Avg sentence BLEU-1: {np.mean(avg_bleu1):.3f}")
```

```
    print("\n--- Sample predictions ---")
    random.seed(seed)
    for i in random.sample(range(len(hyps)), min(n_samples, len(hyps))):
        h = " ".join(hyps[i])
        rs = [" ".join(r) for r in refs[i]]
        bleu4 = sentence_bleu(refs[i], hyps[i], weights=(0.25,0.25,0.25,0.25),
                              smoothing_function=smoothing)
        print(f"[{i}] BLEU-4={bleu4:.3f}")
        print(f"  Hyp: {h}")
        print(f"  Ref: {rs}")
        print("-"*60)
```

```
hyps, refs = test_scores
compare_test_outputs(hyps, refs, n_samples=5)
```

24.21. Homework (Optional)

1. **RNN vs. transformer on the same data.** Train both the RNN seq2seq from the previous chapter and this transformer on the same English/French subset for 15 epochs each. Report validation BLEU, total parameters, and wall-clock time per epoch.
2. **Positional encoding ablation.** Replace the sinusoidal positional encoding with a zero tensor (i.e. remove the positional signal). Report the change in training loss and qualitative output on five test sentences.
3. **Number of heads and layers.** Train variants with `(nhead=2, layers=2)`, `(nhead=4, layers=3)`, and `(nhead=8, layers=6)`. Plot validation loss vs. epoch for all three and discuss the capacity / overfit trade-off.
4. **Label smoothing.** Add `label_smoothing=0.1` to the cross-entropy loss. Report the change in validation loss and translation quality on five held-out examples.
5. **Greedy vs. beam search.** Implement a beam-search decoder with beam sizes 1 (greedy), 4, and 8. Compare BLEU and decoding latency per sentence.
6. **Different language pair.** Retrain the pipeline with a different source / target pair from the Tatoeba links file (e.g. Hungarian -> English). Report BLEU and any language-specific preprocessing needed.

Chapter 25

Hugging Face Transformers and Gradio

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1sgDBap3vU3QiFTWWzRbxpZhlw5Cl9eas?usp=sharing

Hugging Face is a leading platform for natural language processing and machine learning, providing tools such as the `Transformers library`, which offers pre-trained models and efficient implementations of state-of-the-art architectures.

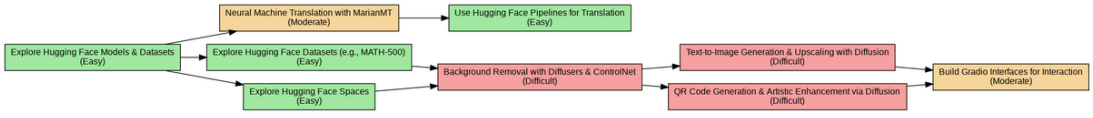
Gradio is an open-source library for creating web-based interfaces to interact with machine learning models. It allows researchers to build interactive demos with minimal code.

25.1. Necessary Imports

```
!pip install --quiet qrcode gradio datasets kornia torchinfo
```

```
import torch
import random
import time
import os
import matplotlib.pyplot as plt
import qrcode
import numpy as np
import gradio as gr
import pandas as pd
import requests
from io import BytesIO
from IPython.display import display
from IPython.display import Image as IMG
from torchinfo import summary
from transformers import MarianMTModel, MarianTokenizer
from transformers import pipeline
from torchvision import transforms
from transformers import AutoModelForImageSegmentation
from datasets import load_dataset
from diffusers import StableDiffusionPipeline, DPMSolverMultistepScheduler, DiffusionPipeline
from diffusers.pipelines.stable_diffusion.safety_checker import StableDiffusionSafetyChecker
from diffusers import ControlNetModel, StableDiffusionControlNetPipeline,
  StableDiffusionControlNetImg2ImgPipeline
from PIL import Image

device = "cuda" if torch.cuda.is_available() else "cpu"
```



25.2. The Hugging Face ecosystem

Hugging Face ships three interlocking products. Understanding which one you are using at any moment makes the API choices obvious.

The Hub hosts pretrained models. Every model has a unique identifier of the form `organization/model-name` (for example `Helsinki-NLP /opus-mt-en-fr`). Calling `AutoModel.from_pretrained(name)` downloads the weights and config, caches them under `~/.cache/huggingface/`, and hands back an instantiated model. The same pattern works for tokenizers, feature extractors, processors, and pipelines.

Datasets is a streaming-friendly library for loading corpora by name. `load_dataset('HuggingFaceH4/MATH-500')` fetches a `DatasetDict` with the standard train/validation/test splits and a consistent column-oriented API. Map, filter, and shuffle behave the same on every dataset. The library can stream from the Hub without downloading everything, which matters once datasets grow past a few hundred gigabytes.

Spaces are hosted demos, typically Gradio or Streamlit apps. You write a small Python file with an `Interface` or `Blocks` layout, push it to a Space repo, and it runs on Hugging Face's infrastructure with a public URL. The same Gradio code runs locally and on a Space, so development iterates in a notebook and demos are one commit away.

The three products are designed to compose: a Space loads a Model from the Hub, fine-tunes it on a Dataset, and exposes it through a Gradio interface. The rest of this chapter works through that composition end-to-end.

25.3. Exploring the Hugging Face Website

The **Hugging Face** model page lists all the trending models uploaded to the platform, including state-of-the-art models for tasks such as text generation, translation, sentiment analysis, and image processing. These models span various architectures, such as Transformers, BERT [18] , GPT, and Stable Diffusion [23] , and can be easily integrated into research and production workflows using the Hugging Face Transformers library.

25.4. Exploring a Model

Each model includes a model card, which serves as a concise documentation, typically detailing the model's architecture, training data, intended use cases, and limitations. Some models also feature an `Inference API`, allowing users to test them directly on the Hugging Face platform. However, most models require integration with the `Transformers library` for local experimentation and deployment.

DeepSeek-R1

Helsinki NLP (NMT)

The `Helsinki-NLP/opus-mt-en-fr` model is a **neural machine translation model** designed to translate text from English to French. Developed by the Language Technology Research Group at the University of Helsinki, it is part of the OPUS-MT [22] project, which aims to provide open translation services for

various language pairs. The model is based on the transformer architecture and was trained on the **OPUS dataset**, utilizing preprocessing steps like normalization and SentencePiece tokenization.

```
model_name = "Helsinki-NLP/opus-mt-en-fr"
tokenizer = MarianTokenizer.from_pretrained(model_name)
model = MarianMTModel.from_pretrained(model_name)
```

```
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94:
UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab
(https://huggingface.co/settings/tokens), set it as secret in your Google Colab
and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access
public models or datasets.
  warnings.warn(
```

```
tokenizer_config.json:   0%|          | 0.00/42.0 [00:00<?, ?B/s]
```

```
source.spm:   0%|          | 0.00/778k [00:00<?, ?B/s]
```

```
target.spm:   0%|          | 0.00/802k [00:00<?, ?B/s]
```

```
vocab.json:   0%|          | 0.00/1.34M [00:00<?, ?B/s]
```

```
config.json:   0%|          | 0.00/1.42k [00:00<?, ?B/s]
```

```
/usr/local/lib/python3.11/dist-
packages/transformers/models/marian/tokenization_marian.py:175: UserWarning:
Recommended: pip install sacremoses.
  warnings.warn("Recommended: pip install sacremoses.")
```

```
pytorch_model.bin:   0%|          | 0.00/301M [00:00<?, ?B/s]
```

```
model.safetensors:   0%|          | 0.00/301M [00:00<?, ?B/s]
```

```
generation_config.json:   0%|          | 0.00/293 [00:00<?, ?B/s]
```

```
src_text = ["Hello, how are you?"]
inputs = tokenizer(src_text, return_tensors="pt", padding=True)
translated_tokens = model.generate(**inputs)
translated_text = tokenizer.batch_decode(translated_tokens, skip_special_tokens=True)

print(translated_text)
```

```
['Bonjour, comment allez-vous ?']
```

```
def translate_text(text, model, tokenizer):
  """
  Translates English text to French using the Helsinki-NLP/opus-mt-en-fr model.

  Args:
      text: The English text to be translated.
      model: The NMT model.
      tokenizer: The tokenizer associated with the model.

  Returns:
      The translated French text.
```

```
  """

  inputs = tokenizer(text, return_tensors="pt", padding=True)
  translated_tokens = model.generate(**inputs)
  translated_text = tokenizer.batch_decode(translated_tokens, skip_special_tokens=True)
  return translated_text[0]
```

```
src_text = ["Despite everything, it's still just matrix multiplication"]
translate_text(src_text, model, tokenizer)
```

```
"Malgré tout, c'est encore une multiplication matricielle."
```

```
model   # display value
```

```
MarianMTModel(
  (model): MarianModel(
    (shared): Embedding(59514, 512, padding_idx=59513)
    (encoder): MarianEncoder(
      (embed_tokens): Embedding(59514, 512, padding_idx=59513)
      (embed_positions): MarianSinusoidalPositionalEmbedding(512, 512)
      (layers): ModuleList(
        (0-5): 6 x MarianEncoderLayer(
          (self_attn): MarianAttention(
            (k_proj): Linear(in_features=512, out_features=512, bias=True)
            (v_proj): Linear(in_features=512, out_features=512, bias=True)
            (q_proj): Linear(in_features=512, out_features=512, bias=True)
            (out_proj): Linear(in_features=512, out_features=512, bias=True)
          )
          (self_attn_layer_norm): LayerNorm((512,), eps=1e-05,
elementwise_affine=True)
          (activation_fn): SiLU()
          (fc1): Linear(in_features=512, out_features=2048, bias=True)
          (fc2): Linear(in_features=2048, out_features=512, bias=True)
          (final_layer_norm): LayerNorm((512,), eps=1e-05,
elementwise_affine=True)
        )
[... 21 lines omitted ...]
          (encoder_attn_layer_norm): LayerNorm((512,), eps=1e-05,
elementwise_affine=True)
          (fc1): Linear(in_features=512, out_features=2048, bias=True)
          (fc2): Linear(in_features=2048, out_features=512, bias=True)
          (final_layer_norm): LayerNorm((512,), eps=1e-05,
elementwise_affine=True)
        )
      )
    )
  )
  (lm_head): Linear(in_features=512, out_features=59514, bias=False)
)
```

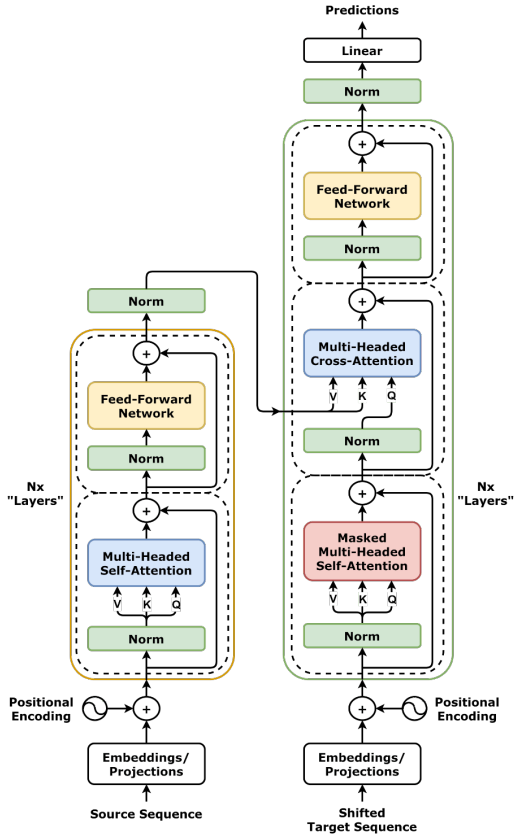


Figure credit: dvgodoy, via Wikimedia Commons, CC BY 4.0.

The **MarianMTModel** adheres to the traditional Transformer architecture as detailed in the *Attention is All You Need* paper. It comprises six encoder and six decoder layers, each following the **original Transformer's design**. The final layer is a linear projection that outputs logits corresponding to the vocabulary's words. Notably, a `Softmax` layer is absent here because it's typically integrated within the loss function during training, especially when using cross-entropy loss.

Additionally, the embedding layers feature a `padding_idx` parameter set to the penultimate index of the vocabulary, rather than zero. This design choice ensures that the padding token is appropriately handled during training and inference. It does not increase the performance of the model in any way.

The `LayerNorm` layer normalizes the inputs across the features of each data sample. The `epsilon` parameter is a small constant added to the denominator to prevent division by zero. When `elementwise_affine` is set to True, the layer learns separate scaling and bias parameters for each feature, allowing for more flexible normalization (extra parameters).

A Different Model

```
model_name = "Helsinki-NLP/opus-mt-en-de"
tokenizer = MarianTokenizer.from_pretrained(model_name)
model = MarianMTModel.from_pretrained(model_name)
```

```
tokenizer_config.json:   0%|          | 0.00/42.0 [00:00<?, ?B/s]
```

```
source.spm:   0%|          | 0.00/768k [00:00<?, ?B/s]
```

```
target.spm:   0%|          | 0.00/797k [00:00<?, ?B/s]
```

```
vocab.json:   0%|          | 0.00/1.27M [00:00<?, ?B/s]
```

```
config.json:   0%|          | 0.00/1.33k [00:00<?, ?B/s]
```

```
pytorch_model.bin:   0%|          | 0.00/298M [00:00<?, ?B/s]
```

```
model.safetensors:   0%|          | 0.00/298M [00:00<?, ?B/s]
```

```
generation_config.json:   0%|          | 0.00/293 [00:00<?, ?B/s]
```

```
src_text = ["Good morning"]
translate_text(src_text, model, tokenizer)
```

```
'Guten Morgen.'
```

```
src_text = ["Despite everything, it's still just matrix multiplication"]
translate_text(src_text, model, tokenizer)
```

```
'Trotz allem, es ist immer noch nur Matrix-Multiplikation'
```

```
model   # display value
```

```
MarianMTModel(
  (model): MarianModel(
    (shared): Embedding(58101, 512, padding_idx=58100)
    (encoder): MarianEncoder(
      (embed_tokens): Embedding(58101, 512, padding_idx=58100)
      (embed_positions): MarianSinusoidalPositionalEmbedding(512, 512)
      (layers): ModuleList(
        (0-5): 6 x MarianEncoderLayer(
          (self_attn): MarianAttention(
            (k_proj): Linear(in_features=512, out_features=512, bias=True)
            (v_proj): Linear(in_features=512, out_features=512, bias=True)
            (q_proj): Linear(in_features=512, out_features=512, bias=True)
            (out_proj): Linear(in_features=512, out_features=512, bias=True)
          )
          (self_attn_layer_norm): LayerNorm((512,), eps=1e-05,
elementwise_affine=True)
          (activation_fn): SiLU()
          (fc1): Linear(in_features=512, out_features=2048, bias=True)
          (fc2): Linear(in_features=2048, out_features=512, bias=True)
          (final_layer_norm): LayerNorm((512,), eps=1e-05,
elementwise_affine=True)
        )
[... 21 lines omitted ...]
          (encoder_attn_layer_norm): LayerNorm((512,), eps=1e-05,
elementwise_affine=True)
          (fc1): Linear(in_features=512, out_features=2048, bias=True)
          (fc2): Linear(in_features=2048, out_features=512, bias=True)
          (final_layer_norm): LayerNorm((512,), eps=1e-05,
elementwise_affine=True)
        )
      )
```

```
    )
  )
  (lm_head): Linear(in_features=512, out_features=58101, bias=False)
)
```

Notice how the german (simplified) vocabulary contains less words compared to the french vocabulary. (58101 to 59514)

```
batch_size = 1
seq_length = 100

model.to(device)

# Input IDs usually come from a tokenizer
dummy_input_ids = torch.randint(0, model.config.vocab_size, (batch_size, seq_length)).to(device)
# Decoder Input IDs are the target sequence (german sequence) to generate when utilizing teacher forcing,
   otherwise a sequence with a single <sos> token.
dummy_decoder_input_ids = torch.randint(0, model.config.vocab_size, (batch_size, seq_length)).to(device)

summary(model, input_data={"input_ids": dummy_input_ids, "decoder_input_ids": dummy_decoder_input_ids})
```

```
==========================================================================================
==============================
Layer (type:depth-idx)                                  Output Shape
Param #
==========================================================================================
==============================
MarianMTModel                                           [1, 100, 512]
--
+-MarianModel: 1-1                                      [1, 100, 512]
--
|    +-MarianEncoder: 2-1                               [1, 100, 512]
48,924,160
|    +-MarianDecoder: 2-2                               --
(recursive)
|    |    +-Embedding: 3-1                              [1, 100, 512]
29,747,712
|    +-MarianEncoder: 2-3                               --
(recursive)
|    |    +-MarianSinusoidalPositionalEmbedding: 3-2    [100, 512]
(262,144)
|    |    +-ModuleList: 3-3                             --
18,914,304
|    +-MarianDecoder: 2-4                               [1, 8, 100, 64]
29,747,712
|    |    +-Embedding: 3-4                              [1, 100, 512]
(recursive)
|    |    +-MarianSinusoidalPositionalEmbedding: 3-5    [100, 512]
(262,144)
|    |    +-ModuleList: 3-6                             --
25,224,192
+-Linear: 1-2                                           [1, 100, 58101]
29,747,712
==========================================================================================
```

```
==============================
Total params: 182,830,080
Trainable params: 182,043,648
Non-trainable params: 786,432
Total mult-adds (Units.MEGABYTES): 185.81
==========================================================================================
==============================
Input size (MB): 0.00
Forward/backward pass size (MB): 114.47
Params size (MB): 416.63
Estimated Total Size (MB): 531.11
==========================================================================================
==============================
```

Pipelines

In the Hugging Face Transformers library, the `pipeline` function provides a high-level API that simplifies the process of using pre-trained models. It abstracts the complexities involved in model loading, tokenization, and inference.

The official Hugging Face `pipeline` API supports 30 different types of tasks, each corresponding to a specific pipeline class. Below is the list of currently accepted tasks:

- `Audio Classification` ("audio-classification"): Returns an AudioClassificationPipeline.
- `Automatic Speech Recognition` ("automatic-speech-recognition"): Returns an AutomaticSpeechRecognitionPipeline.
- `Depth Estimation` ("depth-estimation"): Returns a DepthEstimationPipeline.
- `Document Question Answering` ("document-question-answering"): Returns a DocumentQuestionAnsweringPipeline.
- `Feature Extraction` ("feature-extraction"): Returns a FeatureExtractionPipeline.
- `Fill Mask` ("fill-mask"): Returns a FillMaskPipeline.
- `Image Classification` ("image-classification"): Returns an ImageClassificationPipeline.
- `Image Feature Extraction` ("image-feature-extraction"): Returns an ImageFeatureExtractionPipeline.
- `Image Segmentation` ("image-segmentation"): Returns an ImageSegmentationPipeline.
- `Image Text-to-Text` ("image-text-to-text"): Returns an ImageTextToTextPipeline.
- `Image-to-Image`("image-to-image"): Returns an ImageToImagePipeline.
- `Image-to-Text` ("image-to-text"): Returns an ImageToTextPipeline.
- `Mask Generation` ("mask-generation"): Returns a MaskGenerationPipeline.
- `Object Detection` ("object-detection"): Returns an ObjectDetectionPipeline.
- `Question Answering` ("question-answering"): Returns a QuestionAnsweringPipeline.
- `Summarization` ("summarization"): Returns a SummarizationPipeline.

- `Table Question Answering` ("table-question-answering"): Returns a TableQuestionAnsweringPipeline.
- `Text-to-Text Generation` ("text2text-generation"): Returns a Text2TextGenerationPipeline.
- `Text Classification` ("text-classification", alias "sentiment-analysis" available): Returns a TextClassificationPipeline.
- `Text Generation` ("text-generation"): Returns a TextGenerationPipeline.
- `Text-to-Audio` ("text-to-audio", alias "text-to-speech" available): Returns a TextToAudioPipeline.
- `Token Classification` ("token-classification", alias "ner" available): Returns a TokenClassificationPipeline.
- `Translation` ("translation"): Returns a TranslationPipeline.
- `Translation` (Specific Language Pair) ("translation_xx_to_yy"): Returns a TranslationPipeline.
- `Video Classification` ("video-classification"): Returns a VideoClassificationPipeline.
- `Visual Question Answering` ("visual-question-answering"): Returns a VisualQuestionAnsweringPipeline.
- `Zero-Shot Classification` ("zero-shot-classification"): Returns a ZeroShotClassificationPipeline.
- `Zero-Shot Image Classification` ("zero-shot-image-classification"): Returns a ZeroShotImageClassificationPipeline.
- `Zero-Shot Audio Classification` ("zero-shot-audio-classification"): Returns a ZeroShotAudioClassificationPipeline.
- `Zero-Shot Object Detection` ("zero-shot-object-detection"): Returns a ZeroShotObjectDetectionPipeline.

If no model is specified, the pipeline will default to a pre-defined model for each pipeline class. For "translation_xx_to_yy", the default fallback model is Google's T5 model.

```
translator = pipeline(task = "translation_en_to_de", model = "google-t5/t5-base")
translated_text = translator("This is a test sentence.")
```

```
config.json:   0%|          | 0.00/1.21k [00:00<?, ?B/s]
```

```
model.safetensors:   0%|          | 0.00/892M [00:00<?, ?B/s]
```

```
generation_config.json:   0%|          | 0.00/147 [00:00<?, ?B/s]
```

```
spiece.model:   0%|          | 0.00/792k [00:00<?, ?B/s]
```

```
tokenizer.json:   0%|          | 0.00/1.39M [00:00<?, ?B/s]
```

```
Device set to use cuda:0
```

```
print(translated_text[0])
```

```
{'translation_text': 'Das ist ein Teststrafen.'}
```

```
batch_size = 1
seq_length = 100

dummy_input_ids = torch.randint(0, translator.model.config.vocab_size, (batch_size, seq_length)).to(device)
dummy_decoder_input_ids = torch.randint(0, translator.model.config.vocab_size, (batch_size, seq_length)).
  to(device)
summary(translator.model, input_data={"input_ids": dummy_input_ids, "decoder_input_ids":
  dummy_decoder_input_ids})
```

```
Passing a tuple of `past_key_values` is deprecated and will be removed in
Transformers v4.48.0. You should pass an instance of `EncoderDecoderCache`
instead, e.g. `past_key_values= ... [56 chars truncated]
```

```
==========================================================================================
==========================
Layer (type:depth-idx)                                  Output Shape
Param #
==========================================================================================
==========================
T5ForConditionalGeneration                              [1, 100, 768]
--
+-T5Stack: 1-1                                          [1, 100, 768]
109,628,544
+-T5Stack: 1-2                                          --
(recursive)
|    +-Embedding: 2-1                                   [1, 100, 768]
24,674,304
+-T5Stack: 1-3                                          --
(recursive)
|    +-Dropout: 2-2                                     [1, 100, 768]
--
|    +-ModuleList: 2-3                                  --
--
|    |    +-T5Block: 3-1                                [1, 100, 768]
7,079,808
|    |    +-T5Block: 3-2                                [1, 100, 768]
7,079,424
|    |    +-T5Block: 3-3                                [1, 100, 768]
7,079,424
|    |    +-T5Block: 3-4                                [1, 100, 768]
7,079,424
|    |    +-T5Block: 3-5                                [1, 100, 768]
7,079,424
|    |    +-T5Block: 3-6                                [1, 100, 768]
7,079,424
|    |    +-T5Block: 3-7                                [1, 100, 768]
7,079,424
|    |    +-T5Block: 3-8                                [1, 100, 768]
7,079,424
|    |    +-T5Block: 3-9                                [1, 100, 768]
7,079,424
|    |    +-T5Block: 3-10                               [1, 100, 768]
7,079,424
```

```
[... 24 lines omitted ...]
Total params: 381,880,704
Trainable params: 381,880,704
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 272.33
==========================================================================================
==========================
Input size (MB): 0.00
Forward/backward pass size (MB): 229.15
Params size (MB): 990.31
Estimated Total Size (MB): 1219.46
==========================================================================================
==========================
```

```
print(translated_text[0]['translation_text'])
```

```
Das ist ein Teststrafen.
```

```
translator.model   # display value
```

```
T5ForConditionalGeneration(
  (shared): Embedding(32128, 768)
  (encoder): T5Stack(
    (embed_tokens): Embedding(32128, 768)
    (block): ModuleList(
      (0): T5Block(
        (layer): ModuleList(
          (0): T5LayerSelfAttention(
            (SelfAttention): T5Attention(
              (q): Linear(in_features=768, out_features=768, bias=False)
              (k): Linear(in_features=768, out_features=768, bias=False)
              (v): Linear(in_features=768, out_features=768, bias=False)
              (o): Linear(in_features=768, out_features=768, bias=False)
              (relative_attention_bias): Embedding(32, 12)
            )
            (layer_norm): T5LayerNorm()
            (dropout): Dropout(p=0.1, inplace=False)
          )
          (1): T5LayerFF(
            (DenseReluDense): T5DenseActDense(
[... 106 lines omitted ...]
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (final_layer_norm): T5LayerNorm()
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (lm_head): Linear(in_features=768, out_features=32128, bias=False)
)
```

25.5. Hugging Face Datasets

The **Hugging Face Datasets** library provides a vast collection of datasets for ML and (mostly) NLP tasks. One notable example is the `Common Corpus`, the largest openly licensed text dataset, containing approximately 2 trillion tokens (1,998,647,168,282 tokens). It is a diverse resource, encompassing books, newspapers, scientific articles, government and legal documents, code, and more. The Common Corpus was developed by Pleias in collaboration with several partners and contributed in-kind to the Current AI initiative.

Common Corpus

Datasets can also be filtered by language (e.g., English, Hungarian, Chinese), license (e.g., CC, GPL), and task type (e.g., image classification, image-to-image translation).

```
ds = load_dataset("HuggingFaceH4/MATH-500")
```

```
README.md:   0%|          | 0.00/412 [00:00<?, ?B/s]
```

```
test.jsonl:   0%|          | 0.00/447k [00:00<?, ?B/s]
```

```
Generating test split:   0%|          | 0/500 [00:00<?, ? examples/s]
```

MATH-500 Dataset

```
ds = load_dataset("HuggingFaceH4/MATH-500")

print(ds)

df = ds["test"].to_pandas()

print(df.head())
```

```
DatasetDict({
    test: Dataset({
        features: ['problem', 'solution', 'answer', 'subject', 'level',
'unique_id'],
        num_rows: 500
    })
})
                                             problem  \
0  Convert the point $(0,3)$ in rectangular coord...
1  Define\n\[p = \sum_{k = 1}^\infty \frac{1}{k^2...
2  If $f(x) = \frac{3x-2}{x-2}$, what is the valu...
3  How many positive whole-number divisors does 1...
4  The results of a cross-country team's training...

                                            solution  \
0  We have that $r = \sqrt{0^2 + 3^2} = 3.$  Also...
1  We count the number of times $\frac{1}{n^3}$ a...
2  $f(-2)+f(-1)+f(0)=\frac{3(-2)-2}{-2-2}+\frac{3...
3  First prime factorize $196=2^2\cdot7^2$.  The ...
4  Evelyn covered more distance in less time than...

                          answer               subject  level  \
0  \left( 3, \frac{\pi}{2} \right)           Precalculus      2
1                            p - q  Intermediate Algebra      5
2                    \frac{14}{3}                Algebra      3
3                               9         Number Theory      3
```

```
4                   \text{Evelyn}              Algebra      2


                             unique_id
0            test/precalculus/807.json
1  test/intermediate_algebra/1994.json
2               test/algebra/2584.json
3          test/number_theory/572.json
4               test/algebra/1349.json
```

```
print(df['subject'].unique())
```

```
['Precalculus' 'Intermediate Algebra' 'Algebra' 'Number Theory'
 'Prealgebra' 'Geometry' 'Counting & Probability']
```

```
print(df['level'].unique())
```

```
[2 5 3 4 1]
```

25.6. Hugging Face Spaces

The **Spaces** library hosts Hugging Face `Apps`, which are web-based applications with a user-friendly interface for interacting with AI models or combinations of them.

Some applications integrate multiple models in a pipeline, utilizing different APIs to construct a composite AI system. We will now explore one of these applications.

25.7. Background Removal Tool

The project can be found at Hugging Face. The background removal tool does exactly what you'd expect, it helps extract objects from images with ease. During digital design, you may encounter images where you need specific objects isolated, and this tool is built precisely for that purpose.

It offers multiple input options, including `image upload`, `URL upload`, and a dedicated `output tab` for the final processed file. After specifying the image source, the tool removes the background by using `alpha channel manipulation`, which determines pixel transparency.

If you wanted to **host or recreate a similar project** on your own website instead of hosting it on Hugging Face, it would be entirely feasible since this project uses an open-weight model, provided it's not used for commercial purposes. However, the model's license determines usage rights, **so always check the terms before deploying.**

The easiest way to do this is by navigating to the `Files` tab inside the **Hugging Face Space** and opening the `app.py` file. This file contains the entire pipeline, including function calls and UI elements used in the Space. To recreate a nearly identical project, we will first implement it **without a GUI**, focusing solely on the core functionality. Once the model pipeline is working, we will then integrate **Gradio** on top of it, following the same approach used by Hugging Face to provide an interface.

```
torch.set_float32_matmul_precision("high")

# Load the BiRefNet model
# https://github.com/ZhengPeng7/BiRefNet
```

```
birefnet = AutoModelForImageSegmentation.from_pretrained(
    "ZhengPeng7/BiRefNet", trust_remote_code=True
).to(device)

transform_image = transforms.Compose([
    transforms.Resize((1024, 1024)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225]),
])
```

```
config.json:   0%|          | 0.00/405 [00:00<?, ?B/s]
```

```
BiRefNet_config.py:   0%|          | 0.00/298 [00:00<?, ?B/s]
```

```
A new version of the following files was downloaded from
https://huggingface.co/ZhengPeng7/BiRefNet:
- BiRefNet_config.py
. Make sure to double-check they do not contain any added malicious code. To
avoid downloading new versions of the code file, you can pin a revision.
```

```
birefnet.py:   0%|          | 0.00/92.0k [00:00<?, ?B/s]
```

```
A new version of the following files was downloaded from
https://huggingface.co/ZhengPeng7/BiRefNet:
- birefnet.py
. Make sure to double-check they do not contain any added malicious code. To
avoid downloading new versions of the code file, you can pin a revision.
/usr/local/lib/python3.11/dist-packages/timm/models/layers/__init__.py:48:
FutureWarning: Importing from timm.models.layers is deprecated, please import
via timm.layers
  warnings.warn(f"Importing from {__name__} is deprecated, please import via
timm.layers", FutureWarning)
/usr/local/lib/python3.11/dist-packages/timm/models/registry.py:4:
FutureWarning: Importing from timm.models.registry is deprecated, please import
via timm.models
  warnings.warn(f"Importing from {__name__} is deprecated, please import via
timm.models", FutureWarning)
```

```
model.safetensors:   0%|          | 0.00/885M [00:00<?, ?B/s]
```

25.7.1. Downloading a Test Image

```
!wget -q -O /content/sample.jpg https://images.cocodataset.org/val2017/000000039769.jpg
```

```
image_path = '/content/sample.jpg'
img = Image.open(image_path)
display(img)
```

```
def process(image: Image.Image) -> Image.Image:
    """
    Apply the segmentation model to an image and return the image with an alpha mask.

    Args:
        image (PIL.Image.Image): The input image to process.

    Returns:
        PIL.Image.Image: The processed image with an alpha channel representing the mask.
```

```
    """
    original_size = image.size
    input_tensor = transform_image(image).unsqueeze(0).to(device)

    with torch.no_grad():
        # Get 0-1 mask using sigmoid
        prediction = birefnet(input_tensor)[-1].sigmoid().cpu()

    mask = transforms.ToPILImage()(prediction.squeeze())
    # Bilinear interpolation
    mask = mask.resize(original_size, Image.BILINEAR)

    image = image.convert("RGBA")
    image.putalpha(mask)
    return image

def process_upload(image_path: str) -> str:
    """
    Process an image from a file path and save the result.

    Args:
        image_path (str): The file path to the input image.

    Returns:
        str: The file path to the processed image.
    """
    image = Image.open(image_path).convert("RGB")
    processed_image = process(image)
    output_path = f"{image_path.rsplit('.', 1)[0]}_processed.png"
    processed_image.save(output_path)
    print(f"Processed image saved as: {output_path}")
    return output_path

def process_url(image_url: str) -> str:
    """
    Download an image from a URL, process it, and save the result.

    Args:
        image_url (str): The URL of the image to download and process.

    Returns:
        str: The file path to the processed image.
    """
    response = requests.get(image_url)
    image = Image.open(BytesIO(response.content)).convert("RGB")
    processed_image = process(image)
    output_path = "processed_image.png"

    processed_image.save(output_path)
    print(f"Processed image saved as: {output_path}")
    return output_path
```

```
start_time = time.time()
process_upload(image_path)
print(f"Background Removal completed in {time.time() - start_time:.2f} seconds")
```

```
output_path = '/content/sample_processed.png'
img = Image.open(output_path)
display(img)
```

```
start_time = time.time()
input_url = 'https://images.cocodataset.org/val2017/000000037777.jpg'
process_url(input_url)
print(f"Background Removal completed in {time.time() - start_time:.2f} seconds")
```

```
Processed image saved as: processed_image.png
Background Removal completed in 1.52 seconds
```

```
output_path = '/content/processed_image.png'
img = Image.open(output_path)
display(img)
```

25.8. Adding a Gradio Layer on Top

```
def process_upload(image):
    """
    Process an uploaded image.

    Args:
        image (PIL.Image.Image or numpy.ndarray): The uploaded image.

    Returns:
        tuple: Processed and original images as PIL.Image.Image objects.
    """
    if isinstance(image, np.ndarray): # Gradio returns a NumPy array
        image = Image.fromarray(image) # Convert NumPy array to PIL Image
    original_image = image.convert("RGB")
    processed_image = process(original_image)
    return processed_image, original_image

def process_url(url):
    """
    Process an image from a URL.

    Args:
        url (str): URL of the image.

    Returns:
        tuple: Processed and original images as PIL.Image.Image objects.
    """
    response = requests.get(url)
    original_image = Image.open(BytesIO(response.content)).convert("RGB")
    processed_image = process(original_image)
    return processed_image, original_image
```

25.9. Gradio UI Setup

```
image_upload = gr.Image(label="Upload an Image")
image_file_upload = gr.File(label="Upload an Image File")
url_input = gr.Textbox(label="Paste an Image URL")
output_file = gr.File(label="Output PNG File")

tab1 = gr.Interface(
    process_upload, inputs=image_upload, outputs=[gr.Image(), gr.Image()], examples=['/content/sample.jpg']
)
tab2 = gr.Interface(
```

```
    process_url, inputs=url_input, outputs=[gr.Image(), gr.Image()], examples=['https://images.cocodataset.org/
  val2017/000000037777.jpg']
)

demo = gr.TabbedInterface(
    [tab1, tab2], ["Image Upload", "URL Input"], title="Background Removal Tool"
)
```

```
demo.close()
demo.launch(debug = True, share=True)
```

```
Colab notebook detected. This cell will run indefinitely so that you can see
errors and logs. To turn off, set debug=False in launch().
* Running on public URL: https://d1c443bf7922d76f8b.gradio.live

This share link expires in 72 hours. For free permanent hosting and GPU
upgrades, run `gradio deploy` from the terminal in the working directory to
deploy to Hugging Face Spaces (https://huggingface.co/spaces)
```

```
Keyboard interruption in main thread... closing server.
Killing tunnel 127.0.0.1:7861 <> https://d1c443bf7922d76f8b.gradio.live
```

25.10. Text-to-Image with Diffusion Upscaler

This Hugging Face Space features the Stable Diffusion v1-5 latent diffusion model from **Xyn AI**. The **Stable-Diffusion-v1-5** checkpoint was initialized with the weights of the Stable-Diffusion-v1-2 checkpoint and subsequently fine-tuned on 595k steps at resolution 512x512 on `laion-aesthetics v2 5+` and 10% dropping of the text-conditioning to improve classifier-free guidance sampling. Additionally, the app integrates **Stability AI's** SD-2X latent upscaler to enhance image resolution, ensuring high-quality outputs.

```
def initialize_t2i_pipeline(model_name, safety_checker=None):
    pipe = StableDiffusionPipeline.from_pretrained(model_name, torch_dtype=torch.float16,
  safety_checker=safety_checker)
    # DPMSolver (and the improved version DPMSolver++) is a fast dedicated high-order solver for diffusion
  ODEs with convergence order guarantee.
    # Empirically, DPMSolver sampling with only 20 steps can generate high-quality samples, and it can
  generate quite good samples even in 10 steps.
    pipe.scheduler = DPMSolverMultistepScheduler.from_config(pipe.scheduler.config)
    return pipe

def initialize_upscaler_pipeline(model_name):
    upscaler = DiffusionPipeline.from_pretrained(model_name, torch_dtype=torch.float16)
    return upscaler

def generate_and_upscale_image(prompt, neg_prompt, guidance, steps, model_name, upscaler_name, seed = 0):
    if seed == 0:
        seed = random.randint(0, 2147483647)
    generator = torch.Generator("cuda").manual_seed(seed)

    t2i_pipe = initialize_t2i_pipeline(model_name).to(device)
    upscaler_pipe = initialize_upscaler_pipeline(upscaler_name).to(device)
```

```
    # torch.autocast automatically selects the best floating-point precision for computations (float16,
  float32)
    with torch.autocast("cuda"):
        low_res_latents = t2i_pipe(
            prompt,
            negative_prompt=neg_prompt,
            num_inference_steps=steps,
            guidance_scale=guidance,
            generator=generator,
            output_type="latent",
        ).images

    with torch.no_grad():
        low_res_image = t2i_pipe.decode_latents(low_res_latents)
        low_res_image = t2i_pipe.numpy_to_pil(low_res_image)

    with torch.autocast("cuda"):
        up_res_image = upscaler_pipe(
            prompt=prompt,
            negative_prompt=neg_prompt,
            image=low_res_latents,
            num_inference_steps=20,
            guidance_scale=0,
            generator=generator,
        ).images

    return low_res_image[0], up_res_image[0]
```

```
neg_prompt = "low quality, blurry, nsfw"
guidance = 7.5
steps = 25
model_name = "runwayml/stable-diffusion-v1-5"
upscaler_name = "stabilityai/sd-x2-latent-upscaler"
```

```
prompt = "a photo of an astronaut riding a horse on mars"

start_time = time.time()
low_res_image, up_res_image = generate_and_upscale_image(prompt, neg_prompt, guidance, steps, model_name,
  upscaler_name)
print(f"Image generation and upscaling completed in {time.time() - start_time:.2f} seconds")
```

```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

```
You have disabled the safety checker for <class 'diffusers.pipelines.stable_diff
usion.pipeline_stable_diffusion.StableDiffusionPipeline'> by passing
`safety_checker=None`. Ensure t ... [471 chars truncated]
```

```
model_index.json:   0%|          | 0.00/408 [00:00<?, ?B/s]
```

```
Fetching 12 files:   0%|          | 0/12 [00:00<?, ?it/s]
```

```
text_encoder%2Fconfig.json:   0%|          | 0.00/645 [00:00<?, ?B/s]
```

```
tokenizer%2Fmerges.txt:   0%|          | 0.00/525k [00:00<?, ?B/s]
```

```
scheduler%2Fscheduler_config.json:   0%|          | 0.00/301 [00:00<?, ?B/s]
```

```
tokenizer%2Ftokenizer_config.json:   0%|          | 0.00/831 [00:00<?, ?B/s]
```

```
unet%2Fconfig.json:   0%|          | 0.00/1.23k [00:00<?, ?B/s]
```

```
tokenizer%2Fspecial_tokens_map.json:   0%|          | 0.00/472 [00:00<?, ?B/s]
```

```
model.safetensors:   0%|          | 0.00/492M [00:00<?, ?B/s]
```

```
tokenizer%2Fvocab.json:   0%|          | 0.00/1.06M [00:00<?, ?B/s]
```

```
diffusion_pytorch_model.safetensors:   0%|          | 0.00/1.49G [00:00<?, ?B/s]
```

```
vae%2Fconfig.json:   0%|          | 0.00/653 [00:00<?, ?B/s]
```

```
diffusion_pytorch_model.safetensors:   0%|          | 0.00/335M [00:00<?, ?B/s]
```

```
Loading pipeline components...:   0%|          | 0/5 [00:00<?, ?it/s]
```

```
0%|          | 0/25 [00:00<?, ?it/s]
```

```
/usr/local/lib/python3.11/dist-
packages/diffusers/pipelines/stable_diffusion/pipeline_stable_diffusion.py:594:
FutureWarning: The decode_latents method is deprecated and will be re ... [69
chars truncated]
  deprecate("decode_latents", "1.0.0", deprecation_message, standard_warn=False)
```

```
0%|          | 0/20 [00:00<?, ?it/s]
```

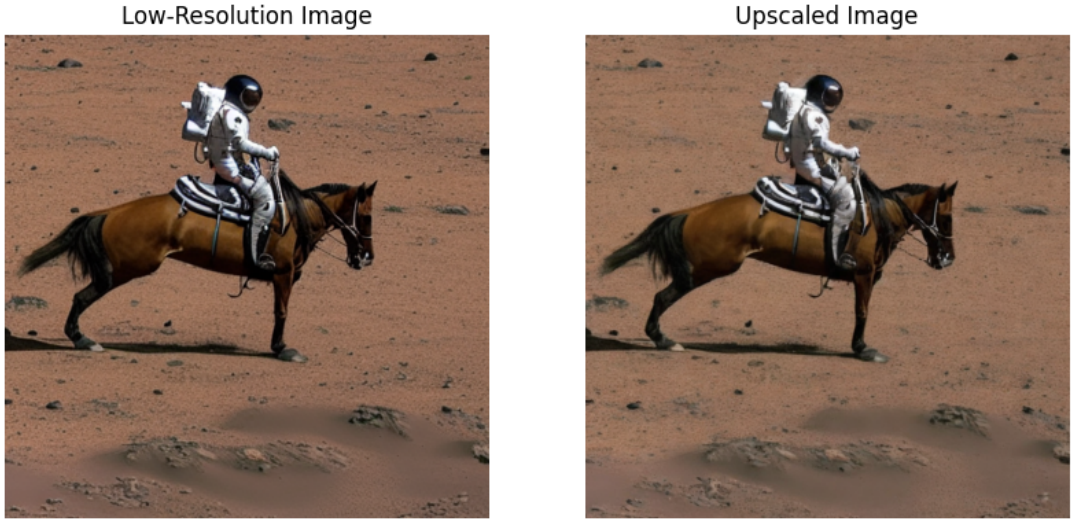
```
Image generation and upscaling completed in 80.08 seconds
```

```
plt.figure(figsize=(10, 5))

plt.subplot(1, 2, 1)
plt.imshow(low_res_image)
plt.title("Low-Resolution Image")
plt.axis('off')

plt.subplot(1, 2, 2)
plt.imshow(up_res_image)
plt.title("Upscaled Image")
plt.axis('off')

plt.show()
```



```
low_res_path = "low_res_image.png"
high_res_path = "high_res_image.png"
low_res_image.save(low_res_path)
up_res_image.save(high_res_path)
```

```
low_res_img = Image.open(low_res_path)
high_res_img = Image.open(high_res_path)
```





25.11. QR Code Generator with Diffusion

QR codes are two-dimensional barcodes designed to encode information such as URLs, contact details, or text, which can be quickly scanned using a smartphone or dedicated scanner. They use a pattern of black and white squares to store data, with error correction mechanisms that allow them to remain functional even if partially damaged.

Traditionally, QR codes are `monochrome` and visually unremarkable. However, by leveraging diffusion models and a control QR code image, it is now possible to generate custom, visually enhanced QR codes that remain fully scannable while incorporating complex designs, colors, and artistic elements.

The QR Code AI Art Generator does exactly that.

```
def resize_for_condition_image(input_image: Image.Image, resolution: int) -> Image.Image:
    """
    Resize an image to a specified resolution while maintaining aspect ratio,
    ensuring dimensions are multiples of 64.
```

```
    Args:
        input_image (Image.Image): The input image to be resized.
        resolution (int): Target resolution for the smallest dimension.

    Returns:
        Image.Image: The resized image.
    """
    input_image = input_image.convert("RGB")
    W, H = input_image.size
    scale_factor = float(resolution) / min(H, W)

    H = int(round((H * scale_factor) / 64.0)) * 64
    W = int(round((W * scale_factor) / 64.0)) * 64

    return input_image.resize((W, H), resample=Image.LANCZOS)

def generate_qr_code(content: str, resolution: int = 768) -> Image.Image:
    """
    Generate a high-error-correction QR code for a given content URL and resize it.

    Args:
        content (str): The text or URL to encode in the QR code. Defaults to Google.
        resolution (int): The target resolution for resizing. Defaults to 768.

    Returns:
        Image.Image: A resized QR code image.
    """
    print("Generating QR Code from content")

    qr = qrcode.QRCode(
        version=1,
        error_correction=qrcode.constants.ERROR_CORRECT_H,
        box_size=10,
        border=4,
    )
    qr.add_data(content)
    qr.make(fit=True)

    qr_image = qr.make_image(fill_color="black", back_color="white")
    return resize_for_condition_image(qr_image, resolution)

def process_qr(prompt: str, content: str, qrcode_image = None, guidance_scale = 8.5, strength = 0.9,
   controlnet_conditioning_scale = 1.8, seed = 444, negative_prompt: str = "ugly, disfigured, low quality,
   blurry, nsfw") -> Image.Image:
    """
    Generate an artistic QR code using a ControlNet-conditioned Stable Diffusion model.

    Args:
        controlnet_model (str): The pre-trained ControlNet model to use.
        diffusion_model (str): The pre-trained Stable Diffusion model.
        prompt (str): The text prompt for generating the stylized QR code.

    Returns:
        Image.Image: The generated artistic QR code.
    """
    print("Initializing ControlNet and Stable Diffusion pipeline...")
    controlnet = ControlNetModel.from_pretrained("DionTimmer/controlnet_qrcode-control_v1p_sd15",
   torch_dtype=torch.float16).to(device)
    pipe = StableDiffusionControlNetPipeline.from_pretrained(
```

```
        "runwayml/stable-diffusion-v1-5",
        controlnet=controlnet,
        torch_dtype=torch.float16
    ).to(device)
    pipe.scheduler = DPMSolverMultistepScheduler.from_config(pipe.scheduler.config)

    if qrcode_image:
        qrcode_image = Image.open(qrcode_image)
    else:
        qrcode_image = generate_qr_code(content = content)

    print("Generating artistic QR code...")
    result = pipe(
        prompt=prompt,
        negative_prompt=negative_prompt,
        image=qrcode_image,
        width=768,
        height=768,
        guidance_scale=guidance_scale,
        controlnet_conditioning_scale=controlnet_conditioning_scale,
        generator=torch.Generator(),
        strength=strength,
        num_inference_steps=20,
        sampler="DPM++ Karras SDE",
        use_qr_code_as_init_image=True,
    ).images[0]

    return result
```

```
content = "google.com"
qrcode_image = generate_qr_code(content = content)
plt.imshow(qrcode_image)
plt.axis('off')
plt.show()
```

```
Generating QR Code from content
```



```
qr_prompt = "a misterious puzzle world with cryptic things and cyphers"
qr_path = "artistic_qr_code.png"

result = process_qr(prompt = qr_prompt, content = content, qrcode_image = "Kép.png")
result.save(qr_path)
```

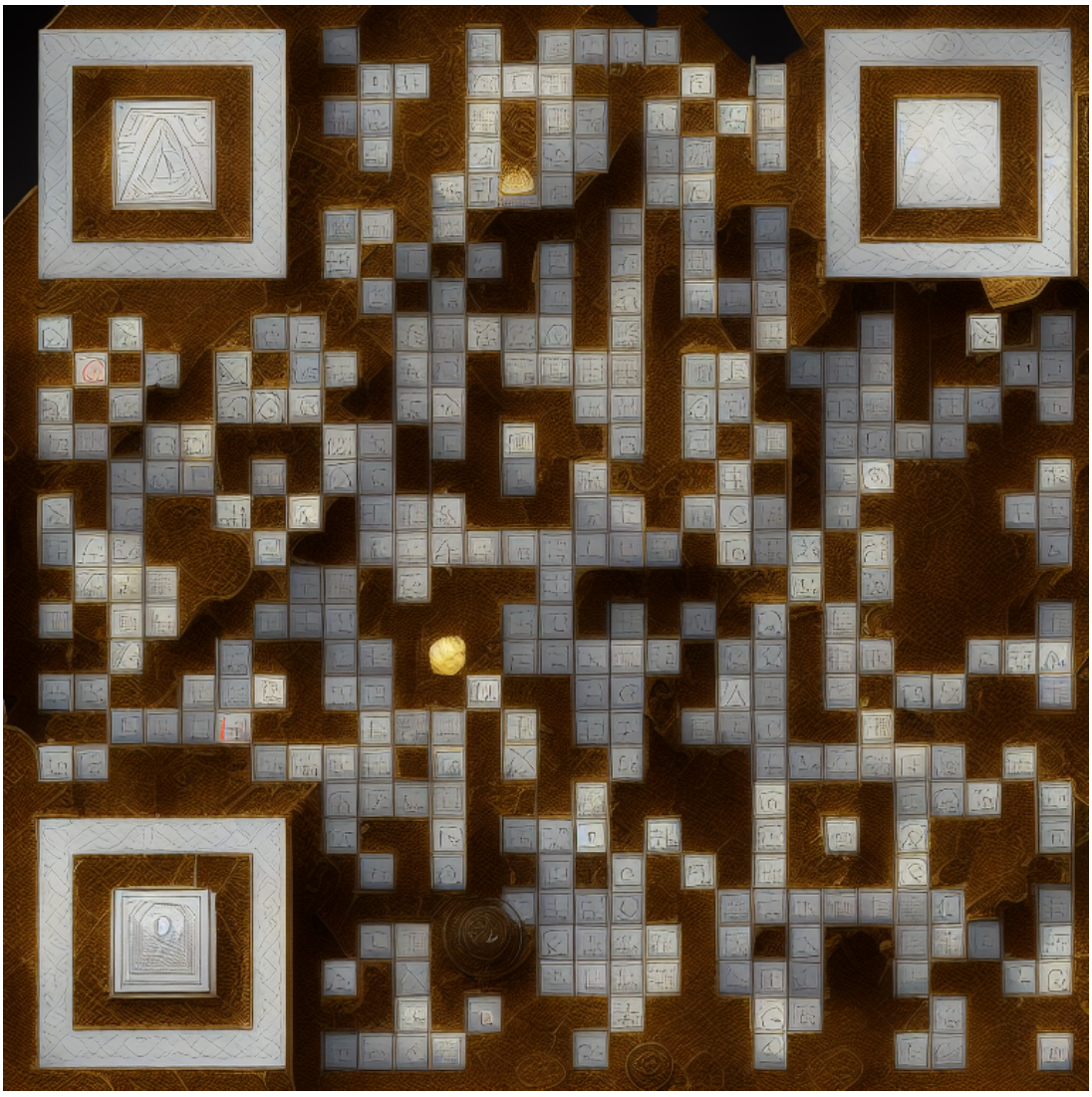
```
Initializing ControlNet and Stable Diffusion pipeline...
```

```
Loading pipeline components...:   0%|          | 0/7 [00:00<?, ?it/s]
```

```
Generating artistic QR code...
```

```
0%|          | 0/20 [00:00<?, ?it/s]
```

```
qr_artistic_img = Image.open(qr_path)
display(qr_artistic_img)
```



25.12. Let's Do that in Gradio as well

```
with gr.Blocks() as qr_code_generator:
    gr.Markdown("## AI-Enhanced QR Code Generator")
    gr.Markdown("### Generate beautiful AI-stylized QR Codes!")

    with gr.Row():
        with gr.Column():
            qr_code_content = gr.Textbox(label="QR Code Content (URL or Text)", placeholder="Enter your text
  or URL here")
            prompt = gr.Textbox(label="AI Style Prompt", placeholder="Describe the style you want (e.g.,
  'Cyberpunk neon lights')")
            negative_prompt = gr.Textbox(label="Negative Prompt", value="ugly, disfigured, low quality,
  blurry, nsfw")
            uploaded_qr_image = gr.Image(label="Upload QR Code (Optional)", type="pil")

        with gr.Column():
            guidance_scale = gr.Slider(1, 15, value=7.5, step=0.1, label="Guidance Scale (Strength of prompt)")
```

```
            strength = gr.Slider(0.9, 1.6, value=1.3, step=0.01, label="ControlNet Conditioning Scale
    (Strength of QR Code Image)")
            controlnet_conditioning_scale = gr.Slider(0.5, 1.0, value=0.8, step=0.01, label="Image
    Modification Strength")
            seed = gr.Slider(0, 2147483647, value=42, step=1, label="Seed (0 = Random)")
            generate_button = gr.Button("Generate AI QR Code")

     with gr.Row():
         output_image = gr.Image(label="AI-Enhanced QR Code", width=768, height=768)

     generate_button.click(
         process_qr,
         inputs=[prompt, guidance_scale, strength, controlnet_conditioning_scale, seed, qr_code_content,
    negative_prompt],
         outputs=[output_image]
     )
```

```
qr_code_generator.close()
qr_code_generator.launch(share=True, debug = True)
```

```
Colab notebook detected. This cell will run indefinitely so that you can see
errors and logs. To turn off, set debug=False in launch().
* Running on public URL: https://f615f81200b3c80a80.gradio.live

This share link expires in 72 hours. For free permanent hosting and GPU
upgrades, run `gradio deploy` from the terminal in the working directory to
deploy to Hugging Face Spaces (https://huggingface.co/spaces)
```

```
Initializing ControlNet and Stable Diffusion pipeline...
```

```
Loading pipeline components...:   0%|          | 0/7 [00:00<?, ?it/s]
```

```
Generating QR Code from content
Generating artistic QR code...
```

```
0%|          | 0/40 [00:00<?, ?it/s]
```

```
Initializing ControlNet and Stable Diffusion pipeline...
```

```
Loading pipeline components...:   0%|          | 0/7 [00:00<?, ?it/s]
```

```
Generating QR Code from content
Generating artistic QR code...
```

```
0%|          | 0/40 [00:00<?, ?it/s]
```

```
Keyboard interruption in main thread... closing server.
Killing tunnel 127.0.0.1:7860 <> https://f615f81200b3c80a80.gradio.live
```

25.13. Homework (Optional)

1. **Pipeline quick-start.** Pick three different tasks from the Hub (e.g. summarisation, sentiment, question-answering) and build a `pipeline(...)` for each. Run one example per task and paste the outputs.

2. **Model comparison.** For English-German translation, compare `Helsinki-NLP/opus-mt-en-de` with `facebook/nllb-200-distilled-600M`. Run the same three source sentences through both, and comment on output quality, latency, and memory footprint.
3. **Custom pipeline.** Build a translation pipeline from scratch (without `pipeline()`): load the tokenizer, model, and do `.generate()` yourself. Reproduce the output of the high-level API and explain each step.
4. **Dataset slicing.** Load a small subset of `wikipedia` in a language of your choice, filter to articles with more than 1 000 tokens, and tokenise the first 50 entries. Compute and plot the token-length distribution.
5. **Gradio UI.** Wrap one of the pretrained translators in a Gradio `Blocks` UI with three components: a source-text input, a target-language dropdown, and the translated output. Test it locally.
6. **Pushing to the Hub.** Save the state-dict of one of the models you trained earlier in the course (e.g. the image classifier). Create a model repo on Hugging Face and push both the weights and a `README.md` card. Paste the public URL.

Chapter 26

The Vision Transformer (ViT)

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1yiXXfgpNoKH8ONzf5RuyAf8yRi8HzrTA?usp=sharingv

The **attention mechanism** has become an important tool in deep learning, allowing models to **weigh different parts of the input when making predictions**. While originally developed for tasks like machine translation, attention has also proven useful in computer vision.

Vision Transformer [21] s (ViTs) apply the same core idea by dividing an image into small patches and processing them as a sequence. Using **self-attention,** the model **can relate information across different parts of the image, without relying on convolutions or fixed local patterns.** This approach makes it possible to capture both short- and long-range relationships in the visual input and has shown strong performance on various image analysis tasks.

26.1. Necessary Imports

```
!pip install --quiet pytorch-lightning
```

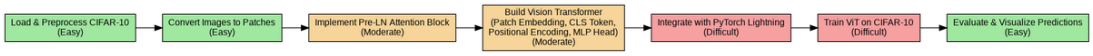
```
import os
import numpy as np
import random
import math
import matplotlib
import gdown
import seaborn as sns
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim
import torchvision
import pytorch_lightning as pl
from torchvision import transforms
from pytorch_lightning.callbacks import ModelCheckpoint, LearningRateMonitor
from pytorch_lightning.loggers import TensorBoardLogger
from IPython.display import Image as IMG
from torchvision.datasets import CIFAR10
from torchmetrics.functional import accuracy
from torch.utils.data import random_split
from functools import partial
from tqdm.notebook import tqdm

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
```

```
matplotlib.rcParams['lines.linewidth'] = 2.0  # Set the default line width for plots to 2.0 for better
   visibility
sns.reset_orig()                              # Reset Seaborn's settings to the original Matplotlib defaults
plt.set_cmap('cividis')                       # Set the default colormap for plots to 'cividis', which is
   colorblind-friendly
pl.seed_everything(42)                        # Set a global random seed (42) for reproducibility across
   different libraries
torch.backends.cudnn.deterministic = True     # Ensure deterministic behavior for CuDNN operations (useful for
   reproducibility)
torch.backends.cudnn.benchmark = False        # Disable CuDNN benchmarking to prevent adaptive optimizations
   that may affect reproducibility
```

```
INFO:lightning_fabric.utilities.seed:Seed set to 42
```

```
<Figure size 640x480 with 0 Axes>
```



We provide a **pre-trained Vision Transformer**, which we download in the next cell. However, Vision Transformers can be relatively quickly trained on **CIFAR-10 [1]** , with an overall training time of **less than an hour** on an **NVIDIA Titan RTX**.

Feel free to experiment with training your own Transformer **once you went through the whole notebook.**

Ensure that your `MODEL_PATH` is correctly set before executing the code. This pre-trained model will be loaded into our Transformer model to bypass the training process and save time.

```
MODEL_PATH = "/content/saved_models/"
os.makedirs(MODEL_PATH, exist_ok=True)

GDRIVE_FILE_ID = "1VYB69EosJ86qdF8mNcdqVwOwCENMyu2J"
MODEL_FILENAME = "ViT.ckpt"
DEST_PATH = os.path.join(MODEL_PATH, MODEL_FILENAME)

gdown.download(f"https://drive.google.com/uc?id={GDRIVE_FILE_ID}", MODEL_FILENAME, quiet=False)
os.rename(MODEL_FILENAME, DEST_PATH)

print(f"Model downloaded and saved to: {DEST_PATH}")
```

```
Downloading...
From: https://drive.google.com/uc?id=1VYB69EosJ86qdF8mNcdqVwOwCENMyu2J
To: /content/ViT.ckpt
100%|##########| 12.8M/12.8M [00:00<00:00, 24.7MB/s]
```

```
Model downloaded and saved to: /content/saved_models/ViT.ckpt
```

26.2. Loading Our Dataset

We load the **CIFAR-10 dataset** below.

The constants in `transforms.Normalize` are chosen to **scale and shift** the data so that it has a **zero mean** and a **standard deviation of one**.

```
test_transform = transforms.Compose([transforms.ToTensor(),
                                     transforms.Normalize([0.49139968, 0.48215841, 0.44653091], [0.24703223, 0.
   24348513, 0.26158784])
                                     ])

train_transform = transforms.Compose([transforms.RandomHorizontalFlip(),
                                      transforms.RandomResizedCrop((32,32),scale=(0.8,1.0),ratio=(0.9,1.1)),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.49139968, 0.48215841, 0.44653091], [0.24703223,
   0.24348513, 0.26158784])
                                     ])

# Loading the training dataset. We need to split it into a training and validation part
# We need to do a little trick because the validation set should not use the augmentation.
train_dataset = CIFAR10(root='/content/data', train=True, transform=train_transform, download=True)
val_dataset = CIFAR10(root='/content/data', train=True, transform=test_transform, download=False)

train_set, _ = torch.utils.data.random_split(train_dataset, [45000, 5000])
_, val_set = torch.utils.data.random_split(val_dataset, [45000, 5000])
test_set = CIFAR10(root='/content/data', train=False, transform=test_transform, download=False)

train_loader = data.DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True, pin_memory=True,
   num_workers=2)
val_loader = data.DataLoader(val_set, batch_size=128, shuffle=False, drop_last=False, num_workers=2)
test_loader = data.DataLoader(test_set, batch_size=128, shuffle=False, drop_last=False, num_workers=2)
```

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
/content/saved_models/cifar-10-python.tar.gz
```

```
100%|##########| 170M/170M [00:02<00:00, 74.0MB/s]
```

```
Extracting /content/saved_models/cifar-10-python.tar.gz to
/content/saved_models/
Files already downloaded and verified
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to
/content/data/cifar-10-python.tar.gz
```

```
100%|##########| 170M/170M [00:02<00:00, 78.4MB/s]
```

```
Extracting /content/data/cifar-10-python.tar.gz to /content/data
```

```
/usr/local/lib/python3.11/dist-packages/torch/utils/data/dataloader.py:617:
UserWarning: This DataLoader will create 4 worker processes in total. Our
suggested max number of worker ... [257 chars truncated]
  warnings.warn(
```

```
num_images = 4
cifar_images = torch.stack([val_set[idx][0] for idx in range(num_images)], dim=0)
img_grid = torchvision.utils.make_grid(cifar_images, nrow=4, normalize=True, pad_value=0.9)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(8,8))
plt.title("Image examples of the CIFAR10 dataset")
plt.imshow(img_grid)
plt.axis('off')
plt.show()
plt.close()
```



26.3. Patch embedding and the [CLS] token

A transformer expects a sequence of vectors. An image is a 2D grid of pixels. Bridging these is the job of *patch embedding*: the input image is split into non-overlapping patches (typically 16x16 pixels for natural images, 4x4 for tiny CIFAR-style inputs), each patch is flattened, and each flat patch goes through a linear projection to the model dimension. For a 32x32 CIFAR image with patch size 4, this produces 64 tokens of the model dimension.

Without position information the resulting sequence is order-agnostic, the network would treat a scrambled image the same as the original. A *learned* positional embedding is added to every token, typically as a lookup table indexed by position. The network discovers an effective positional code during training. Sinusoidal encodings also work and are still common for inputs of unknown size.

The `[CLS]` token is an extra token prepended to the sequence *before* the transformer blocks. It contains learnable parameters just like any other embedding, it attends over all patch tokens throughout the stack, and its final hidden state is fed into the classification head. In effect the network trains `[CLS]` to be a summary representation of the whole image, the same trick BERT [18] uses for sentence classification. Alternatives exist (global average pooling over patch tokens, for instance), but the `[CLS]` approach is the most widely used.

26.4. Transformers for Image Classification

Transformers were originally designed to **process sets**, as they are **permutation-equivariant architectures**, meaning that if the **input order is permuted**, the **output is permuted accordingly**. When applying Transformers to **sequential data**, we introduced **positional encodings** so the model could **learn** the importance of order.

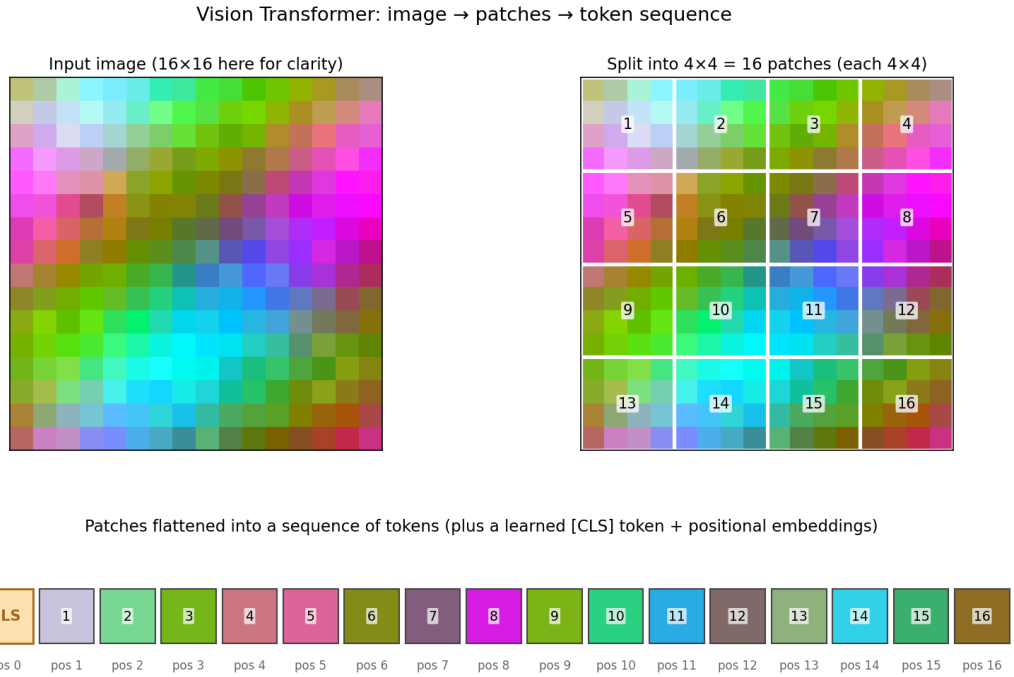
So, why not apply the **same idea to images**?

This is exactly what **Alexey Dosovitskiy et al.** proposed in their paper, **"An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale"**. The **Vision Transformer (ViT)** is an image classification model that treats **images as sequences of smaller patches**.

26.4.1. How does the Vision Transformer work?

1. An image (e.g., 48×48 **pixels**) is **split** into **smaller patches** (e.g., 16×16 **patches**).
2. Each patch is treated as a **"word"** or **"token"** and is **projected into a feature space**.
3. **Positional encodings** are added to retain **spatial information**.
4. A special **classification token** (`[CLS]`) is added to aggregate global information.
5. The sequence is **processed by a Transformer encoder** to **learn meaningful representations**.
6. The final output is used for **image classification**.

A **visualization** of the **Vision Transformer architecture** is shown below
(Figure credit: Phil Wang):



We will **implement the Vision Transformer step by step**, covering **all essential components**.

The **first step** is **image preprocessing**, splitting an image of size $N \times N$ into $(N/M)^2$ **patches** of size $M \times M$, which will serve as the **input tokens** for the Transformer.

```
def img_to_patch(x, patch_size, flatten_channels=True):
    """
    Inputs:
        x - torch.Tensor representing the image of shape [B, C, H, W]
        patch_size - Number of pixels per dimension of the patches (integer)
        flatten_channels - If True, the patches will be returned in a flattened format
                           as a feature vector instead of a image grid.
    """
```

```
    B, C, H, W = x.shape
    x = x.reshape(B, C, H//patch_size, patch_size, W//patch_size, patch_size)
    x = x.permute(0, 2, 4, 1, 3, 5) # [B, H', W', C, p_H, p_W]
    x = x.flatten(1,2)              # [B, H'*W', C, p_H, p_W]
    if flatten_channels:
        x = x.flatten(2,4)          # [B, H'*W', C*p_H*p_W]
    return x
```

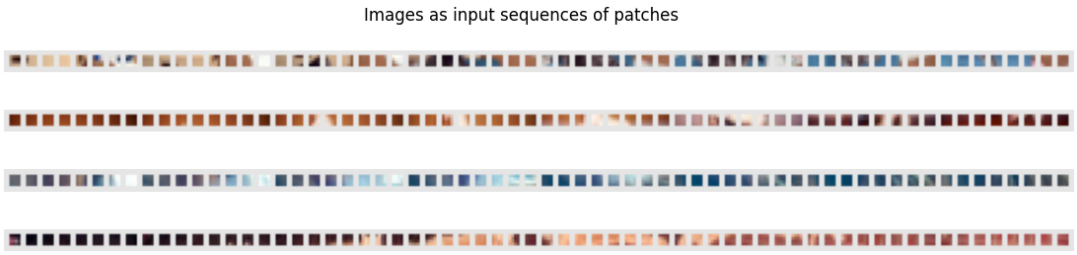
Let's take a look at how this works for our **CIFAR examples**.

For images of size 32×32, we choose a **patch size of 4**. This results in **sequences of 64 patches** (4×4 each).

Below, we visualize how the **image is split into patches** before being processed by the Vision Transformer:

```
img_patches = img_to_patch(cifar_images, patch_size=4, flatten_channels=False)

fig, ax = plt.subplots(cifar_images.shape[0], 1, figsize=(14,3))
fig.suptitle("Images as input sequences of patches")
for i in range(cifar_images.shape[0]):
    img_grid = torchvision.utils.make_grid(img_patches[i], nrow=64, normalize=True, pad_value=0.9)
    img_grid = img_grid.permute(1, 2, 0)
    ax[i].imshow(img_grid)
    ax[i].axis('off')
plt.show()
plt.close()
```



Compared to the **original images**, it is **much harder to recognize objects** from these patch sequences. Still, this is the **input** we provide to the **Transformer** for classification.

The model must **learn by itself** how to **combine patches** to recognize objects. Unlike **CNNs**, which have a **built-in assumption** that images are structured as a **grid of pixels**, the **Transformer lacks this inductive bias** in its input format.

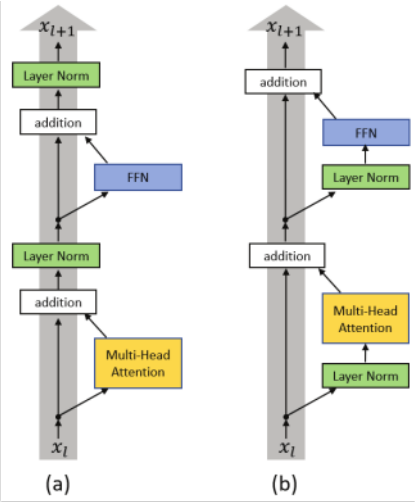
26.4.2. Building the Transformer Model

Now that we've covered **preprocessing**, we can start building the **Transformer model**.

Since we've already discussed the fundamentals of **Multi-Head Attention**, we will **directly use** PyTorch's `nn.MultiheadAttention` module
(**docs**).

Further, we implement the **Pre-Layer Normalization (Pre-LN) Transformer blocks**, as proposed by **Ruibin Xiong et al. (2020)**. Instead of applying **Layer Normalization** between **residual connections**, Pre-LN applies it **before** each residual block.

This **improves gradient flow** and **removes the need for a warm-up stage** in training. The difference between **Post-LN (standard)** and **Pre-LN** is visualized below:



(a) Post-LN Transformer layer; (b) Pre-LN Transformer layer.

26.4.3. Pre-LN Transformer Block Implementation

Now, let's implement the **Pre-LN attention block**:

```
class AttentionBlock(nn.Module):

    def __init__(self, embed_dim, hidden_dim, num_heads, dropout=0.0):
        """
        Inputs:
            embed_dim - Dimensionality of input and attention feature vectors
            hidden_dim - Dimensionality of hidden layer in feed-forward network
                         (usually 2-4x larger than embed_dim)
            num_heads - Number of heads to use in the Multi-Head Attention block
            dropout - Amount of dropout to apply in the feed-forward network
        """
        super().__init__()

        self.layer_norm_1 = nn.LayerNorm(embed_dim)
        self.attn = nn.MultiheadAttention(embed_dim, num_heads,
                                          dropout=dropout)
        self.layer_norm_2 = nn.LayerNorm(embed_dim)
        self.linear = nn.Sequential(
            nn.Linear(embed_dim, hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, embed_dim),
            nn.Dropout(dropout)
        )

    def forward(self, x):
        inp_x = self.layer_norm_1(x)
```

```
        x = x + self.attn(inp_x, inp_x, inp_x)[0]
        x = x + self.linear(self.layer_norm_2(x))
        return x
```

26.4.4. Building the Vision Transformer (ViT)

Now that we have all necessary modules, we can construct our **Vision Transformer**. Besides the **Transformer encoder**, we require the following components:

1. Linear Projection Layer

- Maps each $M \times M$ **patch** independently to a **feature vector**.
- Implemented as a simple **linear layer** that processes patches individually.

2. Classification Token (CLS Token)

- A **learnable token** added to the input sequence.
- The output **feature vector of this token** is used for **classification**.

3. Learnable Positional Encodings

- Added to the tokens before processing by the **Transformer**.
- Required for **position-dependent information**.
- Since we usually work with a **fixed resolution**, we **learn the positional encodings** instead of using sine/cosine functions.

4. MLP Head

- Takes the **final CLS token representation** and **maps it to a classification prediction**.
- Typically implemented as:
  - A **small feed-forward network**.
  - A **single linear layer** in simpler cases.

With these components in place, we can now **implement the full Vision Transformer**:

```
class VisionTransformer(nn.Module):

    def __init__(self, embed_dim, hidden_dim, num_channels, num_heads, num_layers, num_classes, patch_size,
  num_patches, dropout=0.0):
        """
        Inputs:
            embed_dim - Dimensionality of the input feature vectors to the Transformer
            hidden_dim - Dimensionality of the hidden layer in the feed-forward networks
                         within the Transformer
            num_channels - Number of channels of the input (3 for RGB)
            num_heads - Number of heads to use in the Multi-Head Attention block
            num_layers - Number of layers to use in the Transformer
            num_classes - Number of classes to predict
            patch_size - Number of pixels that the patches have per dimension
            num_patches - Maximum number of patches an image can have
            dropout - Amount of dropout to apply in the feed-forward network and
                      on the input encoding
```

```
        """
        super().__init__()

        self.patch_size = patch_size

        # Layers/Networks
        self.input_layer = nn.Linear(num_channels*(patch_size**2), embed_dim)
        self.transformer = nn.Sequential(*[AttentionBlock(embed_dim, hidden_dim, num_heads, dropout=dropout)
    for _ in range(num_layers)])
        self.mlp_head = nn.Sequential(
            nn.LayerNorm(embed_dim),
            nn.Linear(embed_dim, num_classes)
        )
        self.dropout = nn.Dropout(dropout)

        # Parameters/Embeddings
        self.cls_token = nn.Parameter(torch.randn(1,1,embed_dim))
        self.pos_embedding = nn.Parameter(torch.randn(1,1+num_patches,embed_dim))

    def forward(self, x):
        # Preprocess input
        x = img_to_patch(x, self.patch_size)
        B, T, _ = x.shape
        x = self.input_layer(x)

        # Add CLS token and positional encoding
        cls_token = self.cls_token.repeat(B, 1, 1)
        x = torch.cat([cls_token, x], dim=1)
        x = x + self.pos_embedding[:,:T+1]

        # Apply Transforrmer
        x = self.dropout(x)
        x = x.transpose(0, 1)
        x = self.transformer(x)

        # Perform classification prediction
        cls = x[0]
        out = self.mlp_head(cls)
        return out
```

26.4.5. Integrating Vision Transformer into PyTorch Lightning

Finally, we encapsulate everything into a **PyTorch Lightning Module**.

Key Implementation Choices:

- We use **`torch.optim.AdamW`** as the optimizer:
  - Adam **with a corrected weight decay implementation**.
- Since we are using the **Pre-LN Transformer**:
  - **No need** for **learning rate warmup**.
- Instead, we employ the **same learning rate scheduler** used for CNNs in our **previous tutorial on image classification**.

With these considerations in mind, let's implement the **full PyTorch Lightning module** below:

```
class ViT(pl.LightningModule):

    def __init__(self, model_kwargs, lr):
        super().__init__()
        self.save_hyperparameters()
        self.model = VisionTransformer(**model_kwargs)
        self.example_input_array = next(iter(train_loader))[0]

    def forward(self, x):
        return self.model(x)

    def configure_optimizers(self):
        optimizer = optim.AdamW(self.parameters(), lr=self.hparams.lr)
        lr_scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[100,150], gamma=0.1)
        return [optimizer], [lr_scheduler]

    def _calculate_loss(self, batch, mode="train"):
        imgs, labels = batch
        preds = self.model(imgs)
        loss = F.cross_entropy(preds, labels)
        acc = (preds.argmax(dim=-1) == labels).float().mean()

        self.log(f'{mode}_loss', loss)
        self.log(f'{mode}_acc', acc)
        return loss

    def training_step(self, batch, batch_idx):
        loss = self._calculate_loss(batch, mode="train")
        return loss

    def validation_step(self, batch, batch_idx):
        self._calculate_loss(batch, mode="val")

    def test_step(self, batch, batch_idx):
        self._calculate_loss(batch, mode="test")
```

26.5. Experiments

Vision Transformers are **typically applied to large-scale image classification benchmarks** such as **ImageNet** to fully leverage their potential. However, in this experiment, we take a step back and ask:

Can Vision Transformers also perform well on smaller benchmarks like CIFAR10?

To investigate this, we **train a Vision Transformer from scratch** on the **CIFAR10 dataset**.

26.5.1. Training Function

Before we proceed with training, we first implement a **training function** for our **PyTorch Lightning module**. This function will: - Train the Vision Transformer **from scratch** on CIFAR10. - **Load a pre-trained model** (if available) to skip training.

Let's define the **training function** below:

```python
def train_model(**kwargs):
    trainer = pl.Trainer(default_root_dir=os.path.join(MODEL_PATH, "ViT"),
                         accelerator="gpu" if str(device).startswith("cuda") else "cpu",
                         devices=1,
                         max_epochs=180,
                         callbacks=[ModelCheckpoint(save_weights_only=True, mode="max", monitor="val_acc"),
                                    LearningRateMonitor("epoch")])
    trainer.logger._log_graph = True         # If True, we plot the computation graph in tensorboard
    trainer.logger._default_hp_metric = None # Optional logging argument that we don't need

    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(MODEL_PATH, "ViT.ckpt")
    if os.path.isfile(pretrained_filename):
        print(f"Found pretrained model at {pretrained_filename}, loading...")
        model = ViT.load_from_checkpoint(pretrained_filename)
    else:
        model = ViT(**kwargs)
        trainer.fit(model, train_loader, val_loader)
        model = ViT.load_from_checkpoint(trainer.checkpoint_callback.best_model_path)

    val_result = trainer.test(model, val_loader, verbose=False)
    test_result = trainer.test(model, test_loader, verbose=False)
    result = {"test": test_result[0]["test_acc"], "val": val_result[0]["test_acc"]}
    return model, result
```

Now, we can **start training our model**.

26.5.2. Hyperparameter Selection & Intuition

As seen in our implementation, we have several **hyperparameters** that need to be tuned. During the creation of this notebook, we **performed a small grid search** and found the **best performing hyperparameters**, which are listed below. However, let's first discuss **the impact of each hyperparameter** and the **intuition** behind our choices.

26.5.3. Patch Size

- The **patch size** determines how an image is split into smaller patches before being processed by the Transformer.
- **Smaller patches** result in **longer input sequences**, which **increase the model's complexity** but also **require more memory** due to the **quadratic** memory usage in the **attention layer**.
- **Smaller patches** also make it harder for the model to learn spatial relationships.
- We experimented with **patch sizes of 2, 4, and 8**, corresponding to input sequence lengths of **256, 64, and 16**, respectively.
- **Best choice: 4×4 patches (sequence length = 64).**

26.5.4. Embedding & Hidden Dimensionality

- Similar to MLPs, increasing **embedding dimensionality** allows the model to learn more complex patterns but also **increases training time**.

- In Transformers, **query-key size** is an important factor:
  - Each key has a **feature dimensionality** of **`embed_dim / num_heads`**.
  - Since our **input sequence length is 64**, a reasonable key size is **16 or 32**.
  - **Too small key sizes** can restrict the flexibility of the attention mechanism.
- **Observation:** More than **8 heads** do not significantly improve performance.
- **Best choice:**
  - **Embedding dimension:** `256`
  - **Feedforward hidden dimension:** `512` (**usually 2-4× the embedding size**)

26.5.5. Learning Rate & Regularization

- **Transformers typically require small learning rates**, often around **3e-5** in research papers.
- Since CIFAR10 is a **smaller dataset** with a **relatively simple task**, we found that increasing the learning rate to **3e-4** still resulted in stable training.
- To **reduce overfitting**, we apply:
  - **Dropout:** `0.2`
  - **Data augmentation** (already used during training)

26.5.6. Exploring Hyperparameters

Feel free to **experiment with different hyperparameter values** by modifying the settings below.

Observation: The **Vision Transformer on CIFAR10 is not overly sensitive** to hyperparameter choices, making it **relatively robust** to minor changes.

```
model, results = train_model(model_kwargs={
                                'embed_dim': 256,
                                'hidden_dim': 512,
                                'num_heads': 8,
                                'num_layers': 6,
                                'patch_size': 4,
                                'num_channels': 3,
                                'num_patches': 64,
                                'num_classes': 10,
                                'dropout': 0.2
                            },
                            lr=3e-4)
print("ViT results", results)
```

```
INFO:pytorch_lightning.utilities.rank_zero:GPU available: False, used: False
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU
cores
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPUs
```

```
Found pretrained model at /content/saved_models/ViT.ckpt, loading...
```

```
INFO:pytorch_lightning.utilities.migration.utils:Lightning automatically
upgraded your loaded checkpoint from v1.6.4 to v2.5.0.post0. To apply the
upgrade to your files permanently ... [86 chars truncated]
```

```
Testing: |          | 0/? [00:00<?, ?it/s]
```

```
Testing: |          | 0/? [00:00<?, ?it/s]
```

```
ViT results {'test': 0.7713000178337097, 'val': 0.9789999723434448}
```

26.5.7. Vision Transformer vs. CNN Performance on CIFAR10

The **Vision Transformer** achieves a **validation and test accuracy of approximately 75%**. In comparison, almost **all CNN architectures** reach around **90% classification accuracy** on CIFAR10.

26.5.8. Key Observations

- **Performance Gap:** A **significant gap (~15%)** exists between Vision Transformers and CNNs when trained **from scratch** on CIFAR10.
- **Pretraining Matters:** Vision Transformers **excel on large-scale datasets** (e.g., **ImageNet**) but struggle when trained **from scratch** on small datasets.
- **CNN Inductive Bias:** CNNs benefit from **built-in spatial hierarchies**, which makes them **more efficient** at learning image representations with fewer parameters.
- **Transformers Require More Data:** The self-attention mechanism in Transformers **does not inherently capture spatial structure** like CNNs, making **pretraining or large datasets crucial** for strong performance.

26.5.9. Training Curve Comparison

To further understand the differences between **CNNs and Vision Transformers**, let's **analyze the training curves** using **TensorBoard** below:

```
TENSORBOARD_PATH = "/content/saved_models/tensorboards/"

os.makedirs(TENSORBOARD_PATH, exist_ok=True)
vit_dir = os.path.join(TENSORBOARD_PATH, "vit")
resnet_dir = os.path.join(TENSORBOARD_PATH, "resnet")

os.makedirs(vit_dir, exist_ok=True)
os.makedirs(resnet_dir, exist_ok=True)

GDRIVE_FILE_ID_VIT = "1vss1RS05LbkQWnJje0GDMKi0zTyLyjGv"
GDRIVE_FILE_ID_RESNET = "1nyUhrWO8K4j4nrlUZlesCYX0yUPtFzes"

TMP_VIT = "tmp_events_vit"
TMP_RESNET = "tmp_events_resnet"

FINAL_VIT_EVENT = "events.out.tfevents.vit"
FINAL_RESNET_EVENT = "events.out.tfevents.resnet"
```

```
gdown.download(f"https://drive.google.com/uc?id={GDRIVE_FILE_ID_VIT}",
               TMP_VIT,
               quiet=False)

gdown.download(f"https://drive.google.com/uc?id={GDRIVE_FILE_ID_RESNET}",
               TMP_RESNET,
               quiet=False)

os.rename(TMP_VIT, os.path.join(vit_dir, FINAL_VIT_EVENT))
os.rename(TMP_RESNET, os.path.join(resnet_dir, FINAL_RESNET_EVENT))
```

```
Downloading...
From: https://drive.google.com/uc?id=1vss1RS05LbkQWnJje0GDMKi0zTyLyjGv
To: /content/tmp_events_vit
100%|##########| 377k/377k [00:00<00:00, 45.7MB/s]
Downloading...
From: https://drive.google.com/uc?id=1nyUhrWO8K4j4nrlUZlesCYX0yUPtFzes
To: /content/tmp_events_resnet
100%|##########| 455k/455k [00:00<00:00, 77.0MB/s]
```

```
%load_ext tensorboard
%tensorboard --logdir /content/saved_models/tensorboards/
```

26.5.10. Comparing Vision Transformer and ResNet on CIFAR10

The **TensorBoard comparison** between a **Vision Transformer** and a **ResNet** trained on CIFAR10 highlights several key differences in their learning dynamics.

26.5.11. Key Observations from Training Curves

1. **ResNet Learns Faster:**
   - The **ResNet converges significantly faster** in the **first few iterations**.
   - It reaches the **best performance of the Vision Transformer within just 5 epochs (~2000 iterations)**.
2. **Transformer Training is Slow:**
   - While **ResNet's training loss** and **validation accuracy** follow a similar trend, the Vision Transformer struggles:
     - **Validation accuracy stagnates** after **10k iterations**, even though training loss is still **slowly decreasing**.
     - **Training accuracy still reaches nearly 100%**, indicating the model can fit the data but struggles to generalize.

26.5.12. Why Does This Happen?

This performance discrepancy can be explained by a concept we've encountered before: **Inductive Biases**.

CNN Inductive Biases

CNNs **encode prior knowledge** about images: - **Translation Invariance**: Convolutional filters are **shared across the image**, ensuring a consistent feature extraction process. - **Locality and Hierarchy**: - **Nearby pixels** are assumed to be more **related** than distant ones. - **Hierarchical feature learning** progressively builds up from **edges -> textures -> objects**.

Why Do Transformers Struggle?

- A **Vision Transformer does not** inherently understand **which pixels are close or far apart**.
- It must **learn these relationships from scratch**, relying solely on the **classification loss signal**.
- This is a **huge disadvantage for small datasets**, where **spatial structure is critical** for generalization.

26.5.13. Why Do Vision Transformers Work So Well on Large Datasets?

- Given **large enough datasets** or **pretraining**, Vision Transformers can **learn spatial relationships without explicit biases**.
- **Unlike CNNs, Transformers are more flexible**:
  - **Can model long-range dependencies easily** (every patch is equally connected).
  - **Outperform CNNs on large-scale datasets** (e.g., **ImageNet**), where pretraining compensates for the lack of inductive biases.

26.6. Testing Our Model

```
model = ViT.load_from_checkpoint(MODEL_PATH + '/ViT.ckpt')
model.to(device)
model.eval()

class_labels = ["airplane", "automobile", "bird", "cat", "deer", "dog", "frog", "horse", "ship", "truck"]

def visualize_predictions(model, loader, num_images=8):
    """
    Visualizes predictions made by the Vision Transformer on a batch of images.

    Args:
        model (torch.nn.Module): The trained Vision Transformer model.
        loader (torch.utils.data.DataLoader): DataLoader containing test/validation images.
        num_images (int): Number of images to visualize (default: 8).

    Description:
        - Fetches a batch of images from the DataLoader.
        - Passes the images through the model to get predictions.
        - Displays the images alongside their predicted and ground truth labels.

    Note:
        - Assumes `class_labels` is a list mapping class indices to label names.
        - Image normalization is reversed for correct visualization.
```

```
    """
    model.eval()
    images, labels = next(iter(loader))
    images, labels = images[:num_images], labels[:num_images]

    with torch.no_grad():
        outputs = model(images.to(device))
        preds = torch.argmax(outputs, dim=1).cpu()

    fig, axes = plt.subplots(1, num_images, figsize=(15, 5))
    for i, ax in enumerate(axes):
        img = images[i].permute(1, 2, 0).numpy()
        img = (img * np.array([0.247, 0.243, 0.261])) + np.array([0.491, 0.482, 0.446])
        img = np.clip(img, 0, 1)
        ax.imshow(img)
        ax.axis("off")
        ax.set_title(f"Pred: {class_labels[preds[i]]}\nGT: {class_labels[labels[i]]}")

    plt.show()
```

```
INFO:pytorch_lightning.utilities.migration.utils:Lightning automatically
upgraded your loaded checkpoint from v1.6.4 to v2.5.0.post0. To apply the
upgrade to your files permanently ... [86 chars truncated]
/usr/local/lib/python3.11/dist-packages/torch/utils/data/dataloader.py:617:
UserWarning: This DataLoader will create 4 worker processes in total. Our
suggested max number of worker ... [257 chars truncated]
  warnings.warn(
```

```
visualize_predictions(model, test_loader)
```



26.7. Homework (Optional)

1. **Patch-size study.** Train the ViT on CIFAR-10 with patch sizes 2, 4, 8. Report validation accuracy and total training time per configuration. Discuss the trade-off between sequence length and patches-per-object.
2. **CLS vs. GAP.** Replace the `[CLS]`-token classification head with global average pooling over patch tokens. Retrain and report the accuracy change. Which is easier to optimise?
3. **Positional encoding ablation.** Train three variants: learned positional embedding (baseline), sinusoidal (1D over patch index), and no positional encoding at all. Report validation accuracy for each.

4. **Data augmentation.** Add `RandAugment` to the training transforms. Retrain and report the accuracy gain. ViTs typically benefit more from augmentation than equivalent CNNs, do you observe the same?
5. **Pretrained ViT.** Load `google/vit-base-patch16-224` from Hugging Face, swap its classifier for CIFAR-10, and fine-tune for 5 epochs. Compare with your from-scratch ViT in accuracy and training time.
6. **Attention map visualisation.** For three test images, plot the attention map of the last transformer block averaged over heads. Discuss whether the attention focuses on plausible object regions.

Chapter 27

Superpixel Tokenization for ViTs

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1UWI7fQEifJE-wpLf3d-6B8B62GpwSDea?usp=sharing

Superpixel tokenization for Vision Transformer [21] s **aggregates semantically similar pixels into adaptive tokens**, reducing redundancy and enabling the model to focus on more meaningful image regions. This technique is explored in works like *Tokens-to-Token ViT: Training Vision Transformers from Scratch on ImageNet* (Yuan et al., 2021), which demonstrates that dynamically grouping pixels can improve both efficiency and representation quality.

Image is from the paper: A Spitting Image: Modular Superpixel Tokenization in Vision Transformers

In this paper, the authors address a **fundamental limitation of Vision Transformer (ViT)** architectures, which typically rely on a rigid grid-based tokenization strategy that is **independent of an image's semantic content**.

They propose a **modular superpixel tokenization approach** that decouples tokenization from feature extraction, diverging from conventional methods that treat these processes as an inseparable unit. By incorporating online content-aware tokenization and scale- and shape-invariant positional embeddings, they conduct experiments and ablations to compare their method against standard patch-based tokenization and randomized partitions.

27.1. Necessary Imports

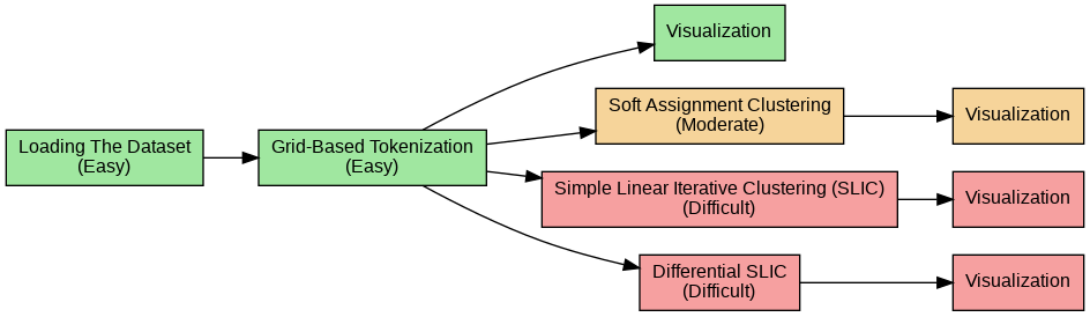
```
import os
import numpy as np
import random
import math
import matplotlib
import gdown
import time
import seaborn as sns
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.transforms.functional as TF
import torch.utils.data as data
import torch.optim as optim
import torchvision
from PIL import Image
from torchvision import transforms
from torchvision.datasets import CIFAR10
```

```
from IPython.display import Image as IMG
from torch.utils.data import random_split
from functools import partial
from skimage.segmentation import slic, mark_boundaries
from skimage.segmentation.slic_superpixels import _enforce_label_connectivity_cython
from tqdm.notebook import tqdm

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")

matplotlib.rcParams['lines.linewidth'] = 2.0  # Set the default line width for plots to 2.0 for better
   visibility
sns.reset_orig()                              # Reset Seaborn's settings to the original Matplotlib defaults
plt.set_cmap('cividis')                       # Set the default colormap for plots to 'cividis', which is
   colorblind-friendly
torch.backends.cudnn.deterministic = True     # Ensure deterministic behavior for CuDNN operations (useful for
   reproducibility)
torch.backends.cudnn.benchmark = False        # Disable CuDNN benchmarking to prevent adaptive optimizations
   that may affect reproducibility
```

```
<Figure size 640x480 with 0 Axes>
```



In this notebook, we will not be loading a Vision Transformer (ViT) model. Instead, the primary goal is to **demonstrate and compare different tokenization techniques** using the **CIFAR-10 [1] dataset**.

27.2. Loading Our Dataset

We load the **CIFAR-10 dataset** below.

The constants in `transforms.Normalize` are chosen to **scale and shift** the data so that it has a **zero mean** and a **standard deviation of one**.

```
mean = np.array([0.49139968, 0.48215841, 0.44653091])
std = np.array([0.24703223, 0.24348513, 0.26158784])

train_transform = transforms.Compose([transforms.ToTensor(),
                                      transforms.Normalize(mean, std)
                                     ])

train_dataset = CIFAR10(root='/content/data', train=True, transform=train_transform, download=True)

train_set, _ = torch.utils.data.random_split(train_dataset, [45000, 5000])

train_loader = data.DataLoader(train_set, batch_size=128, shuffle=True, drop_last=True, pin_memory=True,
   num_workers=2)
```

```
Files already downloaded and verified
```

27.3. Why superpixels? Grid patches vs. semantic regions

The standard Vision Transformer splits every image into an `NxN` grid of equal-size square patches. That is algorithmically clean and fast, but it has two drawbacks for classification.

First, a patch boundary cuts through object boundaries at arbitrary positions. A single patch may contain both the edge of a cat and a slice of background, forcing downstream attention heads to disentangle signals that do not need to be mixed. On a 32x32 image with 4x4 patches the effect is mild. On a 224x224 image with 16x16 patches it is often significant.

Second, the grid is uninformative about *content*. Every image gets the same tokenisation regardless of what is in it. If three of the 64 tokens cover the foreground object and 61 cover the background, the network has to implicitly learn to ignore the background tokens, but it sees 61 of them per image.

Superpixel tokenisation replaces the rigid grid with an adaptive partition that follows low-level image structure. A superpixel is a contiguous region of similar colour and brightness. Each superpixel becomes one token, so objects tend to be covered by a few semantically coherent tokens and background by larger, fewer tokens. The count and shape vary from image to image, which is the whole point.

The rest of this chapter walks through three increasingly sophisticated superpixel methods: a simple soft-assignment clustering, SLIC (Simple Linear Iterative Clustering), and a differentiable variant (`DiffSLIC`) that can be trained end-to-end alongside the classifier. The trade-off is always the same: more faithful tokenisation requires more computation and a more complex gradient path.

27.4. Transformers for Image Classification

Transformers were originally designed to **process sets**, as they are **permutation-equivariant architectures**, meaning that if the **input order is permuted**, the **output is permuted accordingly**. When applying Transformers to **sequential data**, we introduced **positional encodings** so the model could **learn** the importance of order.

However, traditional **image tokenization** for Vision Transformers (ViT) follows a **naïve, static grid-based approach**, dividing an image into **fixed-size patches** without considering the **content, context, or structure** of the image itself. This differs significantly from **NLP tokenization methods**, such as **Byte Pair Encoding (BPE)**, which dynamically segment words based on **frequency and semantics**.

27.4.1. Limitations

1. **Context-agnostic**: The same **grid-based** patching is applied to all images, **ignoring content boundaries**.
2. **Rigid structure**: Unlike **adaptive tokenization in NLP**, ViT patches **do not adjust** based on the underlying image structure.
3. **Loss of fine-grained details**: Important visual features might be **split across multiple patches**, leading to **inefficiencies** in learning.

27.4.2. Smarter Tokenization

To overcome the **static nature** of fixed patch tokenization, researchers have explored **adaptive tokenization** techniques, such as: - **Superpixel-based tokenization**: Dynamically defining patches based on image content. - **Region-aware tokenization**: Assigning tokens based on detected objects or salient regions. - **Hierarchical tokenization**: Using a coarse-to-fine representation to refine image patches progressively.

The **naïve** tokenization technique is **utilizing a grid-based approach**, splitting an image of size $N \times N$ into $(N/M)^2$ **patches** of size $M \times M$, which will serve as the **input tokens** for a Transformer.

Create a Denormalize Function for our Images

```
def denormalize_image(img_tensor, mean, std):
    """
    Denormalizes a normalized image tensor.

    Args:
        img_tensor (torch.Tensor): Normalized image tensor of shape (C, H, W).
        mean (list or tuple): Mean values used for normalization.
        std (list or tuple): Std values used for normalization.

    Returns:
        torch.Tensor: Denormalized image tensor in range [0, 1].
    """
    mean = torch.tensor(mean).view(-1, 1, 1)
    std = torch.tensor(std).view(-1, 1, 1)
    denorm = img_tensor * std + mean
    denorm = torch.clamp(denorm, 0, 1)
    return denorm
```

27.5. Grid-Based Patching Function

```
def img_to_patch(x, patch_size, flatten_channels=True):
    """
    Inputs:
        x - torch.Tensor of shape [B, C, H, W]
        patch_size - integer (size of each patch edge)
        flatten_channels - bool, if True => shape [B, H'*W', C*pH*pW]
                           if False => shape [B, H'*W', C, pH, pW]
    """
    # For an image shaped [1, 3, 32, 32]
    B, C, H, W = x.shape
    # x will be [1, 3, 8, 4, 8, 4]
    x = x.reshape(B, C, H//patch_size, patch_size, W//patch_size, patch_size)
    # x will be [1, 8, 8, 3, 4, 4]
    x = x.permute(0, 2, 4, 1, 3, 5)  # => [B, H', W', C, pH, pW]
    # x will be [1, 64, 3, 4, 4]
    x = x.flatten(1, 2)              # => [B, H'*W', C, pH, pW]
    if flatten_channels:
        # x will be [1, 64, 48], 64 patches with each having a feature vector of size 48
        x = x.flatten(2, 4)          # => [B, H'*W', C*pH*pW]
    return x
```

Let's take a look at how this works for our **CIFAR examples**.

For images of size 32×32, we choose a **patch size of 4**. This results in **sequences of 64 patches** (4×4 each).

```
def patches_to_mosaic(patches):
    """
    Convert patches of shape [B, N, C, pH, pW]
    into an [B, C, sqrt(N), sqrt(N)] mosaic,
    where each patch is a single 'tile' (an averaged color).
    """
    # From last example: [1, 64, 3, 4, 4]
    B, N, C, pH, pW = patches.shape
    # Number of patches along each dimension, e.g. 8 for CIFAR-10 with patch_size=4 (must be square)
    grid_size = int(N**0.5)

    # Average each patch's spatial dimension to get a single color for every channel: shape [1, 64, 3]
    patches_mean = patches.mean(dim=(3, 4))

    # Reshape to [1, 8, 8, 3]
    patches_mean = patches_mean.view(B, grid_size, grid_size, C)

    # Permute to [1, 3, 8, 8] to get a standard image shape
    mosaic = patches_mean.permute(0, 3, 1, 2).contiguous()
    return mosaic
```

```
num_images = 4

cifar_images = torch.stack([train_set[idx][0] for idx in range(num_images)], dim=0)  # [4, 3, 32, 32] (batched
   images)
img_patches = img_to_patch(cifar_images, patch_size=4, flatten_channels=False)  # [4, 64, 3, 4, 4]

mosaic = patches_to_mosaic(img_patches)  # [4, 3, 8, 8]

orig_grid = torchvision.utils.make_grid(cifar_images, nrow=4, normalize=True, pad_value=0.9).permute(1, 2, 0)
print("Original Grid Size: ", orig_grid.shape) # [3, 36, 138], without padding: [3, 32, 128] ([3, 32, 32 * 4
   images])
print("Original Grid Width/Height ratio: ", orig_grid.shape[0] / orig_grid.shape[1])

tokenized_grid = torchvision.utils.make_grid(mosaic, nrow=4, normalize=True, pad_value=0.9)
tokenized_grid = F.pad(tokenized_grid, pad=(2, 2, 0, 0), mode='constant', value=0.9).permute(1, 2, 0) # Adds p
   = 2 to the left and right of the grid
print("\nTokenized Grid Size: ", tokenized_grid.shape) # [3, 12, 42], without padding: [3, 8, 32] ([3, 8, 8 *
   4 images])
print("Tokenized Grid Width/Height ratio: ", tokenized_grid.shape[0] / tokenized_grid.shape[1])
```

```
Original Grid Size:  torch.Size([36, 138, 3])
Original Grid Width/Height ratio:  0.2608695652173913

Tokenized Grid Size:  torch.Size([12, 46, 3])
Tokenized Grid Width/Height ratio:  0.2608695652173913
```

As we can see, the images have been divided into 8×8 **miniature patches**, making them **indistinguishable** to the human eye. For a dataset like **CIFAR-10**, which already has a low resolution (32×32 **pixels**), this **naïve tokenization method severely reduces meaningful context**. Each patch effectively retains only the **average color** of the pixels within its region, resulting in a significant **loss of fine-grained details and structural information** from the original image.

```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(8, 4))
ax1.imshow(orig_grid.cpu().numpy())
ax1.axis('off')
ax1.set_title("Original Images")

ax2.imshow(tokenized_grid.cpu().numpy())
ax2.axis('off')
ax2.set_title("Grid-Based Tokenization")

plt.tight_layout()
plt.show()
```



Let's also plot the **borders** of our patches.

```
def patches_to_mosaic_with_borders(patches, border=1, border_value=0.0):
    """
    Reassembles patches into a mosaic image with borders drawn between patches.

    Args:
        patches (Tensor): Tensor of shape [B, num_patches, C, pH, pW] produced by img_to_patch.
        border (int): Thickness of the border in pixels.
        border_value (float): The pixel value for the border (e.g., 0.0 for black).

    Returns:
        Tensor: Reconstructed mosaic image of shape [B, C, H_total, W_total].
    """
    B, N, C, pH, pW = patches.shape
    num_rows = int(math.sqrt(N))
    num_cols = num_rows
    if num_rows * num_cols != N:
        raise ValueError("Number of patches must be a perfect square.")

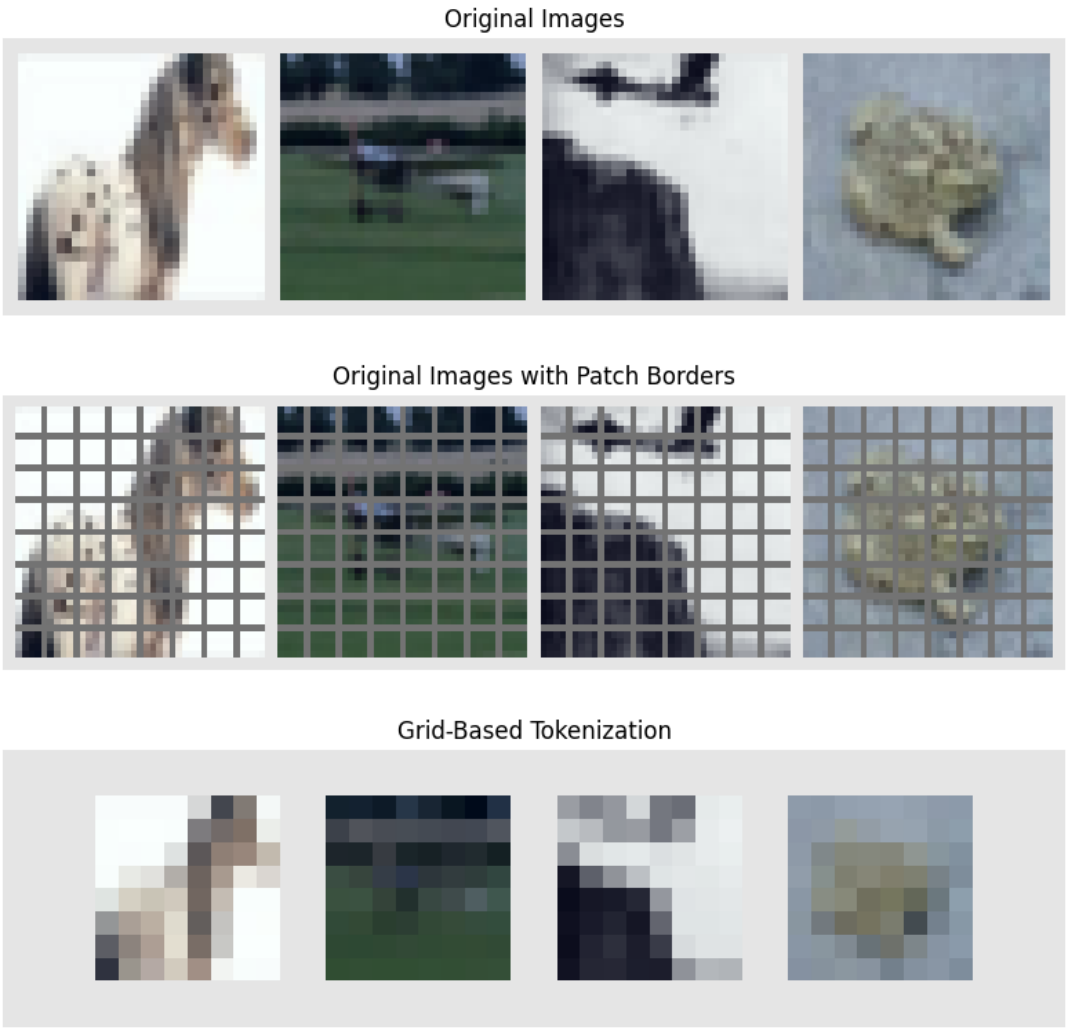
    mosaic_list = []
    for b in range(B):
```

```
        # Reshape patches to [num_rows, num_cols, C, pH, pW]
        patch_grid = patches[b].reshape(num_rows, num_cols, C, pH, pW)
        # Compute the overall dimensions of the mosaic image.
        H_total = num_rows * pH + (num_rows - 1) * border
        W_total = num_cols * pW + (num_cols - 1) * border
        # Create an empty canvas filled with border_value.
        mosaic_img = torch.full((C, H_total, W_total), border_value, dtype=patches.dtype, device=patches.
  device)
        for i in range(num_rows):
            for j in range(num_cols):
                y_start = i * (pH + border)
                y_end = y_start + pH
                x_start = j * (pW + border)
                x_end = x_start + pW
                mosaic_img[:, y_start:y_end, x_start:x_end] = patch_grid[i, j]
        mosaic_list.append(mosaic_img)
    return torch.stack(mosaic_list, dim=0)
```

```
mosaic_with_borders = patches_to_mosaic_with_borders(img_patches, border=1, border_value=0.0)
tokenized_grid_with_borders = torchvision.utils.make_grid(mosaic_with_borders, nrow=4, normalize=True,
  pad_value=0.9).permute(1, 2, 0)

fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(8, 8))
ax1.imshow(orig_grid.cpu().numpy())
ax1.axis("off")
ax1.set_title("Original Images")
ax2.imshow(tokenized_grid_with_borders.cpu().numpy())
ax2.axis("off")
ax2.set_title("Original Images with Patch Borders")
ax3.imshow(tokenized_grid.cpu().numpy())
ax3.axis("off")
ax3.set_title("Grid-Based Tokenization")

plt.tight_layout()
plt.show()
```



Let's Test our Tokenization Method on a Larger Image

First, let's download an image of my cat!

```
file_id = '17w9DQbx0crjke4yBY6_4eQcOy1L88qI1'
output_path = 'cat.png'

if not os.path.exists('/content/cat.png'):
  gdown.download(id=file_id, output=output_path, quiet=True)
else:
  print("cat.png exists")
```

```
cat.png exists
```

```
img = Image.open('cat.png').convert('RGB')

transform = transforms.Compose([
    transforms.Resize((1024, 1024)),  # Resize from (3024, 3024)
    transforms.ToTensor(),
    transforms.Normalize(mean, std)
])
```

```
cat_img_tensor = transform(img)
cat_img_tensor = cat_img_tensor.unsqueeze(0)

img_patches = img_to_patch(cat_img_tensor, patch_size=32, flatten_channels=False)
cat_mosaic = patches_to_mosaic(img_patches)
cat_mosaic_with_borders = patches_to_mosaic_with_borders(img_patches, border=1, border_value=0.0)

print("Cat Image Shape: ", cat_img_tensor.shape)
print("Cat Mosaic Shape: ", mosaic.shape)
print("Cat Mosaic with Borders Shape", mosaic_with_borders.shape)

cat_orig_grid = torchvision.utils.make_grid(cat_img_tensor, nrow=1, normalize=True, pad_value=0.9).permute(1,
  2, 0)
cat_tokenized_grid = torchvision.utils.make_grid(cat_mosaic, nrow=1, normalize=True, pad_value=0.9).permute(1,
  2, 0)
cat_tokenized_grid_borders = torchvision.utils.make_grid(cat_mosaic_with_borders, nrow=1, normalize=True,
  pad_value=0.9).permute(1, 2, 0)

print("\nOriginal Grid Size: ", cat_orig_grid.shape)
print("Mosaic Grid Size: ", cat_tokenized_grid.shape)
print("Mosaic Grid with Borders Size: ", cat_tokenized_grid_borders.shape)
```

```
Cat Image Shape:  torch.Size([1, 3, 1024, 1024])
Cat Mosaic Shape:  torch.Size([4, 3, 8, 8])
Cat Mosaic with Borders Shape torch.Size([4, 3, 39, 39])

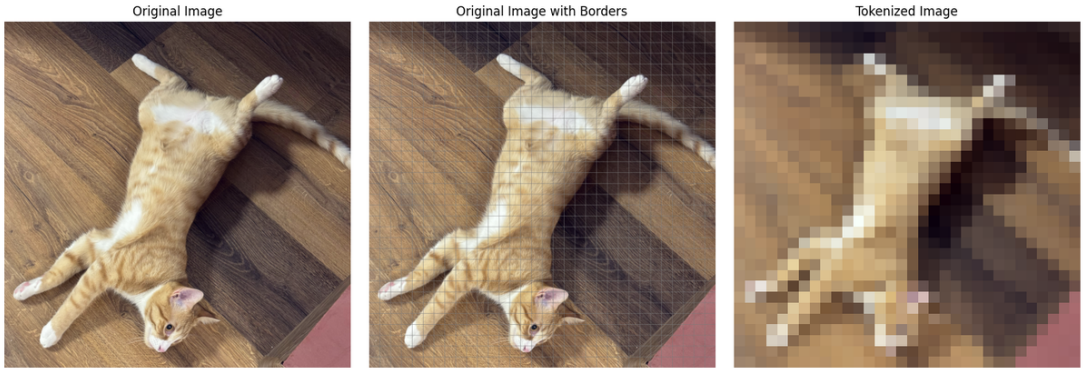
Original Grid Size:  torch.Size([1024, 1024, 3])
Mosaic Grid Size:  torch.Size([32, 32, 3])
Mosaic Grid with Borders Size:  torch.Size([1055, 1055, 3])
```

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(15, 5))
ax1.imshow(cat_orig_grid.cpu().numpy())
ax1.axis('off')
ax1.set_title("Original Image")

ax3.imshow(cat_tokenized_grid.cpu().numpy())
ax3.axis('off')
ax3.set_title("Grid-Based Tokenization")

ax2.imshow(cat_tokenized_grid_borders.cpu().numpy())
ax2.axis('off')
ax2.set_title("Original Image with Patch Borders")

plt.tight_layout()
plt.show()
```



27.6. Soft Assignment Clustering for Superpixel Tokenization

This implementation uses a **differentiable clustering algorithm** to compute soft assignments of pixels to a fixed number of clusters (i.e., superpixels).

In each iteration, the algorithm computes the **Euclidean distance between pixel features** (augmented with spatial coordinates) and cluster centers, then applies a softmax function to obtain a probability distribution over clusters **for every pixel**.

The cluster centers are updated as the **weighted average of pixel features** based on these soft assignments, and after several iterations, the resulting centers, after discarding the coordinate information, represent the aggregated color information for each superpixel.

This **differentiable clustering algorithm** is considered **differentiable** because every operation, such as the `softmax` for computing soft assignments, the `norm` computations, and the `weighted averaging` (via matrix multiplications), is implemented using standard PyTorch operations that are **end-to-end differentiable**. This means gradients can flow through the clustering process, allowing the superpixel tokenization to be **integrated into larger networks** and optimized jointly with downstream losses.

```
def img_to_superpixel_tokens(x, num_superpixels, num_iter=10, lambda_coord=0.1):
    """
    Converts images to superpixel tokens using a simple differentiable clustering algorithm.

    Args:
        x (Tensor): Input images of shape [B, C, H, W].
        num_superpixels (int): Desired number of superpixels (tokens) per image.
        num_iter (int): Number of iterations for the soft clustering.
        lambda_coord (float): Weight for the coordinate features relative to pixel features.

    Returns:
        tokens (Tensor): Superpixel tokens of shape [B, num_superpixels, C] (the aggregated RGB features).
        assignment (Tensor): Soft assignments of shape [B, num_superpixels, H*W] (for inspection if needed).
    """
    B, C, H, W = x.shape # [1, 3, 32, 32]
    N = H * W # 32 * 32 = 1024
    x_flat = x.view(B, C, N) # [1, 3, 1024]

    # Create coordinate grid for pixels, normalized by image size.
    ys = torch.linspace(0, H - 1, H, device=x.device) / H # generates a 1D tensor with H evenly spaced values
  from 0 to H-1.
    xs = torch.linspace(0, W - 1, W, device=x.device) / W
    grid_y, grid_x = torch.meshgrid(ys, xs, indexing='ij') # takes one-dimensional coordinate tensors and
  produces coordinate grids (i.e., multi-dimensional tensors) for each of the input vectors
    coords = torch.stack([grid_y, grid_x], dim=0).view(2, N)  # [2, 1024]
    coords = coords.unsqueeze(0).expand(B, -1, -1)  # Adds batch dimension [1, 2, 1024]

    # Combine pixel features with (weighted) coordinates: [B, C+2, N]
    features = torch.cat([x_flat, lambda_coord * coords], dim=1) # [1, 2+3, 1024]
    D = C + 2

    # Initialize cluster centers: sample num_superpixels pixels from each image (randomly)
```

```
    indices = torch.randint(0, N, (B, num_superpixels), device=x.device) # for 5 superpixels and a batch of 5
  images -> 25 indices
    centers = torch.stack([features[b, :, indices[b]] for b in range(B)], dim=0)  # [B, D, num_superpixels] ->
  [1, 5, 5]

    # Iteratively refine centers using soft-assignment (differentiable analog of k-means)
    # add tqdm to this
    for i in range(num_iter):
        print(f"Refining centers using soft-assignment at step: {i}")
        # Compute Euclidean distances between pixel features and centers.
        diff = features.unsqueeze(2) - centers.unsqueeze(3)  # [B, D, num_superpixels, N] -> [1, 5, 5, 1024]
        dist = torch.norm(diff, dim=1)  # [B, num_superpixels, N] -> [1, 5, 1024]

        # Compute soft assignments: higher weight to closer centers.
        assignment = F.softmax(-dist, dim=1)  # [B, num_superpixels, N] -> [1, 5, 1024]

        # Update centers as weighted average of features.
        centers = torch.matmul(assignment, features.transpose(1,2)).transpose(1,2)  # [B, D, num_superpixels]
  -> [1, 5, 5]
        # Adjust centers by dividing by the sum of assignments.
        # Transpose centers to shape [B, num_superpixels, D] to align with assignment sum.
        centers = (centers.transpose(1,2) / (assignment.sum(dim=2, keepdim=True) + 1e-8)).transpose(1,2)

    # Drop the coordinate part; keep only the RGB channels.
    tokens = centers[:, :C, :].transpose(1, 2)  # [B, num_superpixels, C] -> [1, 5, 3]
    return tokens, assignment
```

```
ys = torch.linspace(0, 4 - 1, 4, device=device) / 4
xs = torch.linspace(0, 4 - 1, 4, device=device) / 4
grid_y, grid_x = torch.meshgrid(ys, xs, indexing='ij')
print("1D Tensors: \n", xs, ys)
print("Meshgrid from xs: \n", grid_x)
print("Meshgrid from ys: \n", grid_y)
```

```
1D Tensors:
 tensor([0.0000, 0.2500, 0.5000, 0.7500]) tensor([0.0000, 0.2500, 0.5000,
0.7500])
Meshgrid from xs:
 tensor([[0.0000, 0.2500, 0.5000, 0.7500],
        [0.0000, 0.2500, 0.5000, 0.7500],
        [0.0000, 0.2500, 0.5000, 0.7500],
        [0.0000, 0.2500, 0.5000, 0.7500]])
Meshgrid from ys:
 tensor([[0.0000, 0.0000, 0.0000, 0.0000],
        [0.2500, 0.2500, 0.2500, 0.2500],
        [0.5000, 0.5000, 0.5000, 0.5000],
        [0.7500, 0.7500, 0.7500, 0.7500]])
```

```
coords = torch.stack([grid_y, grid_x], dim=0).view(2, 16)
print("Coords: \n", coords)
```

```
Coords:
 tensor([[0.0000, 0.0000, 0.0000, 0.0000, 0.2500, 0.2500, 0.2500, 0.2500,
0.5000,
         0.5000, 0.5000, 0.5000, 0.7500, 0.7500, 0.7500, 0.7500],
        [0.0000, 0.2500, 0.5000, 0.7500, 0.0000, 0.2500, 0.5000, 0.7500, 0.0000,
         0.2500, 0.5000, 0.7500, 0.0000, 0.2500, 0.5000, 0.7500]])
```

The idea behind adding the weighted coordinates is to embed spatial information into each pixel's feature vector so that the clustering takes into account **not just color similarity but also the pixels' relative positions**. In this implementation, each pixel is represented by a feature vector of length $C + 2$ (with C being the number of color channels, usually 3 for RGB, and the additional 2 dimensions for the normalized y and x coordinates). The coordinate values are scaled by `lambda_coord` to control how much influence spatial proximity has compared to color differences.

When we initialize the cluster centers and update them, we compute the **Euclidean distance** in this combined feature space. This means that a pixel will be **more likely assigned to a cluster if it is both similar in color and nearby spatially**. Even though the final cluster centers (tokens) are then reduced to just the RGB channels (by dropping the coordinate part), **the spatial arrangement has already been enforced during clustering**. The resulting tensor of shape `[B, num_superpixels, C]` means that for each image in the batch, we have `num_superpixels` tokens, where each token is a representative RGB color of that superpixel.

1. **Feature Preparation:**
   Each pixel in the image is represented by a feature vector that concatenates its RGB color values (3 numbers) with its normalized spatial coordinates (2 numbers), resulting in a 5-dimensional feature per pixel. The image, originally of shape `[B, C, H, W]`, is flattened into `[B, C, N]` where `N = H*W`. The spatial grid is computed and scaled (dividing by H and W) to give each pixel a (y, x) coordinate, which is then concatenated with the RGB values (after being weighted by `lambda_coord`).
2. **Cluster Center Initialization:**
   We randomly select a fixed number (`num_superpixels`) of pixels from each image to serve as initial cluster centers. Each center is a 5-dimensional vector (3 for color and 2 for coordinates), and we gather these into a tensor of shape `[B, D, num_superpixels]` where `D = C + 2`.
3. **Iterative Refinement:**
   For a fixed number of iterations (`num_iter`):
   - **Distance Computation:**
     We compute the Euclidean distance between each pixel's feature vector and each cluster center. This is done by subtracting each center from every pixel feature, yielding a tensor of differences with shape `[B, D, num_superpixels, N]`, then taking the L2 norm over the feature dimension (dimension 1) to get distances of shape `[B, num_superpixels, N]`.
   - **Soft Assignment:**
     We apply a softmax to the negative distances (so that smaller distances yield higher probabilities) along the cluster dimension. This gives us a soft assignment matrix of shape `[B, num_superpixels, N]`, where each pixel has a probability of belonging to each cluster.
   - **Center Update:**
     Each cluster center is then updated as the weighted average of all pixel features, using the soft assignments as weights. This is computed with a matrix multiplication that produces new centers, which are then normalized by dividing by the sum of assignments for each cluster.
4. **Final Token Extraction:**
   After the iterative refinement, we discard the coordinate components from each cluster center and

keep only the RGB values, resulting in the final superpixel tokens of shape `[B, num_superpixels, C]`.

Visualizing Superpixels

```
def build_mosaic(x, tokens, assignment):
    """
    Builds a mosaic image from superpixel tokens and returns the mosaic along with the pixel-to-cluster map.

    Args:
        x (Tensor): Original image tensor of shape [B, C, H, W].
        tokens (Tensor): Superpixel tokens of shape [B, K, C].
        assignment (Tensor): Soft assignments of shape [B, K, H*W].

    Returns:
        mosaic (Tensor): Mosaic image of shape [B, C, H, W], where each pixel is assigned the color of its
   cluster.
        cluster_ids (Tensor): Hard assignment map of shape [B, H, W] with cluster IDs.
    """
    B, C, H, W = x.shape
    # Compute hard assignments: each pixel gets the cluster with maximum soft-assignment.
    cluster_ids = torch.argmax(assignment, dim=1).view(B, H, W)
    # Build mosaic: for each image, each pixel takes the color of its assigned token.
    mosaic = torch.stack([tokens[b][cluster_ids[b]].permute(2, 0, 1) for b in range(B)], dim=0)
    return mosaic, cluster_ids

def superpixel_mosaic(x, tokens, assignment):
    """
    Constructs a mosaic image from superpixel tokens.

    Args:
        x (Tensor): Original image tensor [B, C, H, W].
        tokens (Tensor): Superpixel tokens [B, K, C].
        assignment (Tensor): Soft assignments [B, K, H*W].

    Returns:
        mosaic (Tensor): Mosaic image of shape [B, C, H, W].
    """
    mosaic, _ = build_mosaic(x, tokens, assignment)
    return mosaic

def superpixel_border_mosaic(x, tokens, assignment, border_width = 1, border_value = 0.0):
    """
    Constructs a mosaic image from superpixel tokens, highlighting cluster boundaries.

    Args:
        x (Tensor): Original image tensor [B, C, H, W].
        tokens (Tensor): Superpixel tokens [B, K, C].
        assignment (Tensor): Soft assignments [B, K, H*W].
        border_width (int): Thickness of the boundary in pixels.
        border_value (float): Pixel value for boundary (e.g., 0.0 for black).

    Returns:
        mosaic_with_borders (Tensor): Mosaic image with superpixel boundaries, shape [B, C, H, W].
    """
    mosaic, cluster_ids = build_mosaic(x, tokens, assignment)
    mosaic_with_borders = mosaic.clone()
    B, C, H, W = x.shape
```

```
    for b in range(B):
        cid_map = cluster_ids[b]  # [H, W]
        boundary_mask = torch.zeros((H, W), dtype=torch.bool, device=x.device)
        # Check differences with right neighbors
        diff_right = (cid_map[:, :-1] != cid_map[:, 1:])
        boundary_mask[:, :-1] |= diff_right
        boundary_mask[:, 1:] |= diff_right
        # Check differences with bottom neighbors
        diff_down = (cid_map[:-1, :] != cid_map[1:, :])
        boundary_mask[:-1, :] |= diff_down
        boundary_mask[1:, :] |= diff_down

        if border_width > 1:
            # For thicker boundaries, use a simple dilation
            from torch.nn.functional import conv2d
            kernel_size = 2 * border_width - 1
            kernel = torch.ones((1, 1, kernel_size, kernel_size), device=x.device, dtype=torch.float)
            mask_4d = boundary_mask.unsqueeze(0).unsqueeze(0).float()
            dilated = conv2d(mask_4d, kernel, padding=(border_width - 1, border_width - 1))
            boundary_mask = (dilated.squeeze(0).squeeze(0) > 0).bool()

        # Paint the boundary pixels with border_value
        mosaic_with_borders[b][:, boundary_mask] = border_value

    return mosaic_with_borders
```

```
num_superpixels = 32 # Try setting different values
tokens, assignment = img_to_superpixel_tokens(cifar_images, num_superpixels, num_iter=15, lambda_coord=15) #
  Try setting different values for lambda_coord

superpixel_images = superpixel_mosaic(cifar_images, tokens, assignment)
superpixel_with_borders = superpixel_border_mosaic(cifar_images, tokens, assignment, border_width=1,
  border_value=0.0)

superpixel_grid = torchvision.utils.make_grid(superpixel_images, nrow=4, normalize=True, pad_value=0.9).
  permute(1, 2, 0)
superpixel_bordered_grid = torchvision.utils.make_grid(superpixel_with_borders, nrow=4, normalize=True,
  pad_value=0.9).permute(1, 2, 0)
```

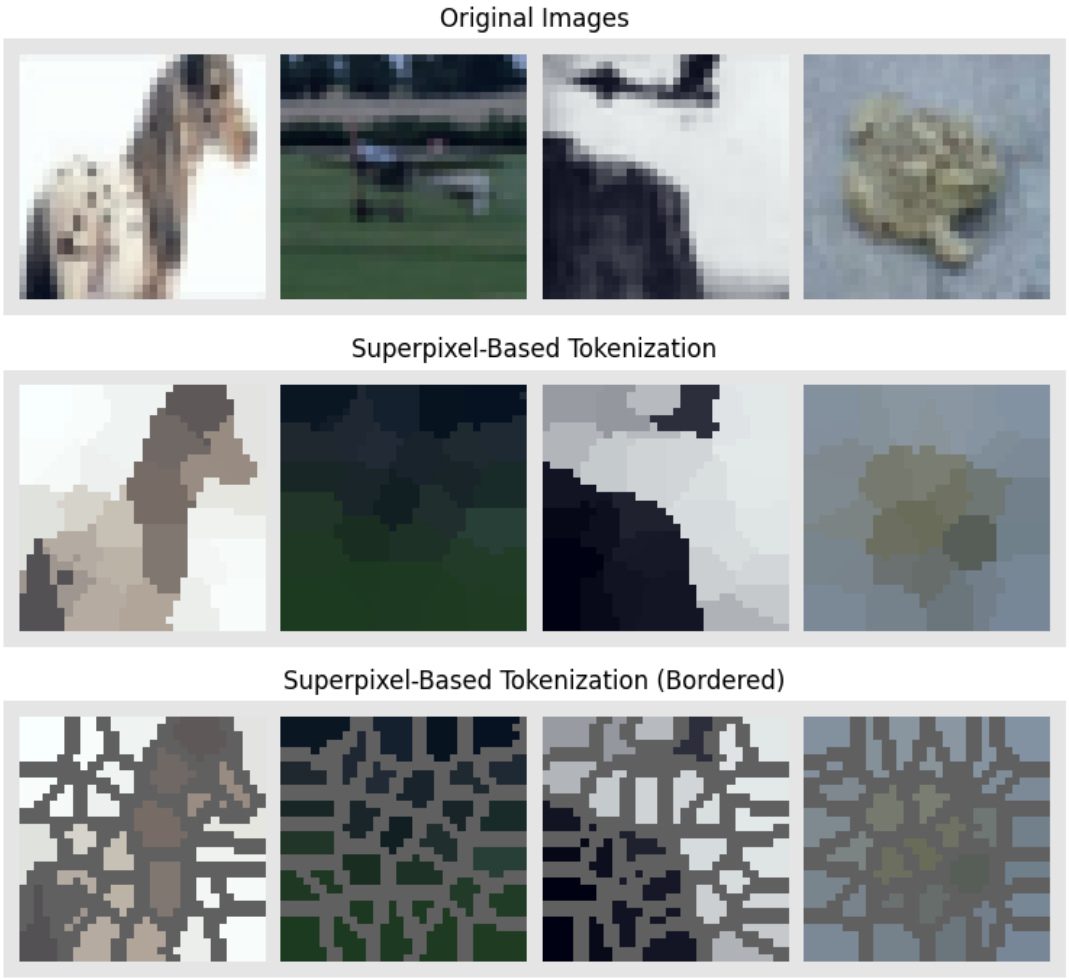
```
Refining centers using soft-assignment at step: 0
Refining centers using soft-assignment at step: 1
Refining centers using soft-assignment at step: 2
Refining centers using soft-assignment at step: 3
Refining centers using soft-assignment at step: 4
Refining centers using soft-assignment at step: 5
Refining centers using soft-assignment at step: 6
Refining centers using soft-assignment at step: 7
Refining centers using soft-assignment at step: 8
Refining centers using soft-assignment at step: 9
Refining centers using soft-assignment at step: 10
Refining centers using soft-assignment at step: 11
Refining centers using soft-assignment at step: 12
Refining centers using soft-assignment at step: 13
Refining centers using soft-assignment at step: 14
```

```
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(24, 7))
ax1.imshow(orig_grid.cpu().numpy())
ax1.axis('off')
ax1.set_title("Original Images")
```

```
ax2.imshow(superpixel_grid.cpu().numpy())
ax2.axis('off')
ax2.set_title("Superpixel-Based Tokenization")

ax3.imshow(superpixel_bordered_grid.cpu().numpy())
ax3.axis('off')
ax3.set_title("Superpixel-Based Tokenization (Bordered)")

plt.tight_layout()
plt.show()
```



Let's Try it with the Larger Cat Image

```
num_superpixels = 64
start_time = time.time()
tokens, assignment = img_to_superpixel_tokens(cat_img_tensor, num_superpixels, num_iter=15, lambda_coord=30) #
  Try setting different values for lambda_coord
elapsed_time = time.time() - start_time
print(f"Time taken: {elapsed_time:.4f} seconds")
```

```
Refining centers using soft-assignment at step: 0
Refining centers using soft-assignment at step: 1
Refining centers using soft-assignment at step: 2
```

```
Refining centers using soft-assignment at step: 3
Refining centers using soft-assignment at step: 4
Refining centers using soft-assignment at step: 5
Refining centers using soft-assignment at step: 6
Refining centers using soft-assignment at step: 7
Refining centers using soft-assignment at step: 8
Refining centers using soft-assignment at step: 9
Refining centers using soft-assignment at step: 10
Refining centers using soft-assignment at step: 11
Refining centers using soft-assignment at step: 12
Refining centers using soft-assignment at step: 13
Refining centers using soft-assignment at step: 14
```

```
cat_superpixel_images = superpixel_mosaic(cat_img_tensor, tokens, assignment)
cat_superpixel_with_borders = superpixel_border_mosaic(cat_img_tensor, tokens, assignment, border_width=1,
  border_value=0.0)

cat_superpixel_grid = torchvision.utils.make_grid(cat_superpixel_images, nrow=1, normalize=True, pad_value=0.
  9).permute(1, 2, 0)
cat_superpixel_bordered_grid = torchvision.utils.make_grid(cat_superpixel_with_borders, nrow=1,
  normalize=True, pad_value=0.9).permute(1, 2, 0)

print("\nOriginal Grid Size: ", cat_orig_grid.shape)
print("Mosaic Grid Size: ", cat_superpixel_grid.shape)
print("Mosaic Grid with Borders Size: ", cat_superpixel_bordered_grid.shape)
```

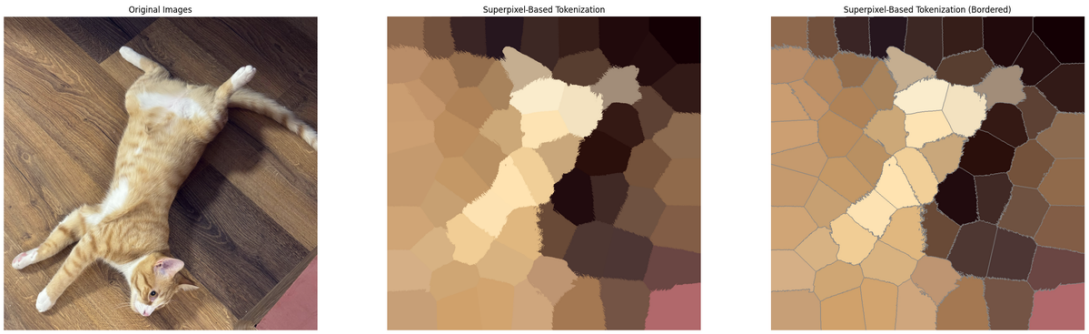
```
Original Grid Size:  torch.Size([1024, 1024, 3])
Mosaic Grid Size:  torch.Size([1024, 1024, 3])
Mosaic Grid with Borders Size:  torch.Size([1024, 1024, 3])
```

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(24, 7))
ax1.imshow(cat_orig_grid.cpu().numpy())
ax1.axis('off')
ax1.set_title("Original Images")

ax2.imshow(cat_superpixel_grid.cpu().numpy())
ax2.axis('off')
ax2.set_title("Superpixel-Based Tokenization")

ax3.imshow(cat_superpixel_bordered_grid.cpu().numpy())
ax3.axis('off')
ax3.set_title("Superpixel-Based Tokenization (Bordered)")

plt.tight_layout()
plt.show()
```



27.7. Simple Linear Iterative Clustering (SLIC)

The **SLIC (Simple Linear Iterative Clustering) algorithm** generates **superpixels** by clustering pixels based on both **color similarity** and **spatial proximity**. It follows an efficient and iterative approach to ensure **compact, well-distributed superpixels** while preserving object boundaries.

27.7.1. 1. Regular Grid Initialization

The algorithm begins by **placing cluster centers** on a **regular grid**, spaced approximately N/K **pixels apart** (an improvement over the previous method), where: - N is the total number of pixels in the image. - K is the desired number of superpixels.
This ensures an **even distribution** of clusters across the image.

27.7.2. 2. Local Search for Compact Clusters

Instead of considering the entire image, each **cluster center** only updates **pixels within a local search radius** of approximately N/K pixels.
This **local constraint** encourages: - **Compact superpixels** that do not merge distant pixels. - **Efficient computations**, as each cluster only interacts with a limited neighborhood (another improvement over the previous method).

27.7.3. 3. Distance Metric for Pixel-Cluster Association

To assign pixels to their nearest cluster, SLIC uses a **combined distance measure** that considers both **color similarity** and **spatial proximity**:

$$d(p, c) = \|\text{color}(p) - \text{color}(c)\|^2 + \left(\frac{m}{S}\right)^2 \|\text{coords}(p) - \text{coords}(c)\|^2$$

where: - **color**(p) **and color**(c): Color vectors of pixel p and cluster center c. - **coords**(p) **and coords**(c): Pixel and cluster coordinates, **normalized by image size**. - S: Approximate superpixel size ($\sqrt{N/K}$). - m: A **compactness parameter** controlling the trade-off between **color similarity** and **spatial proximity**.

A **higher** m **value** prioritizes **spatial smoothness**, leading to **more compact and evenly shaped superpixels**.

27.7.4. 4. Iterative Refinement for Precise Clustering

SLIC runs several **iterations** to refine superpixel assignments:

1. **Pixel Assignment**

- Each pixel is assigned to the **nearest cluster center** based on the **combined color + spatial distance** metric.
- A **global distance buffer** ensures that each pixel's assignment is **only updated if a closer cluster is found**.

2. **Cluster Center Update**
   - Each cluster center is updated as the **mean color and spatial coordinate** of all pixels assigned to it.
   - This ensures that cluster centers progressively **converge to the most representative locations**.

```python
def slic_superpixels(image_np, n_segments=64, compactness=15.0):
    """
    Applies SLIC superpixel segmentation to a single image (in NumPy form).

    Args:
        image_np (ndarray): The image in HxWxC format, range [0..1] or [0..255].
        n_segments (int): Number of superpixels.
        compactness (float): Balances color proximity and space proximity.
                             Higher values => more spatially compact clusters.

    Returns:
        labels (ndarray): Integer labels for each pixel indicating superpixel ID.
                          Shape [H, W].
    """
    # skimage expects float images in [0..1]
    labels = slic(image_np, n_segments=n_segments, compactness=compactness, start_label=0)
    return labels

def superpixel_average_color(image_np, labels):
    """
    Given an image (HxWxC) and superpixel labels (HxW), compute the average color
    for each superpixel and return a mosaic image where each superpixel is replaced
    by its average color.

    Args:
        image_np (ndarray): Image of shape [H, W, C].
        labels (ndarray): Superpixel labels of shape [H, W].

    Returns:
        mosaic_np (ndarray): Mosaic image with the same shape as image_np,
                             where each superpixel is replaced by its average color.
    """
    H, W, C = image_np.shape
    mosaic_np = np.zeros_like(image_np)

    max_label = labels.max()

    for sp_id in range(max_label + 1):
        # mask: True where the label is sp_id
        mask = (labels == sp_id)

        # average color
        avg_color = image_np[mask].mean(axis=0)
        mosaic_np[mask] = avg_color

    return mosaic_np
```

```
def superpixel_mosaic_torch(img_tensor, n_segments=64, compactness=15.0):
    """
    Given a single image in Torch tensor form [C, H, W], run SLIC superpixels,
    and return a mosaic image in NumPy form [H, W, C].

    Args:
        img_tensor (Tensor): shape [C, H, W], in [0..1] or [0..255] range.
        n_segments (int): Number of superpixels for SLIC.
        compactness (float): SLIC compactness factor.

    Returns:
        mosaic_np (ndarray): [H, W, C] mosaic with average superpixel colors.
        boundary_np (ndarray): [H, W, 3] if you want an image with boundaries marked.
    """
    # [H, W, C]
    img_np = img_tensor.permute(1, 2, 0).cpu().numpy()
    labels = slic_superpixels(img_np, n_segments=n_segments, compactness=compactness)
    mosaic_np = superpixel_average_color(img_np, labels)
    boundary_np = mark_boundaries(img_np, labels, color=(1, 0, 0), mode='outer')
    return mosaic_np, boundary_np
```

```
slic_list = []
boundary_list = []
for i in range(num_images):
    img_t = cifar_images[i]  # shape [3, 32, 32]
    slic_np, boundary_np = superpixel_mosaic_torch(
        img_t,
        n_segments=64,
        compactness=15.0
    )
    slic_list.append(slic_np)
    boundary_list.append(boundary_np)

slic_tensors = []
boundary_tensors = []
for mosaic_np, bound_np in zip(slic_list, boundary_list):
    # shape [H, W, C] => [C, H, W]
    mosaic_t = torch.from_numpy(mosaic_np).permute(2, 0, 1)
    bound_t = torch.from_numpy(bound_np).permute(2, 0, 1)

    mosaic_t = denormalize_image(mosaic_t, mean, std)
    bound_t = denormalize_image(bound_t, mean, std)

    # If needed, clamp to [0..1], convert to float, etc.
    mosaic_t = mosaic_t.float().clamp(0,1)
    bound_t = bound_t.float().clamp(0,1)

    slic_tensors.append(mosaic_t)
    boundary_tensors.append(bound_t)

slic_tensor_batch = torch.stack(slic_tensors, dim=0)
slic_grid = torchvision.utils.make_grid(slic_tensor_batch, nrow=4, normalize=True, pad_value=0.9).permute(1,
   2, 0)

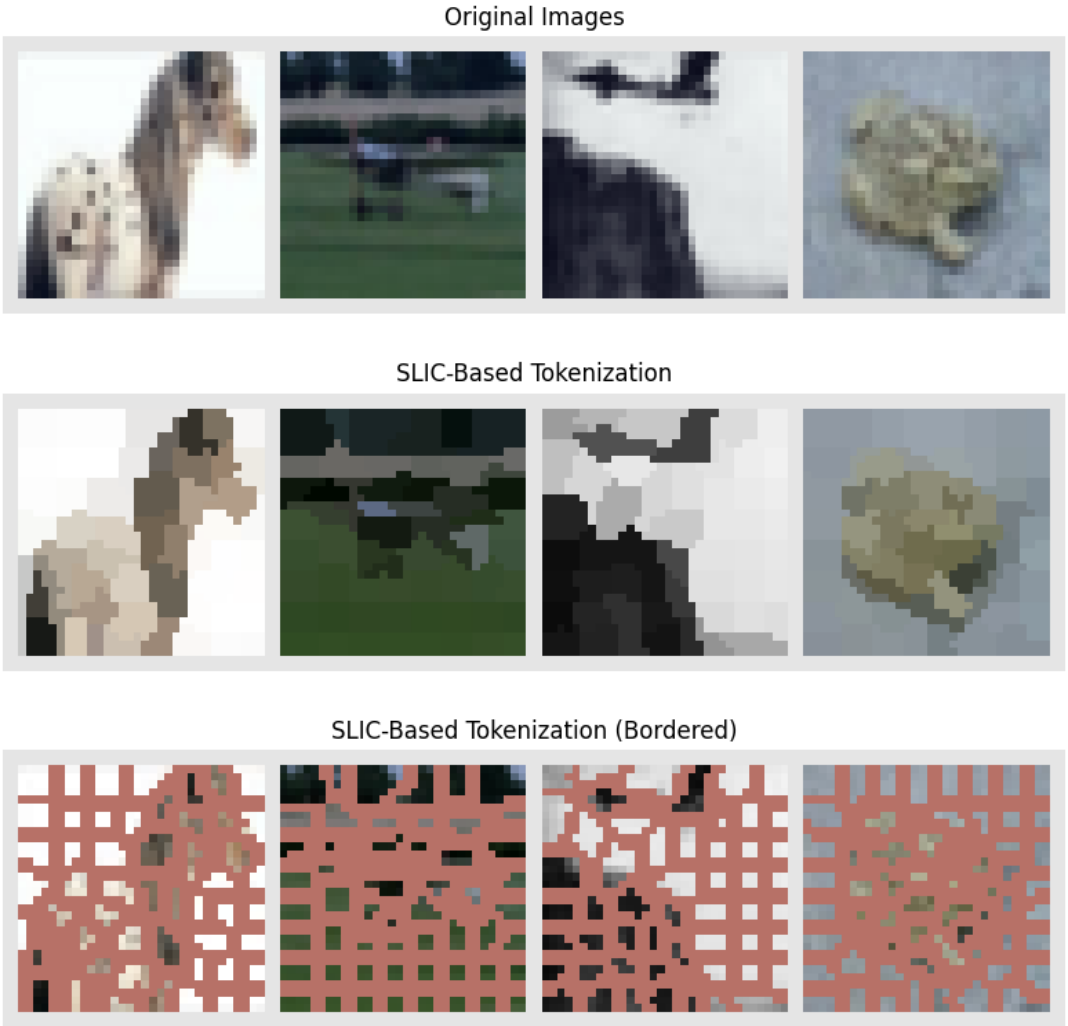
slic_boundary_tensor_batch = torch.stack(boundary_tensors, dim=0)
slic_boundary_grid = torchvision.utils.make_grid(slic_boundary_tensor_batch, nrow=4, normalize=True,
   pad_value=0.9).permute(1, 2, 0)
```

```
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(8, 8))
ax1.imshow(orig_grid.cpu().numpy())
ax1.axis('off')
ax1.set_title("Original Images")

ax2.imshow(slic_grid.cpu().numpy())
ax2.axis('off')
ax2.set_title("SLIC-Based Tokenization")

ax3.imshow(slic_boundary_grid.cpu().numpy())
ax3.axis('off')
ax3.set_title("SLIC-Based Tokenization (Bordered)")

plt.tight_layout()
plt.show()
```



Let's Try it with the Larger Cat Image

```
start_time = time.time()
cat_slic_np, cat_boundary_np = superpixel_mosaic_torch(cat_img_tensor.squeeze(0), n_segments=512,
   compactness=30.0)
```

```
elapsed_time = time.time() - start_time
print(f"Time taken: {elapsed_time:.4f} seconds")
```

```
Time taken: 5.4068 seconds
```

```
cat_mosaic_t = torch.from_numpy(cat_slic_np).permute(2, 0, 1)
cat_bound_t = torch.from_numpy(cat_boundary_np).permute(2, 0, 1)

cat_mosaic_t = denormalize_image(cat_mosaic_t, mean, std)
cat_bound_t = denormalize_image(cat_bound_t, mean, std)

cat_mosaic_t = cat_mosaic_t.float().clamp(0,1)
cat_bound_t = cat_bound_t.float().clamp(0,1)

cat_slic_grid = torchvision.utils.make_grid(cat_mosaic_t, nrow=1, normalize=True, pad_value=0.9).permute(1, 2,
  0)
cat_slic_boundary_grid = torchvision.utils.make_grid(cat_bound_t, nrow=4, normalize=True, pad_value=0.9).
  permute(1, 2, 0)
```

```
fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(24, 7))
ax1.imshow(cat_orig_grid.cpu().numpy())
ax1.axis('off')
ax1.set_title("Original Images")

ax2.imshow(cat_slic_grid.cpu().numpy())
ax2.axis('off')
ax2.set_title("SLIC-Based Tokenization")

ax3.imshow(cat_slic_boundary_grid.cpu().numpy())
ax3.axis('off')
ax3.set_title("SLIC-Based Tokenization (Bordered)")

plt.tight_layout()
plt.show()
```



27.8. An End-2-End Solution (Soft Assignment Differentiable Clustering for Superpixel Tokenization)

27.9. Soft Assignment Differentiable Clustering for Superpixel Tokenization (diffSLIC)

The **diffSLIC** algorithm implements a differentiable version of the traditional SLIC (Simple Linear Iterative Clustering) superpixel segmentation method, **allowing it to be integrated into deep neural networks** and **optimized end-to-end**. Instead of hard assigning each pixel to a single cluster, it computes **soft assignments** using local attention mechanisms, allowing gradients to propagate through the segmentation process. Below is an explanation of the main components:

This work is closely related to two influential papers: - **Implicit Integration of Superpixel Segmentation into Fully Convolutional Networks** (Link)
This paper demonstrates how superpixel segmentation can be embedded directly into convolutional networks, enabling the network to learn spatially coherent regions. - **Clustering as Attention: Unified Image Segmentation with Hierarchical Clustering** (Link)
This work formulates clustering as an attention mechanism, providing a unified framework for image segmentation that leverages hierarchical clustering.

27.9.1. `compute_stride_and_padding`

This function calculates the **stride** and **padding** needed to partition an image into a grid that matches the desired superpixel shape. Given an image shape and a target superpixel grid shape, it c omputes how many pixels to move (stride) and how much padding to add so that the image dimensions become compatible with a uniform grid division.

27.9.2. `spixel_upsampling` and `spixel_downsampling`

- **Upsampling:** This function takes low-resolution superpixel features and the corresponding pixel-to-superpixel assignments, then "unfolds" and "folds" the data to reconstruct an upsampled feature map that matches the original image resolution.
- **Downsampling:** This function performs the reverse operation by aggregating pixel-level features into a compact superpixel representation using the provided assignments. Both functions rely on PyTorch's unfolding and folding operations to manage the conversion between pixel and grid representations.

```
def compute_stride_and_padding(img_shape, spixel_shape):
    """
    Args:
        img_shape (Tuple[int, int]): input image shape (height, width)
        spixel_shape (Tuple[int, int]): superpixel image shape (height, width)

    Returns:
        stride (Tuple[int, int]): (stride_h, stride_w)
        padding (Tuple[int, int]): (pad_x, pad_y)
```

```
    """
    height, width = img_shape
    height_s, width_s = spixel_shape
    stride_h = (height + height_s - 1) // height_s
    stride_w = (width + width_s - 1) // width_s
    pad_y = (height_s - height % height_s) % height_s
    pad_x = (width_s - width % width_s) % width_s
    return (stride_h, stride_w), (pad_x, pad_y)

def spixel_upsampling(x, assignments, candidate_radius, stride = None,):
    """
    Upsamples a feature map based on superpixel assignments.

    Args:
        x (torch.Tensor): Superpixel features of shape [B, C, height_s, width_s].
        assignments (torch.Tensor): Pixel-to-superpixel assignments of shape [B, (2*candidate_radius + 1)**2,
  H, W].
        candidate_radius (int): Radius for candidate region.
        stride (Optional[Tuple[int, int]]): Grid stride.

    Returns:
        torch.Tensor: Upsampled features of shape [B, C, H, W].
    """
    batch_size, _, H, W = assignments.shape
    n_channels = x.shape[1]
    height_s, width_s = x.shape[-2:]
    if stride is None:
        stride, padding = compute_stride_and_padding((H, W), (height_s, width_s))
    else:
        _, padding = compute_stride_and_padding((H, W), (height_s, width_s))
    pad_x, pad_y = padding
    assignments = F.pad(assignments, (0, pad_x, 0, pad_y))
    H += pad_y
    W += pad_x
    neighbor_range = candidate_radius * 2 + 1
    candidate_clusters = F.unfold(x, kernel_size=neighbor_range, padding=candidate_radius)
    n_spixels = height_s * width_s
    candidate_clusters = candidate_clusters.reshape(batch_size, n_channels, neighbor_range**2, n_spixels)
    assignments = F.unfold(assignments, kernel_size=stride, stride=stride)
    assignments = assignments.reshape(batch_size, neighbor_range**2, stride[0] * stride[1], n_spixels)
    upsampled_features = torch.einsum('bkcn,bcpn->bkpn', (candidate_clusters, assignments))
    upsampled_features = upsampled_features.contiguous().reshape(batch_size * n_channels, stride[0] *
  stride[1], -1)
    upsampled_features = F.fold(upsampled_features, (H, W), kernel_size=stride, stride=stride)
    upsampled_features = upsampled_features.reshape(batch_size, n_channels, H, W)
    if pad_y > 0:
        upsampled_features = upsampled_features[..., :-pad_y, :]
    if pad_x > 0:
        upsampled_features = upsampled_features[..., :-pad_x]
    return upsampled_features

def spixel_downsampling(x, assignments, stride, candidate_radius):
    """
    Downsamples a feature map based on superpixel assignments.

    Args:
        x (torch.Tensor): Pixel features of shape [B, C, H, W].
        assignments (torch.Tensor): Superpixel-to-pixel assignments of shape [B, (2*candidate_radius + 1)**2,
  height_s, width_s].
        stride (Optional[Tuple[int, int]]): Grid stride.
```

```
        candidate_radius (int): Radius for candidate region.

    Returns:
        torch.Tensor: Downsampled features of shape [B, C, height_s, width_s].
    """
    batch, _, height_s, width_s = assignments.shape
    height, width = x.shape[-2:]
    channels = x.shape[1]
    if stride is None:
        stride, padding = compute_stride_and_padding((height, width), (height_s, width_s))
    else:
        _, padding = compute_stride_and_padding((height, width), (height_s, width_s))
    pad_x, pad_y = padding
    x = F.pad(x, (0, pad_x, 0, pad_y))
    height += pad_y
    width += pad_x
    neighbor_range = candidate_radius * 2 + 1
    kernel_size = (stride[0] * neighbor_range, stride[1] * neighbor_range)
    padding_ = (stride[0] * candidate_radius, stride[1] * candidate_radius)
    n_candidate_pixels = kernel_size[0] * kernel_size[1]
    unfold_elem_feats = F.unfold(x, kernel_size, stride=stride, padding=padding_)
    unfold_elem_feats = unfold_elem_feats.reshape(batch, channels, n_candidate_pixels, height_s, width_s)
    downsampled_features = torch.einsum('bphw,bcphw->bchw', (assignments, unfold_elem_feats))
    return downsampled_features
```

27.9.3. `compute_elem_to_center_assignment` and `compute_center_to_elem_assignment`

These functions compute the **soft assignment matrices**: - **Pixel-to-Center Assignment:** For every pixel in the image, this function calculates its similarity to each cluster center (which includes both color and spatial components) and uses a softmax (with a temperature parameter `tau`) to obtain a probability distribution over the clusters. - **Center-to-Pixel Assignment:** This function computes the reverse assignment, estimating how each cluster center "attends" to the pixels in a local neighborhood. Both assignments are computed with local attention windows defined by a candidate radius.

27.9.4. `update_clst_feats`

After obtaining soft assignments, this function updates the cluster (superpixel) centers by computing a **weighted average of the pixel features**. The weights are the soft assignments computed in the previous step. This is analogous to the update step in the k-means clustering algorithm but in a differentiable form.

```
def compute_elem_to_center_assignment(clst_feats, elem_feats, stride, tau = 0.01, candidate_radius = 1, stable
  = False):
    """
    Computes soft assignments from pixel features to cluster centers.

    Args:
        clst_feats (torch.Tensor): Cluster features [B, C, H_c, W_c].
        elem_feats (torch.Tensor): Element features [B, C, H, W].
        stride (Tuple[int, int]): Grid stride.
        tau (float): Temperature parameter.
        candidate_radius (int): Candidate region radius.
        stable (bool): Whether to use stable softmax.

    Returns:
```

```
        soft_assignment (torch.Tensor): Soft assignments [B, (2*candidate_radius+1)**2, H, W].
        similarities (torch.Tensor): Similarity scores.
    """
    batch_size, channels, height, width = elem_feats.shape
    # Compute the total number of superpixels (from cluster feature map dimensions)
    n_spixels = clst_feats.shape[2] * clst_feats.shape[3]
    # Define the number of candidate clusters in the local neighborhood (e.g., 3x3 if candidate_radius=1)
    neighbor_range = candidate_radius * 2 + 1

    # Use unfold to extract local patches from cluster features with given kernel size and padding
    candidate_clusters = F.unfold(clst_feats, kernel_size=neighbor_range, padding=candidate_radius)
    # Reshape to get candidate cluster features of shape: [batch_size, channels, neighbor_range**2, n_spixels]
    candidate_clusters = candidate_clusters.reshape(batch_size, channels, neighbor_range**2, n_spixels)

    # Unfold the element features using the given stride to obtain patches over the full image
    unfold_elem_feats = F.unfold(elem_feats, kernel_size=stride, stride=stride)
    # Reshape to: [batch_size, channels, (stride_h * stride_w), n_spixels]
    unfold_elem_feats = unfold_elem_feats.reshape(batch_size, channels, stride[0] * stride[1], n_spixels)

    # Compute similarities between candidate cluster features and the unfolded element features using Einstein
  summation
    # 'bkcn,bkpn->bcpn' multiplies candidate clusters and element features over the channel dimension.
    similarities = torch.einsum('bkcn,bkpn->bcpn', (candidate_clusters, unfold_elem_feats))
    # Reshape similarities to merge the candidate dimension: [batch_size * neighbor_range**2, -1, n_spixels]
    similarities = similarities.contiguous().reshape(batch_size * neighbor_range**2, -1, n_spixels)
    # Fold back to spatial dimensions: output shape becomes [batch_size, neighbor_range**2, height, width]
    similarities = F.fold(similarities, (height, width), kernel_size=stride, stride=stride)
    similarities = similarities.reshape(batch_size, neighbor_range**2, height, width)

    # Replace zero similarities with -infinity so that they do not contribute in softmax
    similarities = torch.where(similarities == 0, -torch.inf, similarities)
    # If stable computation is desired, subtract the maximum similarity for numerical stability
    if stable:
        similarities = similarities - similarities.max(1, keepdim=True).values.detach()
    # Apply softmax over the candidate dimension (neighbor_range**2) scaled by temperature tau to obtain soft
  assignments
    soft_assignment = (similarities / tau).softmax(1)
    # Return the soft assignments and the computed similarity scores
    return soft_assignment, similarities

def compute_center_to_elem_assignment(clst_feats, elem_feats, stride, tau = 0.01, candidate_radius = 1, stable
  = False):
    """
    Computes soft assignments from cluster centers to pixel elements.

    Args:
        clst_feats (torch.Tensor): Cluster features [B, C, H_c, W_c].
        elem_feats (torch.Tensor): Element features [B, C, H, W].
        stride (Tuple[int, int]): Grid stride.
        tau (float): Temperature parameter.
        candidate_radius (int): Candidate region radius.
        stable (bool): Whether to use stable softmax.

    Returns:
        soft_assignment (torch.Tensor): Soft assignments [B, (2*candidate_radius+1)**2, H_c, W_c].
        similarities (torch.Tensor): Similarity scores.
    """
    b, c, h, w = clst_feats.shape
    # Determine the number of candidates in the local neighborhood (e.g., 3x3 if candidate_radius=1)
    neighbor_range = candidate_radius * 2 + 1
```

```
    # Define the kernel size for unfolding based on stride and neighborhood size
    kernel_size = (stride[0] * neighbor_range, stride[1] * neighbor_range)
    # Set padding proportional to the candidate radius
    padding = (stride[0] * candidate_radius, stride[1] * candidate_radius)
    # Calculate the total number of candidate pixels from the local neighborhood window
    n_candidate_pixels = kernel_size[0] * kernel_size[1]
    # Unfold element features with the given kernel, padding, and stride
    unfold_elem_feats = F.unfold(elem_feats, kernel_size, padding=padding, stride=stride)
    # Reshape to get: [batch, channels, n_candidate_pixels, h, w]
    unfold_elem_feats = unfold_elem_feats.reshape(b, c, n_candidate_pixels, h, w)
    # Compute similarities via Einstein summation: for each candidate, compute inner product with cluster
features
    similarities = torch.einsum('bcphw,bchw->bphw', (unfold_elem_feats, clst_feats))
    # Replace zeros with -infinity so that these do not affect softmax
    similarities = torch.where(similarities == 0, -torch.inf, similarities)
    # Optionally, perform a stable softmax by subtracting the max value
    if stable:
        similarities = similarities - similarities.max(1, keepdim=True).values.detach()
    # Compute soft assignments from the similarities divided by tau, applying softmax along the candidate
dimension
    soft_assignment = torch.softmax(similarities / tau, dim=1)
    # Return the soft assignments and the computed similarities
    return soft_assignment, similarities

def update_clst_feats(elem_feats, clst_feats, stride, tau = 0.01, candidate_radius = 1, stable = False):
    """
    Updates cluster features via a local attention mechanism.

    Args:
        elem_feats (torch.Tensor): Element features [B, C, H, W].
        clst_feats (torch.Tensor): Cluster features [B, C, H_c, W_c].
        stride (Tuple[int, int]): Grid stride.
        tau (float): Temperature parameter.
        candidate_radius (int): Candidate region radius.
        stable (bool): Whether to use stable softmax.

    Returns:
        new_clst_feats (torch.Tensor): Updated cluster features.
        soft_assignment (torch.Tensor): Soft assignments.
        similarities (torch.Tensor): Similarity scores.
    """
    b, c, h, w = clst_feats.shape
    # Define candidate neighborhood range (e.g., 3x3 if candidate_radius=1)
    neighbor_range = candidate_radius * 2 + 1
    # Define the kernel size based on stride and neighborhood range
    kernel_size = (stride[0] * neighbor_range, stride[1] * neighbor_range)
    # Set the required padding based on the candidate radius and stride
    padding = (stride[0] * candidate_radius, stride[1] * candidate_radius)
    # Calculate total number of candidate pixels
    n_candidate_pixels = kernel_size[0] * kernel_size[1]
    # Unfold element features using the specified kernel size, stride, and padding
    unfold_elem_feats = F.unfold(elem_feats, kernel_size, padding=padding, stride=stride)
    # Reshape the unfolded element features to: [batch, channels, n_candidate_pixels, h, w]
    unfold_elem_feats = unfold_elem_feats.reshape(b, c, n_candidate_pixels, h, w)
    # Compute similarities between unfolded element features and cluster features via inner product using
Einstein summation
    similarities = torch.einsum('bcphw,bchw->bphw', (unfold_elem_feats, clst_feats))
    # Replace zeros with -infinity to ensure they do not contribute to the softmax
    similarities = torch.where(similarities == 0, -torch.inf, similarities)
    # If stable computation is desired, subtract the maximum similarity for numerical stability
```

```
    if stable:
        similarities = similarities - similarities.max(1, keepdim=True).values.detach()
    # Compute soft assignments by applying softmax on the scaled similarities along the candidate dimension
    soft_assignment = torch.softmax(similarities / tau, dim=1)
    # Update the cluster features as the weighted sum of the unfolded element features, weighted by the soft
assignments
    new_clst_feats = torch.einsum('bphw,bcphw->bchw', (soft_assignment, unfold_elem_feats))
    # Return the updated cluster features, the soft assignments, and the similarity scores
    return new_clst_feats, soft_assignment, similarities
```

27.9.5. `DiffSLIC` Module

The `DiffSLIC` class encapsulates the entire differentiable SLIC process: - **Initialization:** If no initial cluster features are provided, it initializes them via adaptive average pooling, effectively partitioning the image into a uniform grid. - **Iterative Refinement:** For a fixed number of iterations (`n_iter`), it refines the cluster centers by updating them with `update_clst_feats` using the computed soft assignments. Optional L2 normalization is applied to keep the features stable. - **Final Assignment:** Finally, it computes the pixel-to-cluster (superpixel) assignment and returns the updated cluster features along with both assignment matrices.

```
class DiffSLIC(nn.Module):
    r"""Differentiable SLIC

    Args:
        n_spixels (int): Number of superpixels.
        n_iter (int): Number of iterations for updating cluster centers.
        tau (float): Temperature parameter (as tau → 0, assignments become deterministic).
        candidate_radius (int): Radius for candidate region.
        normalize (bool): If True, L2-normalize pixel and cluster features.
        stable (bool): Use stable softmax computation.
    """
    def __init__(self,
                 n_spixels: int,
                 n_iter: int = 5,
                 tau: float = 0.01,
                 candidate_radius: int = 1,
                 normalize: bool = True,
                 stable: bool = False) -> None:
        super().__init__()
        # Store parameters for later use.
        self.n_spixels = n_spixels
        self.n_iter = n_iter
        self.tau = tau
        self.candidate_radius = candidate_radius
        self.normalize = normalize
        self.stable = stable

    def forward(self, x, clst_feats=None):
        """
        Args:
            x (torch.Tensor): Input image tensor of shape [B, C, H, W].
            clst_feats (Optional[torch.Tensor]): Initial cluster features [B, C, H_c, W_c].
                If None, they are initialized by adaptive average pooling of x.

        Returns:
            clst_feats (torch.Tensor): Updated cluster features [B, C, H_c, W_c].
```

```
        p2s_assign (torch.Tensor): Pixel-to-superpixel assignments of shape [B, (2*candidate_radius+1)**2,
H, W].
        s2p_assign (torch.Tensor): Superpixel-to-pixel assignments of shape [B,
stride_h*stride_w*(2*candidate_radius+1)**2, H_c, W_c].
    """
    # Get the spatial dimensions of the input image.
    height, width = x.shape[-2:]

    # If no initial cluster features are provided, initialize them via adaptive pooling.
    if clst_feats is None:
        # Compute the target spatial size for superpixels based on desired number of superpixels.
        height_s = int(math.sqrt(self.n_spixels * height / width))
        width_s = int(math.sqrt(self.n_spixels * width / height))
        # Compute the stride for dividing the image into a grid of superpixels.
        stride_h = (height + height_s - 1) // height_s
        stride_w = (width + width_s - 1) // width_s
        stride = (stride_h, stride_w)
        # Initialize cluster features by averaging x over a grid.
        clst_feats = F.adaptive_avg_pool2d(x, (height_s, width_s))
    else:
        height_s, width_s = clst_feats.shape[-2:]
        stride = ((height + height_s) // height_s, (width + width_s) // width_s)

    # Optionally L2-normalize both the pixel features and the cluster features.
    if self.normalize:
        x = x / x.norm(dim=1, keepdim=True)
        clst_feats = clst_feats / clst_feats.norm(dim=1, keepdim=True)

    # Compute necessary padding so that the image dimensions are divisible by the grid size.
    pad_x = (width_s - width % width_s) % width_s
    pad_y = (height_s - height % height_s) % height_s
    x = F.pad(x, (0, pad_x, 0, pad_y))

    # Iteratively update the cluster (superpixel) features using a local attention mechanism.
    for _ in range(self.n_iter):
        clst_feats, s2p_assign, _ = update_clst_feats(x, clst_feats, stride, self.tau, self.
candidate_radius)
        if self.normalize:
            clst_feats = clst_feats / clst_feats.norm(dim=1, keepdim=True)

    # Compute the soft pixel-to-superpixel assignments.
    p2s_assign, _ = compute_elem_to_center_assignment(clst_feats, x, stride, self.tau, self.
candidate_radius)

    # Remove any padding from the assignments.
    if pad_y > 0:
        p2s_assign = p2s_assign[..., :-pad_y, :]
    if pad_x > 0:
        p2s_assign = p2s_assign[..., :-pad_x]

    # Return the final cluster features, pixel-to-superpixel, and superpixel-to-pixel assignments.
    return clst_feats, p2s_assign, s2p_assign

 def extra_repr(self):
    # This function helps with printing the module representation.
    return f'n_spixels={self.n_spixels}, n_iter={self.n_iter}, tau={self.tau}, candidate_radius={self.
candidate_radius}'
```

27.9.6. Utility Functions for Visualization

- **`lbl_to_rgb`** converts a label map to an RGB image using a random color palette.
- **`mark_boundaries_with_labels`** overlays the superpixel boundaries (and their labels) on the original image, for visual inspection.

```
def lbl_to_rgb(lbl, color_palette=None):
    """
    Converts an integer label map to an RGB image using a color palette.

    Args:
        lbl (np.ndarray): Label map of shape (H, W) with integer labels.
        color_palette (Optional[np.ndarray]): Optional palette of shape (n_segments, 3) where each row is an
  RGB color.

    Returns:
        Tuple[np.ndarray, np.ndarray]: A tuple containing:
            - The RGB label image of shape (H, W, 3), where each label in lbl is replaced with its
  corresponding RGB color.
            - The color palette used.
    """
    height, width = lbl.shape  # Get the dimensions of the label map
    if color_palette is None:
        # If no palette is provided, create a random palette.
        # Here, we create a palette with as many colors as there are pixels, which is overkill,
        # but this ensures that every unique label can get a distinct color.
        n_segments = height * width
        color_palette = np.random.randint(0, 255, (n_segments, 3), dtype=np.uint8)
    # Map each label in the label map to its corresponding RGB color.
    rgb_lbl = color_palette[lbl]
    return rgb_lbl, color_palette

def mark_boundaries_with_labels(image, lbl, color_palette=None):
    """
    Overlays superpixel boundaries and labels on an image.

    Args:
        image (np.ndarray): Original image of shape (H, W, C) with pixel values in [0, 255].
        lbl (np.ndarray): Superpixel label map of shape (H, W), where each pixel has an integer label.
        color_palette (Optional[np.ndarray]): Optional color palette to use for label coloring.

    Returns:
        np.ndarray: Image with superpixel boundaries and labels overlaid.
    """
    # Draw boundaries on the image using skimage's mark_boundaries.
    # The boundary color is set to red (1, 0, 0) in normalized RGB space.
    boundary_image = mark_boundaries(image.astype(np.uint8), lbl, color=(1, 0, 0))

    # Get the dimensions of the label map.
    H, W = lbl.shape
    # Find all unique labels in the label map.
    unique_labels = np.unique(lbl)

    # Copy the boundary image to overlay text without modifying the original.
    output_image = boundary_image.copy()

    # Loop over each unique label to compute the centroid and place the label text.
    for label in unique_labels:
        # Get the (row, col) coordinates of all pixels with this label.
```

```
        coords = np.column_stack(np.where(lbl == label))
        # Compute the centroid of these coordinates.
        centroid = coords.mean(axis=0).astype(int)

        # Choose a color for the label text:
        # If a palette is provided and the label index is within range, use the corresponding color
   (normalized to [0,1]);
        # otherwise, default to white.
        if color_palette is not None and label < len(color_palette):
            color = color_palette[label] / 255.0
        else:
            color = (1, 1, 1)

        # Overlay the label text at the computed centroid.
        plt.text(centroid[1], centroid[0], str(label), color=color, fontsize=8, weight='bold')

    return output_image
```

```
n_spix = 256          # Number of superpixels
n_iter = 10           # Number of iterations
tau = 0.01            # Temperature parameter
candidate_radius = 2  # Candidate radius
stable = False        # Use stable softmax
n_freq = 2            # Number of frequency components for positional encoding
pos_weight = 2        # Weight for positional encoding

# Use our cat image
img = cat_img_tensor
H, W = img.shape[-2:]

# Creating a meshgrid from coordinates just like in the first algorithm
coords = torch.stack(torch.meshgrid(torch.linspace(-1, 1, H, dtype=torch.float), torch.linspace(-1, 1, W,
   dtype=torch.float)), -1).unsqueeze(0)

# Sinusoidal positional embedding similar to the ones in Transformers
# Creates a tensor of frequencies (e.g., [1, 2, 4, ...]) to capture positional information at multiple scales
freqs = 2 ** torch.arange(n_freq, dtype=torch.float)

# For each pixel, its coordinates are multiplied by each frequency, effectively creating a vector of scaled
   coordinates
shape = coords.shape[:-1] + (-1,)
scaled_x = (coords[..., None, :] * freqs[..., None]).reshape(shape)

# Two versions of the scaled coordinates are stacked: one unchanged and one shifted by pi/2
scaled_x = torch.stack([scaled_x, scaled_x + 0.5 * torch.pi], -2).reshape(shape)
embedded_x = torch.sin(scaled_x).permute(0, 3, 1, 2) * pos_weight

print("Positional Embedding Shape: ", embedded_x.shape)

# Concatenate image with positional embeddings
inputs = torch.cat([img, embedded_x], 1)

# Creating the DiffSLIC module
diff_slic = DiffSLIC(n_spix, n_iter, tau, candidate_radius, stable=stable)

# Starting a timer and sending the image through the pipeline
start_time = time.time()
feats, assign, p2s = diff_slic(inputs)
end_time = time.time() - start_time
print(f"Computation time: {end_time:.4f} sec")
```

```
assert assign.shape[-2:] == inputs.shape[-2:], f"Mismatch: {assign.shape[-2:]} vs. {inputs.shape[-2:]}"

# Upsample assignments to get a label map
H_s, W_s = feats.shape[-2:]

# Convert the soft assignments to hard assignments
# For each pixel, choose the superpixel (cluster) with the maximum assignment score
# Convert the resulting indices into one-hot encoding with depth equal to (2*candidate_radius + 1)**2
# Rearrange dimensions to get a tensor of shape [B, depth, H_s, W_s]
hard_assign = F.one_hot(assign.argmax(1), (2 * candidate_radius + 1)**2).permute(0, 3, 1, 2).float()

# Create a sequential label map for the superpixels
# Generate a tensor containing values from 0 to (H_s * W_s - 1)
# Reshape it to form a grid of shape [1, 1, H_s, W_s]
label = torch.arange(H_s * W_s, dtype=torch.float).reshape(1, 1, H_s, W_s)

# Upsample the label map back to the full image resolution using the hard assignments
# This produces a label map where each pixel is assigned the label of its corresponding superpixel
label = spixel_upsampling(label, hard_assign, candidate_radius=candidate_radius)

# Compute the expected area (number of pixels) per superpixel
segment_size = H * W / (H_s * W_s)

# Set minimum and maximum size thresholds for enforcing connectivity
min_size = int(0.06 * segment_size)
max_size = int(3.0 * segment_size)

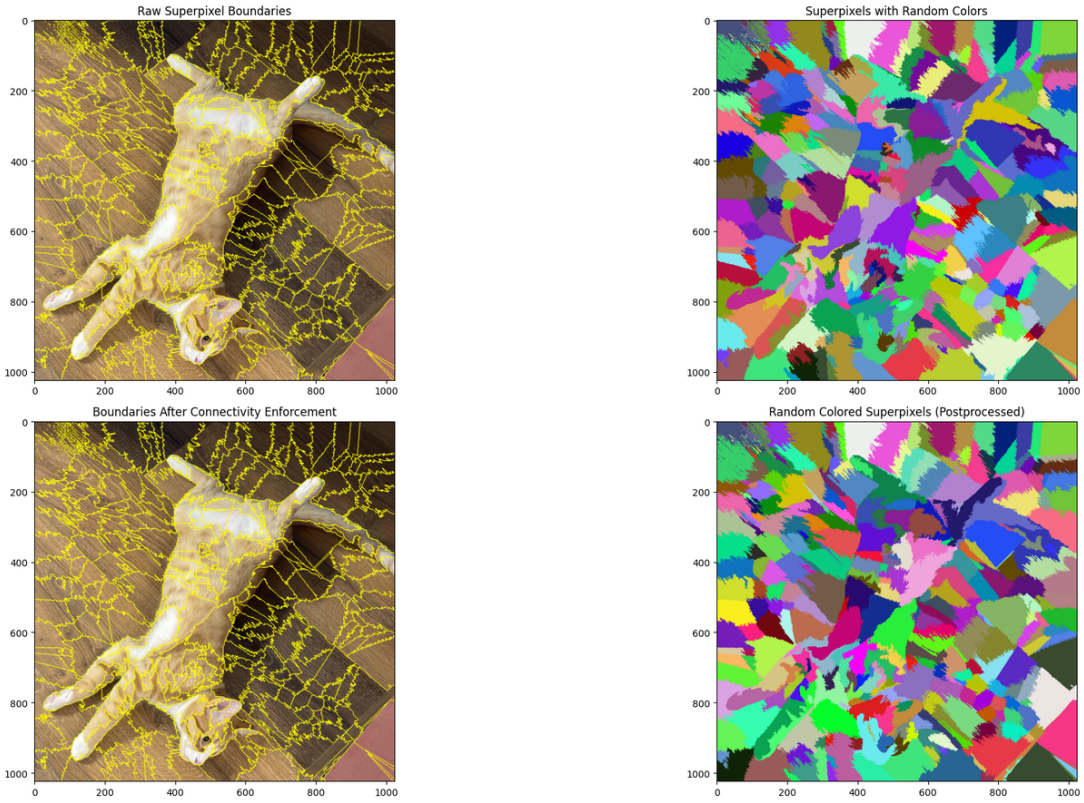
# Convert the label map (from PyTorch tensor) to a NumPy array
np_lbl = label[0, 0].long().numpy()

# Enforce connectivity on the label map using a Cython function
# This ensures that each superpixel forms a contiguous region and meets the size constraints
np_lbl = _enforce_label_connectivity_cython(np_lbl[None], min_size, max_size)[0]
valid_n_spixel = len(np.unique(np_lbl))

print(f"Number of superpixels after connectivity enforcement: {valid_n_spixel}")
```

```
Positional Embedding Shape:  torch.Size([1, 8, 1024, 1024])
Computation time: 110.9753 sec
Number of superpixels after connectivity enforcement: 315
```

```
rgb_face_cat = denormalize_image(cat_img_tensor, mean, std).squeeze(0).permute(1, 2, 0).numpy()
fig = plt.figure(figsize=(24, 12))
ax1 = fig.add_subplot(2, 2, 1)
ax1.imshow(mark_boundaries(rgb_face_cat, np_lbl))
ax1.set_title("Raw Superpixel Boundaries")
ax2 = fig.add_subplot(2, 2, 2)
rgb_lbl, color_palette = lbl_to_rgb(np_lbl)
ax2.imshow(rgb_lbl)
ax2.set_title("Superpixels with Random Colors")
ax3 = fig.add_subplot(2, 2, 3)
np_lbl_conn = _enforce_label_connectivity_cython(np_lbl[None], min_size, max_size)[0]
ax3.imshow(mark_boundaries(rgb_face_cat, np_lbl_conn))
ax3.set_title("Boundaries After Connectivity Enforcement")
ax4 = fig.add_subplot(2, 2, 4)
ax4.imshow(lbl_to_rgb(np_lbl_conn, color_palette)[0])
ax4.set_title("Random Colored Superpixels (Postprocessed)")
plt.tight_layout()
plt.show()
```



27.10. Homework (Optional)

1. **Token-count sensitivity.** Train the grid ViT and a superpixel ViT with matched token counts (say 64 tokens each) on CIFAR-10. Report accuracy, training time, and memory usage.
2. **Adaptive token counts.** Vary the number of superpixels (16, 32, 64, 128) used at training time. Plot accuracy vs. token count and discuss diminishing returns.
3. **SLIC hyperparameters.** For fixed `n_segments`, sweep the `compactness` parameter over {1, 5, 10, 25}. Report accuracy and qualitatively inspect how the superpixels change.
4. **DiffSLIC end-to-end.** Train DiffSLIC jointly with the classifier and compare with SLIC-then-ViT (two stages). Report accuracy and the stability of the learned cluster centres.
5. **Background-heavy images.** Pick an image dataset with lots of background (e.g. aerial imagery, satellite photos). Qualitatively compare grid patches vs. superpixels on a few samples. Where does the superpixel approach shine most?
6. **Speed benchmarking.** Measure wall-clock time per image for the three tokenisers (grid, SLIC, DiffSLIC) across three image resolutions (32, 128, 512 pixels). Discuss scaling behaviour.

Chapter 28

Monocular Depth Estimation

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1qn0Xo-L3RMtchTVvJgH1gYiyIgOTnIGr?usp=sharing

Monocular depth estimation involves predicting the **depth of each pixel in a scene** using only a single RGB image, without the need for stereo input or specialized hardware. Recent advances in deep learning have significantly improved the accuracy of monocular depth estimation by using convolutional neural networks and transformer-based architectures trained on large-scale datasets.

28.1. 1. Setup & Data Download

28.1.1. 1.1 Mount Drive & Install Libraries

We begin by installing all required libraries. Our setup includes: - Core PyTorch libraries for deep learning - Vision-specific packages for image processing - Acceleration libraries for GPU computation

```
!pip install --quiet torch torchvision timm opencv-python matplotlib tqdm decord huggingface_hub
!pip install --quiet --upgrade pip
!pip install --quiet --upgrade "jax[cuda12_pip]" jaxlib
!pip install --quiet --upgrade diffusers["torch"] transformers accelerate safetensors torch torchvision

import os
import random
import xml.etree.ElementTree as ET
from io import BytesIO
import requests
import numpy as np
import torch
from PIL import Image
import cv2
import matplotlib.pyplot as plt
from torchvision.utils import flow_to_image
from torchvision.transforms import Compose, Resize
import torchvision.transforms.functional as F
from torchvision.datasets import VOCDetection, VOCSegmentation
from torchvision.models.optical_flow import raft_large, Raft_Large_Weights
from diffusers import StableDiffusionInpaintPipeline
from decord import VideoReader, cpu
import textwrap

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
```

```
WARNING: jax 0.6.0 does not provide the extra 'cuda12-pip'
Using device: cuda
```

28.1.2. 1.2 Download PASCAL VOC 2012 [6]

The **PASCAL Visual Object Classes (VOC)** dataset is a benchmark for object detection and segmentation research. We'll use the 2012 version which includes: - Over **11,000 images** with 20 object categories - **Bounding box annotations** for every object instance - **Segmentation masks** for a subset of objects

PyTorch provides a convenient `VOCDetection` class to download and parse this dataset automatically. We'll use the training split for our experiments.

```
dataset = VOCSegmentation(root='/data',
                          year='2012',
                          image_set='train',
                          download=True)
```

```
100%|##########| 2.00G/2.00G [01:59<00:00, 16.8MB/s]
```

28.1.3. 1.3 Quick Data Inspection

Visualize a few images, with the corresponding segmentation masks.

```
def show_voc_seg(idx):
    img, mask = dataset[idx]
    im = np.array(img)
    mask_arr = np.array(mask).astype(int)

    num_classes = mask_arr.max() + 1
    rng = np.random.default_rng(seed=42)
    colors = rng.integers(0, 256, size=(num_classes, 3), dtype=np.uint8)
    mask_color = colors[mask_arr]

    overlay = cv2.addWeighted(im, 0.7, mask_color, 0.3, 0)  # :contentReference[oaicite:3]{index=3}

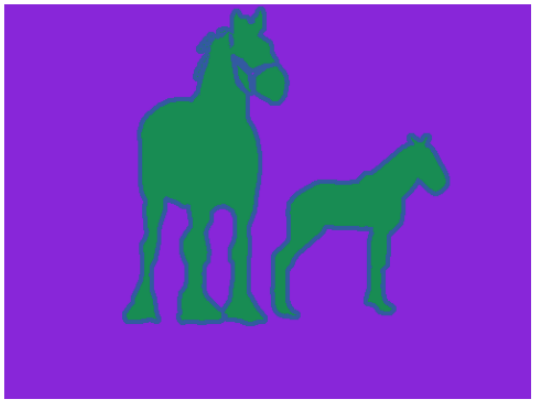
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
    ax1.imshow(im)
    ax1.axis('off')
    ax1.set_title('Image')
    ax2.imshow(mask_color)
    ax2.axis('off')
    ax2.set_title('Segmentation')
    plt.tight_layout()
    plt.show()

for i in random.sample(range(len(dataset)), 3):
    show_voc_seg(i)
```

Image



Segmentation



Image



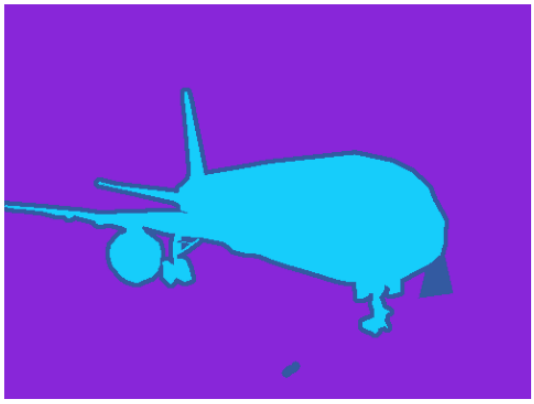
Segmentation



Image



Segmentation



28.2. 2. Monocular Depth Estimation

Extracting depth information from a single image is a fascinating problem with numerous applications in autonomous driving, augmented reality, and robotics. Unlike traditional stereo systems that rely on multiple cameras, monocular depth estimation uses deep learning to infer depth from a single view.

28.2.1. 2.1 Understanding the Challenge

Monocular depth estimation attempts to solve an **inherently ill-posed problem**: recovering 3D geometry from a 2D projection. The key challenges include:

- **Scale ambiguity**: Objects of different sizes at different distances can appear identical in a 2D image
- **Textureless regions**: Areas without distinctive features provide minimal depth cues
- **Material properties**: Reflective or transparent surfaces break assumptions about light transport

Modern deep learning approaches overcome these challenges through extensive training on diverse datasets, leveraging contextual cues, and multi-scale feature extraction.

28.3. Relative vs. absolute depth

Models like MiDaS [25] do not output depth in metres. They output *relative* depth: a per-pixel value that preserves ordering and ratios, but whose overall scale and offset are unknown. From a single uncalibrated image with no prior on camera parameters, absolute depth is fundamentally ambiguous: the same image could be taken of a small table at close distance or of a large table far away and look identical.

In practice this limitation matters less than it sounds. Many downstream applications only need ordering ("what is closer, the cup or the laptop?") or relative scale ("the chair is about half as far as the wall"). For robotics or 3D reconstruction where metric distances are required, the standard workaround is to anchor the relative prediction with at least one sparse ground-truth measurement, a single LiDAR point, an ArUco marker, or a known-size object in the scene, and linearly rescale the whole depth map.

MiDaS, Depth Anything, and ZoeDepth are the three architectures worth knowing. MiDaS is the classic mixed-dataset baseline; Depth Anything adds a much larger training corpus; ZoeDepth fine-tunes a MiDaS-style backbone on metric-depth datasets and can produce absolute depth for specific scene categories.

28.3.1. 2.2 MiDaS Architecture

We'll use the **MiDaS** (Mixed Depth Supervision) model, which represents the state-of-the-art in monocular depth estimation. Key characteristics include:

- **Multi-dataset training**: Combines diverse datasets with different scale factors and depth ranges
- **Scale and shift invariance**: Produces consistent relative depth regardless of camera parameters
- **Dense prediction network**: Uses a DPT (Dense Prediction Transformer) architecture for high-resolution output

For this notebook, we'll use the **DPT-Large** variant which offers the highest accuracy with a reasonable inference speed on GPU hardware.

```
# Load the MiDaS model and prepare the transform
model_type = "DPT_Large"     # MiDaS v3 - Large     (highest accuracy, slowest inference speed)
#model_type = "DPT_Hybrid"   # MiDaS v3 - Hybrid    (medium accuracy, medium inference speed)
#model_type = "MiDaS_small"  # MiDaS v2.1 - Small   (lowest accuracy, highest inference speed)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
midas = torch.hub.load("intel-isl/MiDaS", model_type).to(device)
midas.eval()
midas_transforms = torch.hub.load("intel-isl/MiDaS", "transforms")

if model_type == "DPT_Large" or model_type == "DPT_Hybrid":
    transform = midas_transforms.dpt_transform
else:
    transform = midas_transforms.small_transform
```

```
/usr/local/lib/python3.11/dist-packages/torch/hub.py:330: UserWarning: You are
about to download and run code from an untrusted repository. In a future
release, this won't be allow ... [448 chars truncated]
  warnings.warn(
```

```
Downloading: "https://github.com/intel-isl/MiDaS/zipball/master" to
/root/.cache/torch/hub/master.zip
```

```
/usr/local/lib/python3.11/dist-packages/timm/models/layers/__init__.py:48:
FutureWarning: Importing from timm.models.layers is deprecated, please import
via timm.layers
  warnings.warn(f"Importing from {__name__} is deprecated, please import via
timm.layers", FutureWarning)
```

```
Downloading: "https://github.com/isl-
org/MiDaS/releases/download/v3/dpt_large_384.pt" to
/root/.cache/torch/hub/checkpoints/dpt_large_384.pt
```

```
100%|##########| 1.28G/1.28G [00:37<00:00, 36.4MB/s]
Using cache found in /root/.cache/torch/hub/intel-isl_MiDaS_master
```

28.3.2. 2.3 Inference and Visualization

Let's apply our depth estimation model to images from the PASCAL VOC dataset. We'll: 1. Select an image from our dataset 2. Preprocess it according to MiDaS requirements 3. Run inference to generate a depth map 4. Visualize the result using the 'magma' colormap, where **closer objects appear yellow/white** and **distant objects appear purple/black**

```
def predict_and_visualize_depth(img_idx):
    """
    Predicts depth for an image from our dataset and visualizes the results.

    Args:
        img_idx (int): Index of the image in the dataset.
    """
    img, _ = dataset[img_idx]
    input_img = np.array(img)[...,::-1].copy()  # RGB to BGR
    input_tensor = transform(input_img).to(device)

    with torch.no_grad():
        depth_map = midas(input_tensor).cpu().squeeze().numpy()

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))
```

```
    ax1.imshow(img)
    ax1.set_title("Original Image")
    ax1.axis('off')

    depth_plot = ax2.imshow(depth_map, cmap='magma')
    ax2.set_title("Depth Estimation")
    ax2.axis('off')

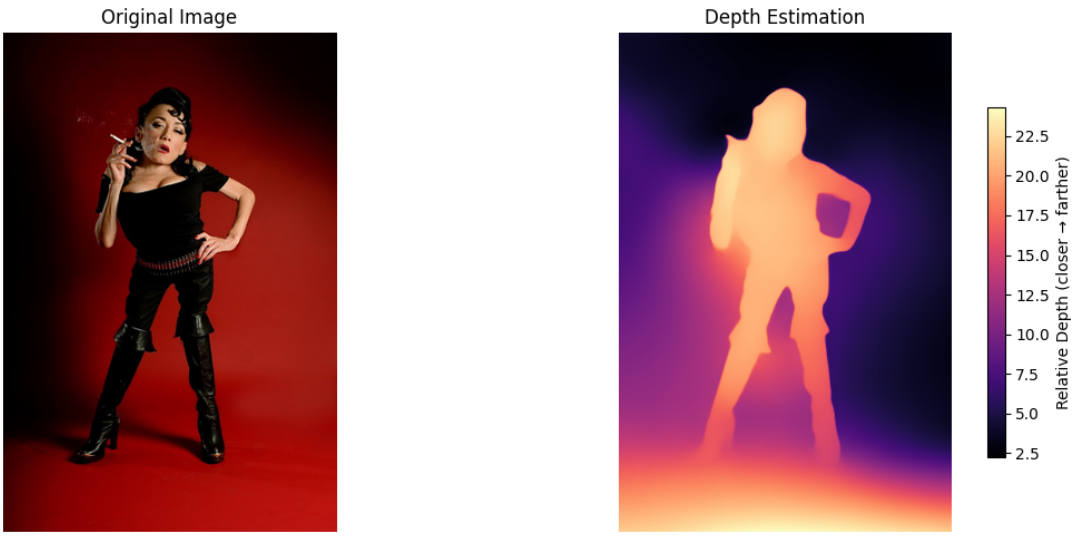
    cbar = fig.colorbar(depth_plot, ax=ax2, shrink=0.7)
    cbar.set_label('Relative Depth (closer → farther)')

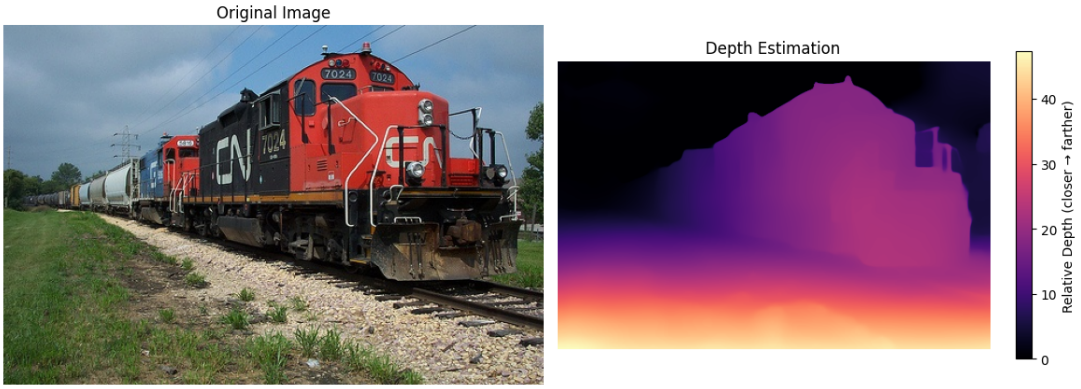
    plt.tight_layout()
    plt.show()

    return depth_map

for i in random.sample(range(len(dataset)), 3):
  depth_result = predict_and_visualize_depth(i)
```







28.4. Homework (Optional)

1. **MiDaS variants.** Load `MiDaS`, `MiDaS_small`, and `DPT_Large` via `torch.hub`. Run each on three test images and compare qualitatively. Report inference latency and memory usage.
2. **Relative -> metric via anchoring.** For an image where you can estimate one known distance (e.g. a standard doorway is 2 m tall), compute a scale factor that maps the MiDaS output to metres. Apply the same scale to the whole depth map and verify plausibility on other objects.
3. **Video consistency.** Run MiDaS frame-by-frame on a short video clip. Observe temporal flicker in the depth predictions. Propose and implement a simple smoothing (e.g. exponential moving average) that reduces flicker.
4. **Depth-based blur.** Use the predicted depth to add a depth-of-field effect: blur pixels in proportion to their distance from a chosen focal depth. Present before / after for two images.
5. **3D reconstruction.** Using the camera's intrinsic matrix (pick reasonable defaults for a smartphone image), back-project the depth map to a 3D point cloud. Save it as a `.ply` file and visualise in any 3D viewer.
6. **Compare with ground truth.** Pick a scene from the KITTI or NYU Depth V2 dataset where ground-truth depth is available. Compute absolute depth error (after optimal scale/shift alignment) and analyze regions where MiDaS struggles.

Chapter 29

Optical Flow

Authors: Tamás Takács, Imre Molnár
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1SuzO8k4I6rtKsGGvcMhzB0lBh56DL4ov?usp=sharing

Optical flow estimates the **apparent motion of objects between consecutive video frames** by analyzing changes in pixel intensity. Traditional methods rely on assumptions about brightness and motion consistency, but deep learning has enabled more robust and accurate flow estimation by learning complex motion patterns directly from large video datasets using CNNs and encoder-decoder architectures.

29.1. 1. Setup & Data Download

29.1.1. 1.1 Mount Drive & Install Libraries

We begin by installing all required libraries. Our setup includes: - Core PyTorch libraries for deep learning - Vision-specific packages for image processing - Acceleration libraries for GPU computation

```
!pip install --quiet torch torchvision timm opencv-python matplotlib tqdm decord huggingface_hub
!pip install --quiet --upgrade pip
!pip install --quiet --upgrade "jax[cuda12_pip]" jaxlib
!pip install --quiet --upgrade diffusers["torch"] transformers accelerate safetensors torch torchvision

import os
import random
import xml.etree.ElementTree as ET
from io import BytesIO
import requests
import numpy as np
import torch
from PIL import Image
import cv2
import matplotlib.pyplot as plt
from torchvision.utils import flow_to_image
from torchvision.transforms import Compose, Resize
import torchvision.transforms.functional as F
from torchvision.datasets import VOCDetection, VOCSegmentation
from torchvision.models.optical_flow import raft_large, Raft_Large_Weights
from diffusers import StableDiffusionInpaintPipeline
from decord import VideoReader, cpu
import textwrap

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
```

```
WARNING: jax 0.6.0 does not provide the extra 'cuda12-pip'
Using device: cuda
```

29.1.2. 1.2 Download PASCAL VOC 2012 [6]

The **PASCAL Visual Object Classes (VOC)** dataset is a benchmark for object detection and segmentation research. We'll use the 2012 version which includes: - Over **11,000 images** with 20 object categories - **Bounding box annotations** for every object instance - **Segmentation masks** for a subset of objects

PyTorch provides a convenient `VOCDetection` class to download and parse this dataset automatically. We'll use the training split for our experiments.

```
dataset = VOCSegmentation(root='data',
                                year='2012',
                                image_set='train',
                                download=True)
```

```
100%|##########| 2.00G/2.00G [02:44<00:00, 12.1MB/s]
```

29.1.3. 1.3 Quick Data Inspection

Visualize a few images, with the corresponding segmentation masks.

```
def show_voc_seg(idx):
    img, mask = dataset[idx]
    im = np.array(img)
    mask_arr = np.array(mask).astype(int)

    num_classes = mask_arr.max() + 1
    rng = np.random.default_rng(seed=42)
    colors = rng.integers(0, 256, size=(num_classes, 3), dtype=np.uint8)
    mask_color = colors[mask_arr]

    overlay = cv2.addWeighted(im, 0.7, mask_color, 0.3, 0)  # :contentReference[oaicite:3]{index=3}

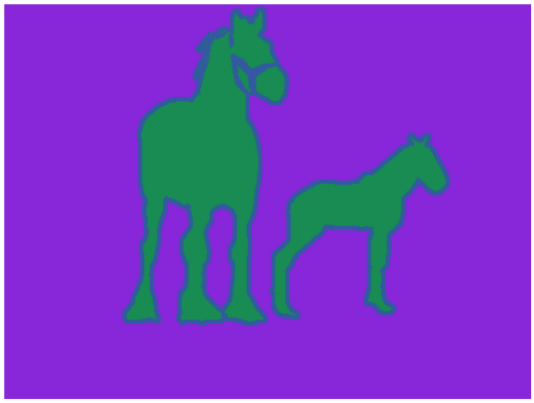
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
    ax1.imshow(im)
    ax1.axis('off')
    ax1.set_title('Image')
    ax2.imshow(mask_color)
    ax2.axis('off')
    ax2.set_title('Segmentation')
    plt.tight_layout()
    plt.show()

for i in random.sample(range(len(dataset)), 3):
    show_voc_seg(i)
```

Image



Segmentation



Image



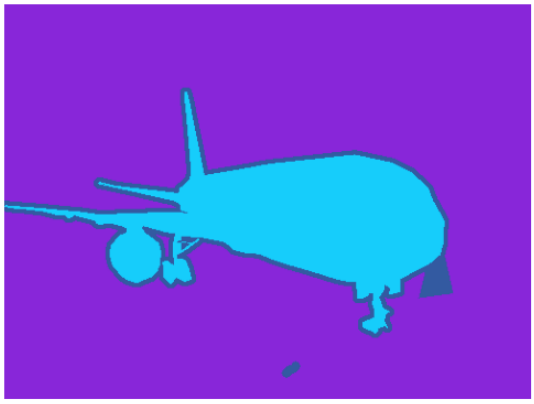
Segmentation



Image



Segmentation



29.2. 2. Optical Flow with RAFT

This section demonstraters how to use Optical Flow wiht RAFT. We will:

1. Prepare and load sample video frames.
2. Load and preprocess frames for RAFT.
3. Run RAFT to estimate motion and visualize results.

29.2.1. 2.1 Introduction

Optical Flow is a technique that estimates per-pixel motion between two images or video frames. It outputs a vector field representing how each pixel moves from one frame to the next. Common assumptions for optical flow are:

- **Brightness constancy**: pixel intensity remains the same between frames.
- **Spatial smoothness**: nearby pixels tend to have similar motion.

RAFT is a modern optical flow model that:

- Constructs a full correlation volume between all pairs of pixels.
- Updates an initial flow estimate through many recurrent refinement iterations.
- Achieves accurate flow even in challenging scenes.

29.2.2. 2.2 Prepare Video and Frames

We'll download a short clip and extract specific frames for demonstration.

```
# If you are on Google Colab you can use the following command to download the video file
!wget -q https://download.pytorch.org/tutorial/pexelscom_pavel_danilyuk_basketball_hd.mp4 -O data/dance.mp4
```

```
vr = VideoReader("data/dance.mp4", ctx=cpu(0))

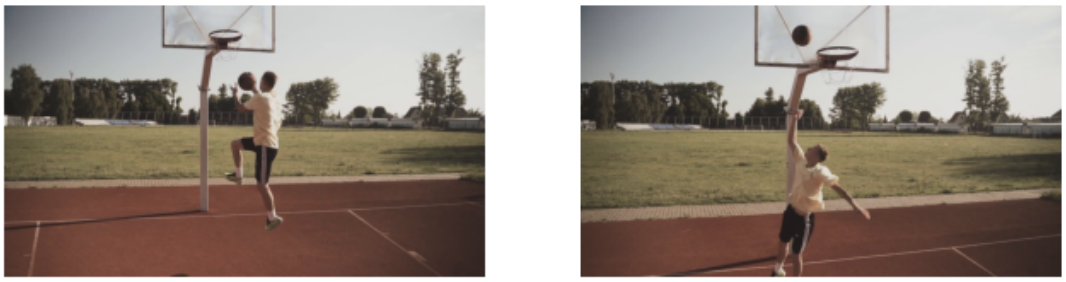
# Choose frame indices
frame_ids = [100, 101, 150, 151]
np_frames = vr.get_batch(frame_ids).asnumpy()  # shape (4, H, W, C)
frames = torch.from_numpy(np_frames).permute(0, 3, 1, 2)  # (4, C, H, W)
batch1 = frames[[0, 2]]  # frame 100 and 150
batch2 = frames[[1, 3]]  # frame 101 and 151
```

Visualize example frames:

```
# Utility to plot batches in grid
plt.rcParams["savefig.bbox"] = "tight"

def plot(imgs):
    if not isinstance(imgs[0], list):
        imgs = [imgs]
    rows, cols = len(imgs), len(imgs[0])
    fig, axes = plt.subplots(rows, cols, figsize=(4*cols, 3*rows))
    for i, row in enumerate(imgs):
        for j, img in enumerate(row):
            ax = axes[i][j] if rows>1 else axes[j]
            ax.imshow(F.to_pil_image(img.cpu()))
            ax.axis('off')
    plt.show()
```

```
plot(batch1)
```



29.3. What RAFT learned that Lucas-Kanade could not

Dense optical flow has a long history before deep learning. Classical methods like Lucas-Kanade and Horn-Schunck write down an explicit smoothness prior and solve a constrained optimisation over the flow field. They work well when motion is small, lighting is stable, and textures are rich. They degrade gracefully in one or two failure modes at a time but struggle with combinations: large motion, thin structures, occlusion boundaries, low-texture regions.

RAFT (Recurrent All-pairs Field Transforms) replaces the hand-designed prior with a learned recurrent update. For each target pixel the network queries a pre-computed all-pairs correlation volume over the source image, then iteratively refines a flow estimate through a GRU-like update cell. The correlation volume is the key: it tells the refinement cell where each pixel "could be" in the other frame, independent of the current flow estimate. That indirection lets the network handle large motions that would trap a gradient-descent classical method in a local minimum.

Practically, the trade-offs are: RAFT needs more memory than any classical method (the correlation volume is `H * W * H * W` entries), is slower per frame at high resolution, requires training data, and struggles on motion patterns that were rare in the training distribution (e.g. extreme camera shake, unusual textures, non-rigid deformations of novel objects). Classical methods are still the right choice for embedded deployment and for domains where no training data is available.

29.4. 2.3 Load RAFT Model and Preprocessing

We'll use the raft_large model from torchvision, with pretrained weights.

```
weights = Raft_Large_Weights.DEFAULT
transforms = weights.transforms()
model = raft_large(weights=weights, progress=False).to(device).eval()
```

```
Downloading:
"https://download.pytorch.org/models/raft_large_C_T_SKHT_V2-ff5fadd5.pth" to
/root/.cache/torch/hub/checkpoints/raft_large_C_T_SKHT_V2-ff5fadd5.pth
```

RAFT expect inputs at a fixed size, so we must resize, and also apply the model's built-in normalization.

```
# Resize + normalize
def preprocess(img1, img2):
    img1 = F.resize(img1, [520, 960], antialias=False)
    img2 = F.resize(img2, [520, 960], antialias=False)
    return transforms(img1, img2)

batch1_proc, batch2_proc = preprocess(batch1, batch2)
```

29.5. 2.4 Compute and Visualize Optical Flow

We pass the two-frame batches to RAFT. RAFT returns a list of flow tensors (one per refinement iteration). We take the last as our final output.

```
with torch.no_grad():
    flows_iter = model(batch1_proc.to(device), batch2_proc.to(device))
final_flow = flows_iter[-1]  # (N, 2, H, W)
```

29.6. 2.5 Visualizing Optical Flow Results

Now that we have computed the optical flow for our input images, let's inspect the outputs.
The cell below defines `visualize_flow_batch`, which displays each image alongside its estimated flow as (1) overlaid quiver arrows colored by speed, (2) a dense color-coded flow map, and (3) a magnitude heatmap, then runs it on our processed batch.

```
def visualize_flow_batch(orig_batch, flow_batch,
                         step=16, cmap='inferno',
                         figsize=(16, 8), dpi=150):
    """
    orig_batch: tensor of shape (N,3,H,W) in [0,1]
    flow_batch: tensor of shape (N,2,H,W) in pixel units
    step: subsampling step for quiver arrows
    figsize: overall figure size (width, height) in inches
    dpi: dots per inch for higher resolution in notebooks
    """
    N, _, H, W = flow_batch.shape
    ncols = 3
    fig, axes = plt.subplots(N, ncols,
                             figsize=figsize, dpi=dpi)
    if N == 1:
        axes = axes[np.newaxis, ...]

    for i in range(N):
        orig = orig_batch[i].permute(1,2,0).cpu().numpy()
        flow = flow_batch[i].cpu()
        u, v = flow[0].numpy(), flow[1].numpy()
        mag = np.sqrt(u**2 + v**2)

        # PANEL 1: Overlay + Arrows (now first)
        ax = axes[i,0]
        ax.imshow(orig)
        ys, xs = np.mgrid[step//2:H:step, step//2:W:step]
        ax.quiver(
            xs, ys,
            u[step//2:H:step, step//2:W:step],
            v[step//2:H:step, step//2:W:step],
```

```
            mag[step//2:H:step, step//2:W:step],
            angles='xy', scale_units='xy', scale=0.5,
            width=0.005, headwidth=4, headlength=6,
            headaxislength=5, cmap=cmap
        )
        ax.set_title("Overlay + Arrows")
        ax.axis('off')

        # PANEL 2: Color-coded flow
        flow_img_t = flow_to_image(flow)
        if isinstance(flow_img_t, torch.Tensor):
            fimg = flow_img_t.cpu().numpy()
        else:
            fimg = flow_img_t
        if fimg.ndim == 3 and fimg.shape[0] == 3:
            fimg = fimg.transpose(1,2,0)
        fimg = fimg.astype(np.float32) / 255.0

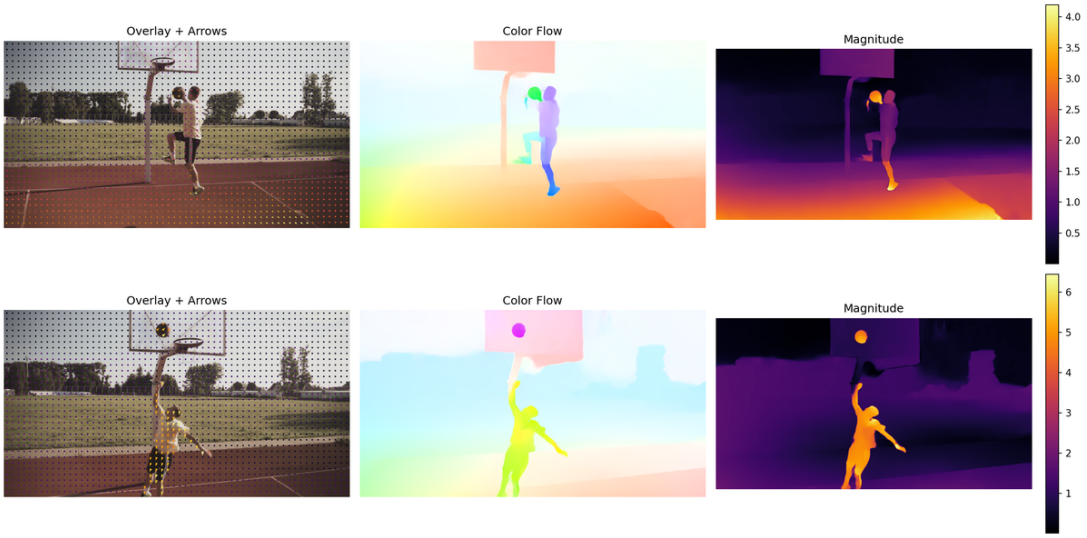
        ax = axes[i,1]
        ax.imshow(fimg)
        ax.set_title("Color Flow")
        ax.axis('off')

        # PANEL 3: Magnitude heatmap
        ax = axes[i,2]
        im = ax.imshow(mag, cmap=cmap)
        ax.set_title("Magnitude")
        ax.axis('off')
        fig.colorbar(im, ax=ax, fraction=0.046, pad=0.04)

    plt.tight_layout()
    plt.show()

orig_norm = (batch1_proc + 1) / 2

visualize_flow_batch(orig_norm.cpu(), final_flow.cpu(), step=16)
```



29.7. Homework (Optional)

1. **Compare RAFT with OpenCV Farneback.** Run both methods on three consecutive pairs from a test video. Visualise both flow fields side-by-side and discuss where they disagree.
2. **Large motion stress test.** Create a pair of frames with a fast-moving object (skip 5 frames apart in a 30 fps video). Run RAFT and observe whether the flow field captures the object motion correctly. Repeat with 15 and 30 frames apart.
3. **Occlusion detection.** Run RAFT in both directions (forward and backward flow) for two consecutive frames. Compute a consistency map: regions where `F(forward) + F(backward)` has large magnitude are likely occlusions. Visualise the occlusion mask.
4. **Flow magnitude histogram.** On a 100-frame clip, compute per-frame median flow magnitude. Plot the time series and identify frames of high motion.
5. **RAFT-Small vs. RAFT-Large.** Load the small and large variants. Report accuracy (on any ground-truth pair you have) and per-frame latency. When does the small model's speed win over the large model's accuracy?
6. **Flow-based frame interpolation.** Given frames `A` and `C`, compute the flow `A -> C`, warp `A` halfway toward `C` (by scaling the flow by 0.5), and compare to the true middle frame `B`. Discuss where the simple warping fails.

Chapter 30

Text-Guided Image Inpainting

Authors: Máté Badó Miklós
Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1H0ewoxo7PG01r1P9ePJCsd2DHhXAOHNF?usp=sharing

Image inpainting refers to the process of **filling in missing or corrupted regions of an image** in a visually plausible way. Neural inpainting uses deep learning models, such as convolutional autoencoders or generative adversarial networks (GANs), to learn semantic context and texture patterns, allowing them to reconstruct complex image regions with realistic detail and structure.

30.1. 1. Setup & Data Download

30.1.1. 1.1 Mount Drive & Install Libraries

We begin by installing all required libraries. Our setup includes: - Core PyTorch libraries for deep learning - Vision-specific packages for image processing - Acceleration libraries for GPU computation

```
!pip install --quiet torch torchvision timm opencv-python matplotlib tqdm decord huggingface_hub
!pip install --quiet --upgrade pip
!pip install --quiet --upgrade "jax[cuda12_pip]" jaxlib
!pip install --quiet --upgrade diffusers["torch"] transformers accelerate safetensors torch torchvision

import os
import random
import xml.etree.ElementTree as ET
from io import BytesIO
import requests
import numpy as np
import torch
from PIL import Image
import cv2
import matplotlib.pyplot as plt
from torchvision.utils import flow_to_image
from torchvision.transforms import Compose, Resize
import torchvision.transforms.functional as F
from torchvision.datasets import VOCDetection, VOCSegmentation
from torchvision.models.optical_flow import raft_large, Raft_Large_Weights
from diffusers import StableDiffusionInpaintPipeline
from decord import VideoReader, cpu
import textwrap

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
```

```
WARNING: jax 0.6.0 does not provide the extra 'cuda12-pip'
Using device: cuda
```

30.1.2. 1.2 Download PASCAL VOC 2012 [6]

The **PASCAL Visual Object Classes (VOC)** dataset is a benchmark for object detection and segmentation research. We'll use the 2012 version which includes: - Over **11,000 images** with 20 object categories - **Bounding box annotations** for every object instance - **Segmentation masks** for a subset of objects

PyTorch provides a convenient `VOCDetection` class to download and parse this dataset automatically. We'll use the training split for our experiments.

```
dataset = VOCSegmentation(root='data',
                          year='2012',
                          image_set='train',
                          download=True)
```

```
100%|##########| 2.00G/2.00G [04:59<00:00, 6.69MB/s]
```

30.1.3. 1.3 Quick Data Inspection

Visualize a few images, with the corresponding segmentation masks.

```
def show_voc_seg(idx):
    img, mask = dataset[idx]
    im = np.array(img)
    mask_arr = np.array(mask).astype(int)

    num_classes = mask_arr.max() + 1
    rng = np.random.default_rng(seed=42)
    colors = rng.integers(0, 256, size=(num_classes, 3), dtype=np.uint8)
    mask_color = colors[mask_arr]

    overlay = cv2.addWeighted(im, 0.7, mask_color, 0.3, 0)  # :contentReference[oaicite:3]{index=3}

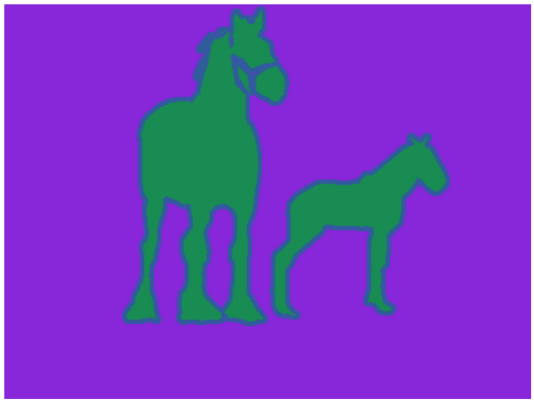
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 5))
    ax1.imshow(im)
    ax1.axis('off')
    ax1.set_title('Image')
    ax2.imshow(mask_color)
    ax2.axis('off')
    ax2.set_title('Segmentation')
    plt.tight_layout()
    plt.show()

for i in random.sample(range(len(dataset)), 3):
    show_voc_seg(i)
```

Image



Segmentation



Image



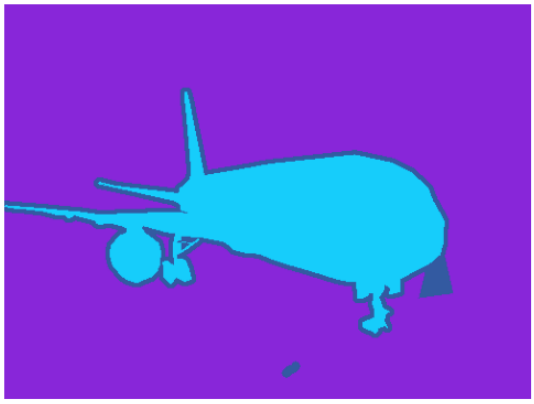
Segmentation



Image



Segmentation



30.2. 2. Text-Guided Inpainting with Stable Diffusion

This section demonstrates a workflow for performing text-guided image inpainting using a state-of-the-art diffusion model (Stable Diffusion). Specifically, we will:

1. Load and initialize the pretrained Stable Diffusion inpainting pipeline.
2. Prepare a source image and construct a corresponding binary mask.
3. Explore key hyperparameters for inpainting, such as inference steps and guidance scale.
4. Apply the inpainting model with multiple text prompts to compare semantic outputs.
5. Visualize and analyze results, discussing qualitative differences.

30.3. 2.1 Introduction

Text-Guided Inpainting involves filling in missing or masked regions of an image based on a natural language prompt. By leveraging a powerful diffusion model, we can generate new content that aligns with semantic guidance while integrating seamlessly with the original context. Common use cases include:

- Restoring damaged photos by describing missing elements.
- Creative content generation (adding objects, characters, scenery).
- Interactive image editing in design pipelines.

Fundamentally, diffusion-based inpainting operates by gradually denoising an image conditioned on both the visible pixels and a textual embedding. Stable Diffusion's inpainting pipeline extends the base model by incorporating mask-aware attention and blending strategies to prevent artifacts at mask boundaries.

Key components:

- **Source Image**: the original image with the region to be inpainted.
- **Mask**: a binary map (255 inside, 0 outside) indicating where to generate new pixels.
- **Text Prompt**: natural language guide that shapes the content in the masked area.
- **Hyperparameters**: inference steps (trade-off speed vs. quality), and guidance scale (controls adherence to prompt).

30.4. Mask quality and prompt engineering

Text-guided inpainting rewards discipline on two inputs that are easy to produce carelessly: the binary mask and the text prompt.

Mask quality. The pipeline replaces every pixel inside the mask with a diffusion-model sample conditioned on the surrounding context and the text prompt. A mask that is too tight crops into the target object and produces visible boundary artefacts. A mask that is too loose gives the model more canvas than necessary and can hallucinate unwanted content. The sweet spot is a mask that fully covers the object plus a small safety margin (5-10 pixels). Anti-aliasing the mask boundary helps the diffusion model blend the generated region into the original, reducing the tell-tale "pasted" look.

Prompt engineering. The prompt has to describe not just *what* you want, but how it fits into the surrounding image. "A red apple" works; "a red apple on a wooden table next to the existing objects, natural lighting" usually works much better because it anchors the generation to the context. Negative prompts are equally important: `"blurry, low quality, text, watermark"` reliably removes common failure modes from the output distribution.

A useful debugging move: generate with four different random seeds and pick the best. Seed variance at a given prompt is often larger than the gain from a better prompt, so seed exploration should happen before prompt rewriting.

30.5. 2.2 Loading the Pretrained Inpainting Pipeline

Initialize the Stable Diffusion inpainting model using half-precision (float16) for faster inference on GPU. This cell may take several seconds to download weights if not cached.

```
pipe = StableDiffusionInpaintPipeline.from_pretrained(
    "CompVis/stable-diffusion-v1-4",  # base model checkpoint
    revision="fp16",                   # use fp16 weights
    torch_dtype=torch.float16,          # enforce half-precision
    safety_checker=None                 # optional: disable safety for custom content
).to(device)

pipe.enable_attention_slicing()  # reduce memory footprint
```

```
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94:
UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab
(https://huggingface.co/settings/tokens), set it as secret in your Google Colab
and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access
public models or datasets.
  warnings.warn(
```

```
model_index.json:   0%|          | 0.00/543 [00:00<?, ?B/s]
```

```
/usr/local/lib/python3.11/dist-
packages/diffusers/pipelines/pipeline_loading_utils.py:285: FutureWarning: You
are loading the variant fp16 from CompVis/stable-diffusion-v1-4 via `r ... [255
chars truncated]
 The Diffusers team and community would be very grateful if you could open an
issue: https://github.com/huggingface/diffusers/issues/new with the title
```

```
'CompVis/stable-diffusion-v1 ... [72 chars truncated]
  warnings.warn(
```

```
Fetching 14 files:   0%|          | 0/14 [00:00<?, ?it/s]
```

```
Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed.
Falling back to regular HTTP download. For better performance, install the
package with: `pip insta ... [51 chars truncated]
WARNING:huggingface_hub.file_download:Xet Storage is enabled for this repo, but
the 'hf_xet' package is not installed. Falling back to regular HTTP download.
For better performance ... [89 chars truncated]
```

```
preprocessor_config.json:   0%|          | 0.00/342 [00:00<?, ?B/s]
```

```
scheduler_config-checkpoint.json:   0%|           | 0.00/209 [00:00<?, ?B/s]
```

```
scheduler_config.json:   0%|          | 0.00/307 [00:00<?, ?B/s]
```

```
tokenizer_config.json:   0%|          | 0.00/788 [00:00<?, ?B/s]
```

```
merges.txt:   0%|          | 0.00/525k [00:00<?, ?B/s]
```

```
pytorch_model.bin:   0%|          | 0.00/246M [00:00<?, ?B/s]
```

```
special_tokens_map.json:   0%|          | 0.00/472 [00:00<?, ?B/s]
```

```
Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed.
Falling back to regular HTTP download. For better performance, install the
package with: `pip insta ... [51 chars truncated]
WARNING:huggingface_hub.file_download:Xet Storage is enabled for this repo, but
the 'hf_xet' package is not installed. Falling back to regular HTTP download.
For better performance ... [89 chars truncated]
```

```
config.json:   0%|          | 0.00/572 [00:00<?, ?B/s]
```

```
config.json:   0%|          | 0.00/772 [00:00<?, ?B/s]
```

```
vocab.json:   0%|          | 0.00/1.06M [00:00<?, ?B/s]
```

```
diffusion_pytorch_model.bin:   0%|           | 0.00/1.72G [00:00<?, ?B/s]
```

```
config.json:   0%|          | 0.00/550 [00:00<?, ?B/s]
```

```
Xet Storage is enabled for this repo, but the 'hf_xet' package is not installed.
Falling back to regular HTTP download. For better performance, install the
package with: `pip insta ... [51 chars truncated]
WARNING:huggingface_hub.file_download:Xet Storage is enabled for this repo, but
the 'hf_xet' package is not installed. Falling back to regular HTTP download.
For better performance ... [89 chars truncated]
```

```
diffusion_pytorch_model.bin:   0%|           | 0.00/167M [00:00<?, ?B/s]
```

```
/usr/local/lib/python3.11/dist-
packages/diffusers/pipelines/pipeline_loading_utils.py:889: FutureWarning: You
are using a legacy checkpoint for inpainting with Stable Diffusion, th ... [770
chars truncated]
  deprecate("StableDiffusionInpaintPipelineLegacy", "1.0.0",
deprecation_message, standard_warn=False)
```

```
Loading pipeline components...:   0%|          | 0/6 [00:00<?, ?it/s]
```

```
An error occurred while trying to fetch /root/.cache/huggingface/hub/models--
CompVis--stable-
diffusion-v1-4/snapshots/2880f2ca379f41b0226444936bb7a6766a227587/unet: Error no
file n ... [184 chars truncated]
Defaulting to unsafe serialization. Pass `allow_pickle=False` to raise an error
instead.
/usr/local/lib/python3.11/dist-
packages/transformers/models/clip/feature_extraction_clip.py:28: FutureWarning:
The class CLIPFeatureExtractor is deprecated and will be removed in v ... [64
chars truncated]
  warnings.warn(
An error occurred while trying to fetch /root/.cache/huggingface/hub/models--
CompVis--stable-
diffusion-v1-4/snapshots/2880f2ca379f41b0226444936bb7a6766a227587/vae: Error no
file na ... [182 chars truncated]
Defaulting to unsafe serialization. Pass `allow_pickle=False` to raise an error
instead.
You have disabled the safety checker for <class 'diffusers.pipelines.stable_diff
usion.pipeline_stable_diffusion_inpaint.StableDiffusionInpaintPipeline'> by
passing `safety_checker= ... [486 chars truncated]
```

30.6. 2.3 Preprocessing

Get an image and the segmentation masks from the previously loaded Pascal VOC dataset.

```
idx = 100 # choose an image

image_pil, mask_pil = dataset[idx]
img_np = np.array(image_pil)
mask_np = np.array(mask_pil)
h, w = img_np.shape[:2]
```

Creating the binary mask.

```
labels = np.unique(mask_np)
obj_labels = labels[labels != 0]
if len(obj_labels) == 0:
    raise ValueError(f"No objects found in segmentation mask for index {idx}")
chosen_label = random.choice(obj_labels)
mask = (mask_np == chosen_label).astype(np.uint8) * 255
if mask.shape[:2] != (h, w):
    mask = cv2.resize(mask, (w, h), interpolation=cv2.INTER_NEAREST)
torch_mask_vis = cv2.cvtColor(mask, cv2.COLOR_GRAY2RGB)
plt.imshow(torch_mask_vis)
plt.title("Mask Preview")
plt.axis('off')
```

```
(np.float64(-0.5), np.float64(332.5), np.float64(499.5), np.float64(-0.5))
```




```
target_dim = 512
scale = min(target_dim / w, target_dim / h)
new_w, new_h = int(w * scale), int(h * scale)
image_resized = image_pil.resize((new_w, new_h), resample=Image.LANCZOS)
mask_resized = Image.fromarray(mask).resize((new_w, new_h), resample=Image.NEAREST) # nearest for mask to
   preserve edges
```

30.7. 2.4 Hyperparameters

Two main hyperparameters influence output:

- **num_inference_steps**: number of diffusion steps . More steps generally improve detail at the cost of speed.
- **guidance_scale**: strength of adherence to text prompt . Higher values make output more aligned to prompt but risk artifacts.

Feel free to experiment:

```
num_steps = 40
guidance = 10
```

30.8. 2.5 Running the pipeline

We also has to define the prompts.

```
prompts = [
    "Baby in a classic navy-blue sailor suit with white collar and cap, seated politely in a chair, hands on
   the sides , high-detail portrait",
    "A golden-doodle puppy tilting its head, tongue out, fluffy coat, natural daylight, high-resolution
   portrait",
    "Five helium balloons (pastel pink, mint, lavender, yellow, sky blue) tied together, floating just above
   the seat, soft backlight."
]

inpainted_results = []
for txt in prompts:
    result = pipe(
        prompt=txt,
        image=image_resized,
        mask_image=mask_resized,
        num_inference_steps=num_steps,
        guidance_scale=guidance
    ).images[0]
    inpainted_results.append((txt, result))
```

```
0%|          | 0/40 [00:00<?, ?it/s]
```

```
0%|          | 0/40 [00:00<?, ?it/s]
```

```
0%|          | 0/40 [00:00<?, ?it/s]
```

30.9. 2.6 Visualization

Display the original image overlaid with the mask, then each inpainted image side by side. Observe how the same region adapts to different prompts.

```
# Overlay mask in semi-transparent red for reference
overlay = img_np.copy()
red_overlay = np.array([255, 0, 0], dtype=np.uint8)
overlay[mask > 0] = (0.5 * overlay[mask > 0] + 0.5 * red_overlay).astype(np.uint8)

total = len(inpainted_results) + 1
plt.figure(figsize=(4 * total, 4))

plt.subplot(1, total, 1)
plt.imshow(overlay)
plt.title("Original + Mask")
plt.axis('off')

def wrap_title(s, width=20):
    return "\n".join(textwrap.wrap(s, width=width))

for idx, (txt, img) in enumerate(inpainted_results, start=2):
    plt.subplot(1, total, idx)
    plt.imshow(np.array(img))
    plt.title(wrap_title(txt, width=20))
    plt.axis('off')

plt.tight_layout()
plt.show()
```



30.10. Homework (Optional)

1. **Mask-tightness study.** For one image, create three masks: tight (just the object), medium (object + 10 px margin), loose (object + 50 px margin). Run the same prompt against all three and discuss the quality difference.
2. **Prompt engineering.** Pick one simple source mask (e.g. an empty table surface) and try 5 prompts of increasing specificity. Rank the outputs by subjective quality and write one sentence per prompt on why it worked (or did not).
3. **Seed variance.** For a fixed mask + prompt, generate 8 outputs with different random seeds. Pick the best and the worst and compare. What is the range of quality you get without changing any other input?
4. **Negative prompts.** Add `"blurry, low quality, text, watermark"` as a negative prompt and compare against the same positive prompt without one. Quantify the difference in generation quality by looking at 5 seeds per configuration.
5. **Iterative masking.** Take the inpainting output and use it as the input for a second inpainting pass with a different mask and prompt. Build up a composite image through 2-3 rounds and describe failures (drift, boundary artefacts).
6. **Compare pipelines.** Swap the default Stable Diffusion Inpainting with `stabilityai/stable-diffusion-2-inpainting` and `runwayml/stable-diffusion-inpainting`. Report qualitative differences and inference latency.

Chapter 31

Generative Modelling

Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/13eSW0bXfqzeA6SuDUHMMCi95YkSJMesd?usp=sharing

Generative modelling is a branch of machine learning focused on **learning to generate new data samples** that resemble the training distribution. This notebook explores key generative architectures including **Autoencoders (AE)**, **Variational Autoencoders (VAE)**, **Generative Adversarial Networks (GANs)**, and **Diffusion Models**, demonstrating how neural networks can learn to create realistic images from learned latent representations.

31.1. Necessary Imports

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from torchvision.utils import make_grid

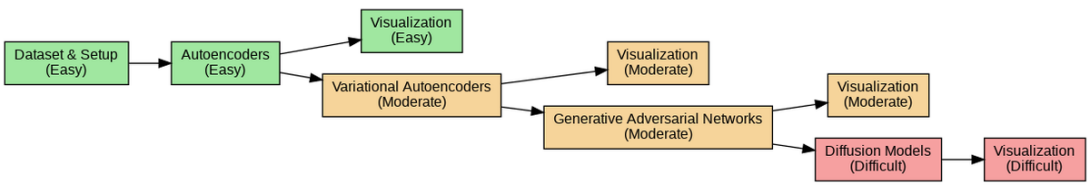
import matplotlib.pyplot as plt
import numpy as np

from diffusers import StableDiffusionPipeline
from PIL import Image
from tqdm.auto import tqdm
from IPython.display import Image as IMG
from sklearn.manifold import TSNE

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
print(f"Using device: {device}")
```

```
Using device: cuda:0
```

```
IMG('generative_modelling_flowchart.png')
```



31.2. Loading Our Dataset

The **Fashion MNIST [3]** dataset is a drop-in replacement for MNIST [2] containing 70,000 grayscale images of clothing items. We normalize the images to the range [-1, 1] to match the output of Tanh activation.

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_dataset = torchvision.datasets.FashionMNIST(
    root='./data', train=True, download=True, transform=transform
)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)

test_dataset = torchvision.datasets.FashionMNIST(
    root='./data', train=False, download=True, transform=transform
)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

print(f"\nFashion MNIST dataset loaded.")
print(f"Training samples: {len(train_dataset)}")
print(f"Test samples: {len(test_dataset)}")
```

```
[... 3 progress-bar updates ...]
100%|##########| 5.15k/5.15k [00:00<00:00, 27.6MB/s]
```

```
Fashion MNIST dataset loaded.
Training samples: 60000
Test samples: 10000
```

```
def show_images(images, title=''):
    images = images * 0.5 + 0.5  # Denormalize from [-1, 1] to [0, 1]
    grid_img = make_grid(images.cpu(), nrow=8)
    plt.figure(figsize=(10, 10))
    plt.imshow(grid_img.permute(1, 2, 0))
    plt.title(title, fontsize=16)
    plt.axis('off')
    plt.show()

dataiter = iter(train_loader)
images, labels = next(dataiter)
show_images(images, 'Sample Fashion MNIST Images')
```



31.3. 2. Autoencoders

Autoencoders are **unsupervised neural networks** designed to learn efficient representations of data by compressing inputs into a low-dimensional **latent space** and reconstructing them.

31.3.1. 2.1 Understanding the Architecture

- **Encoder**: Compresses the input into a latent vector, learning essential features while discarding redundant information
- **Decoder**: Reconstructs the input from the latent vector, attempting to reproduce the original input as accurately as possible

31.3.2. 2.2 Autoencoder Implementation

The encoder flattens and progressively reduces the input dimensionality through linear layers, outputting a small latent vector (default size = 2). The decoder mirrors this structure, expanding the latent vector back to the original image dimensions using **Tanh** activation to match normalized data in the range [-1, 1].

```
class ConvEncoder(nn.Module):
    """Convolutional encoder for Autoencoder."""

    def __init__(self, latent_dim=32, hidden_dims=(32, 64, 128), dropout_rate=0.1):
        super().__init__()
        self.latent_dim = latent_dim

        # Build convolutional layers
        modules = []
        in_channels = 1

        for h_dim in hidden_dims:
            modules.append(nn.Sequential(
                nn.Conv2d(in_channels, h_dim, kernel_size=3, stride=2, padding=1),
                nn.BatchNorm2d(h_dim),
                nn.LeakyReLU(0.2, inplace=True),
                nn.Dropout2d(dropout_rate)
            ))
            in_channels = h_dim

        self.conv_layers = nn.Sequential(*modules)

        # For 28x28 input with 3 conv layers (stride 2): 28 -> 14 -> 7 -> 4
        self.flatten_size = hidden_dims[-1] * 4 * 4
        self.fc = nn.Linear(self.flatten_size, latent_dim)

    def forward(self, x):
        h = self.conv_layers(x)
        h = h.view(h.size(0), -1)
        z = self.fc(h)
        return z

class ConvDecoder(nn.Module):
    """Convolutional decoder for Autoencoder."""

    def __init__(self, latent_dim=32, hidden_dims=(32, 64, 128)):
        super().__init__()
        self.hidden_dims = hidden_dims

        # Project from latent space
        self.fc = nn.Linear(latent_dim, hidden_dims[-1] * 7 * 7)

        # Build transposed convolutional layers
        modules = []
        reversed_dims = list(reversed(hidden_dims))

        for i in range(len(reversed_dims) - 1):
            modules.append(nn.Sequential(
                nn.ConvTranspose2d(reversed_dims[i], reversed_dims[i + 1],
                                   kernel_size=3, stride=1, padding=1),
                nn.BatchNorm2d(reversed_dims[i + 1]),
                nn.LeakyReLU(0.2, inplace=True)
            ))
```

```
        # Final layer to get back to 1 channel
        modules.append(nn.Sequential(
            nn.ConvTranspose2d(reversed_dims[-1], reversed_dims[-1],
                               kernel_size=4, stride=2, padding=1),
        ))
        modules.append(nn.Sequential(
            nn.ConvTranspose2d(reversed_dims[-1], 1,
                               kernel_size=4, stride=2, padding=1),
            nn.Tanh()
        ))

        self.conv_layers = nn.Sequential(*modules)

    def forward(self, z):
        h = self.fc(z)
        h = h.view(h.size(0), self.hidden_dims[-1], 7, 7)
        x_recon = self.conv_layers(h)
        return x_recon

class Autoencoder(nn.Module):
    def __init__(self, latent_dim=32, hidden_dims=(32, 64, 128)):
        super().__init__()
        self.latent_dim = latent_dim
        self.encoder = ConvEncoder(latent_dim, hidden_dims)
        self.decoder = ConvDecoder(latent_dim, hidden_dims)

        self.apply(self._init_weights)

    def _init_weights(self, m):
        if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d, nn.Linear)):
            nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='leaky_relu')
            if m.bias is not None:
                nn.init.constant_(m.bias, 0)
        elif isinstance(m, nn.BatchNorm2d):
            nn.init.constant_(m.weight, 1)
            nn.init.constant_(m.bias, 0)

    def encode(self, x):
        return self.encoder(x)

    def decode(self, z):
        return self.decoder(z)

    def forward(self, x):
        z = self.encode(x)
        x_recon = self.decode(z)
        return x_recon, z

    def count_parameters(self):
        return sum(p.numel() for p in self.parameters() if p.requires_grad)

class EarlyStopping:
    """Early stopping to prevent overfitting."""

    def __init__(self, patience=10, min_delta=1e-4, verbose=True):
        self.patience = patience
        self.min_delta = min_delta
        self.verbose = verbose
        self.counter = 0
```

```
        self.best_score = None
        self.early_stop = False
        self.best_epoch = 0

    def __call__(self, score, epoch):
        is_improvement = self.best_score is None or score < self.best_score - self.min_delta

        if is_improvement:
            if self.verbose and self.best_score is not None:
                print(f"Validation loss improved: {self.best_score:.4f} → {score:.4f}")
            self.best_score = score
            self.best_epoch = epoch
            self.counter = 0
        else:
            self.counter += 1
            if self.verbose:
                print(f"No improvement for {self.counter}/{self.patience} epochs")
            if self.counter >= self.patience:
                self.early_stop = True

        return self.early_stop
```

```
LATENT_DIM = 32
HIDDEN_DIMS = (32, 64, 128)

model_ae = Autoencoder(latent_dim=LATENT_DIM, hidden_dims=HIDDEN_DIMS).to(device)
print(model_ae)
print(f"\nTotal parameters: {model_ae.count_parameters():,}")
```

```
Autoencoder(
  (encoder): ConvEncoder(
    (conv_layers): Sequential(
      (0): Sequential(
        (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2, inplace=True)
        (3): Dropout2d(p=0.1, inplace=False)
      )
      (1): Sequential(
        (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2, inplace=True)
        (3): Dropout2d(p=0.1, inplace=False)
      )
      (2): Sequential(
        (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2, inplace=True)
        (3): Dropout2d(p=0.1, inplace=False)
[... 19 lines omitted ...]
      )
      (3): Sequential(
        (0): ConvTranspose2d(32, 1, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
        (1): Tanh()
      )
    )
```

```
  )
)

Total parameters: 475,041
```

31.3.3. 2.3 Training the Autoencoder

We train the autoencoder using **MSE loss** with a training loop featuring: - **Early stopping** with patience tracking - **Learning rate scheduling** with ReduceLROnPlateau - **Gradient clipping** for training stability - **Best model checkpointing**

```
# Hyperparameters
NUM_EPOCHS = 30
LEARNING_RATE = 1e-3
WEIGHT_DECAY = 1e-5
GRAD_CLIP = 1.0
PATIENCE = 10

# Loss function
criterion = nn.MSELoss()

# Optimizer with weight decay
optimizer = optim.AdamW(model_ae.parameters(), lr=LEARNING_RATE, weight_decay=WEIGHT_DECAY)

# Learning rate scheduler
scheduler = optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='min', factor=0.5, patience=5, min_lr=1e-6
)

# Early stopping
early_stopper = EarlyStopping(patience=PATIENCE, min_delta=1e-4, verbose=True)

# Training history
ae_history = {'train_loss': [], 'val_loss': [], 'learning_rate': []}
best_val_loss = float('inf')
best_model_state = None

print(f"Model parameters: {model_ae.count_parameters():,}")
print(f"Latent dimension: {LATENT_DIM}")
print(f"Device: {device}")

for epoch in range(NUM_EPOCHS):
    print(f"\nEpoch {epoch + 1}/{NUM_EPOCHS}")
    print("-" * 40)

    # Training phase
    model_ae.train()
    train_loss = 0.0

    train_pbar = tqdm(train_loader, desc="Training", leave=False)
    for img, _ in train_pbar:
        img = img.to(device)

        # Forward pass
        recon, z = model_ae(img)
        loss = criterion(recon, img)

        # Backward pass
        optimizer.zero_grad()
```

```
        loss.backward()

        # Gradient clipping
        torch.nn.utils.clip_grad_norm_(model_ae.parameters(), GRAD_CLIP)

        optimizer.step()

        train_loss += loss.item()
        train_pbar.set_postfix({'loss': f'{loss.item():.4f}'})

    train_loss /= len(train_loader)

    # Validation phase
    model_ae.eval()
    val_loss = 0.0

    with torch.no_grad():
        for img, _ in test_loader:
            img = img.to(device)
            recon, z = model_ae(img)
            loss = criterion(recon, img)
            val_loss += loss.item()

    val_loss /= len(test_loader)
    current_lr = optimizer.param_groups[0]['lr']

    # Update history
    ae_history['train_loss'].append(train_loss)
    ae_history['val_loss'].append(val_loss)
    ae_history['learning_rate'].append(current_lr)

    print(f"  Train Loss: {train_loss:.4f}")
    print(f"  Val Loss:   {val_loss:.4f}")
    print(f"  LR:         {current_lr:.2e}")

    # Learning rate scheduling
    scheduler.step(val_loss)

    # Save best model
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        best_model_state = model_ae.state_dict().copy()

    # Early stopping
    if early_stopper(val_loss, epoch):
        print(f"\n! Early stopping triggered at epoch {epoch + 1}")
        print(f"  Best epoch: {early_stopper.best_epoch + 1}")
        print(f"  Best val loss: {early_stopper.best_score:.4f}")
        break

# Restore best model
if best_model_state is not None:
    model_ae.load_state_dict(best_model_state)

print(f"Best validation loss: {best_val_loss:.4f}")
```

```
Model parameters: 475,041
Latent dimension: 32
Device: cuda:0
```

```
Epoch 1/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0728
  Val Loss:   0.0460
  LR:         1.00e-03

Epoch 2/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0465
  Val Loss:   0.0396
  LR:         1.00e-03
Validation loss improved: 0.0460 -> 0.0396

Epoch 3/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0417
  Val Loss:   0.0371
  LR:         1.00e-03
Validation loss improved: 0.0396 -> 0.0371

Epoch 4/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0391
  Val Loss:   0.0355
  LR:         1.00e-03
Validation loss improved: 0.0371 -> 0.0355

Epoch 5/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0375
  Val Loss:   0.0344
  LR:         1.00e-03
Validation loss improved: 0.0355 -> 0.0344

Epoch 6/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0363
  Val Loss:   0.0335
  LR:         1.00e-03
Validation loss improved: 0.0344 -> 0.0335

Epoch 7/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0354
  Val Loss:   0.0330
  LR:         1.00e-03
Validation loss improved: 0.0335 -> 0.0330

Epoch 8/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0347
  Val Loss:   0.0327
  LR:         1.00e-03
Validation loss improved: 0.0330 -> 0.0327

Epoch 9/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0341
  Val Loss:   0.0320
  LR:         1.00e-03
Validation loss improved: 0.0327 -> 0.0320

Epoch 10/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0336
  Val Loss:   0.0315
  LR:         1.00e-03
Validation loss improved: 0.0320 -> 0.0315

Epoch 11/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0332
  Val Loss:   0.0313
  LR:         1.00e-03
Validation loss improved: 0.0315 -> 0.0313

Epoch 12/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0329
  Val Loss:   0.0311
  LR:         1.00e-03
Validation loss improved: 0.0313 -> 0.0311

Epoch 13/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0326
  Val Loss:   0.0308
  LR:         1.00e-03
Validation loss improved: 0.0311 -> 0.0308

Epoch 14/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0323
  Val Loss:   0.0307
  LR:         1.00e-03
Validation loss improved: 0.0308 -> 0.0307

Epoch 15/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0320
  Val Loss:   0.0309
  LR:         1.00e-03
No improvement for 1/10 epochs

Epoch 16/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0318
  Val Loss:   0.0303
  LR:         1.00e-03
Validation loss improved: 0.0307 -> 0.0303

Epoch 17/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0316
  Val Loss:   0.0299
  LR:         1.00e-03
Validation loss improved: 0.0303 -> 0.0299

Epoch 18/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0314
  Val Loss:   0.0299
  LR:         1.00e-03
No improvement for 1/10 epochs

Epoch 19/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0313
  Val Loss:   0.0298
  LR:         1.00e-03
No improvement for 2/10 epochs

Epoch 20/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0311
  Val Loss:   0.0299
  LR:         1.00e-03
No improvement for 3/10 epochs

Epoch 21/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0310
  Val Loss:   0.0295
  LR:         1.00e-03
Validation loss improved: 0.0299 -> 0.0295

Epoch 22/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0308
  Val Loss:   0.0298
  LR:         1.00e-03
No improvement for 1/10 epochs

Epoch 23/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0307
  Val Loss:   0.0293
  LR:         1.00e-03
Validation loss improved: 0.0295 -> 0.0293

Epoch 24/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0306
  Val Loss:   0.0292
  LR:         1.00e-03
Validation loss improved: 0.0293 -> 0.0292

Epoch 25/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0304
  Val Loss:   0.0291
  LR:         1.00e-03
No improvement for 1/10 epochs

Epoch 26/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0304
  Val Loss:   0.0291
  LR:         1.00e-03
No improvement for 2/10 epochs

Epoch 27/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0302
  Val Loss:   0.0291
  LR:         1.00e-03
Validation loss improved: 0.0292 -> 0.0291

Epoch 28/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0302
  Val Loss:   0.0288
  LR:         1.00e-03
Validation loss improved: 0.0291 -> 0.0288

Epoch 29/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 0.0301
  Val Loss:   0.0289
  LR:         1.00e-03
No improvement for 1/10 epochs

Epoch 30/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

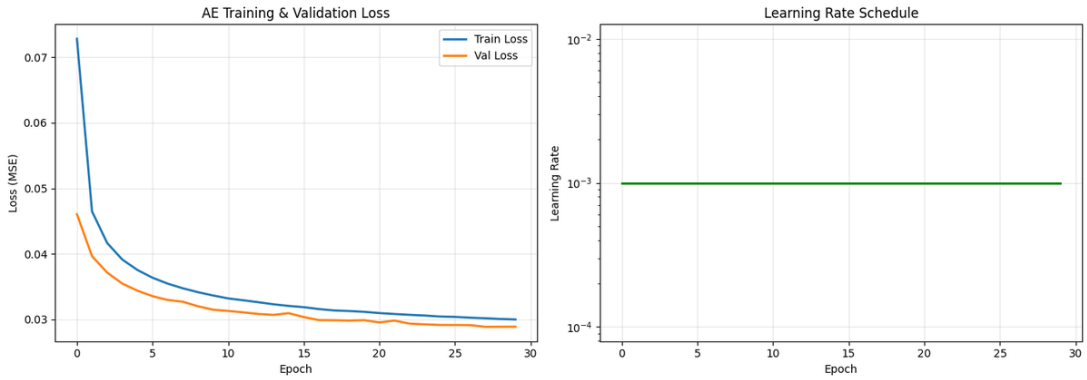
```
Train Loss: 0.0300
  Val Loss:   0.0289
  LR:         1.00e-03
No improvement for 2/10 epochs
Best validation loss: 0.0288
```

```
# Plot training curves
fig, axes = plt.subplots(1, 2, figsize=(14, 5))
```

```
# Loss curves
ax1 = axes[0]
ax1.plot(ae_history['train_loss'], label='Train Loss', linewidth=2)
ax1.plot(ae_history['val_loss'], label='Val Loss', linewidth=2)
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Loss (MSE)')
ax1.set_title('AE Training & Validation Loss')
ax1.legend()
ax1.grid(True, alpha=0.3)

# Learning rate
ax2 = axes[1]
ax2.plot(ae_history['learning_rate'], linewidth=2, color='green')
ax2.set_xlabel('Epoch')
ax2.set_ylabel('Learning Rate')
ax2.set_title('Learning Rate Schedule')
ax2.set_yscale('log')
ax2.grid(True, alpha=0.3)

plt.tight_layout()
plt.show()
```



31.3.4. 2.4 Visualization & Latent Space Analysis

We evaluate the autoencoder by visualizing: 1. **Image Reconstruction**: Comparing original vs. reconstructed images 2. **Latent Space**: A 2D scatter plot showing how different classes are organized in the learned representation

```
model_ae.eval()

# Reconstruction visualization
with torch.no_grad():
    images, labels = next(iter(test_loader))
    images = images[:16].to(device)
    recon, _ = model_ae(images)

    # Denormalize
    images_show = images * 0.5 + 0.5
    recon_show = recon * 0.5 + 0.5

fig, axes = plt.subplots(4, 8, figsize=(16, 8))
```

```
for i in range(16):
    # Original
    ax = axes[i // 8 * 2, i % 8]
    ax.imshow(images_show[i, 0].cpu().numpy(), cmap='gray')
    ax.axis('off')
    if i == 0:
        ax.set_title('Original', fontsize=10)

    # Reconstructed
    ax = axes[i // 8 * 2 + 1, i % 8]
    ax.imshow(recon_show[i, 0].cpu().numpy(), cmap='gray')
    ax.axis('off')
    if i == 0:
        ax.set_title('Reconstructed', fontsize=10)

plt.suptitle('Autoencoder Reconstructions', fontsize=14)
plt.tight_layout()
plt.show()

with torch.no_grad():
    latent_vectors = []
    all_labels = []
    for images, labels in test_loader:
        images = images.to(device)
        z = model_ae.encode(images)
        latent_vectors.append(z.cpu().numpy())
        all_labels.append(labels.numpy())

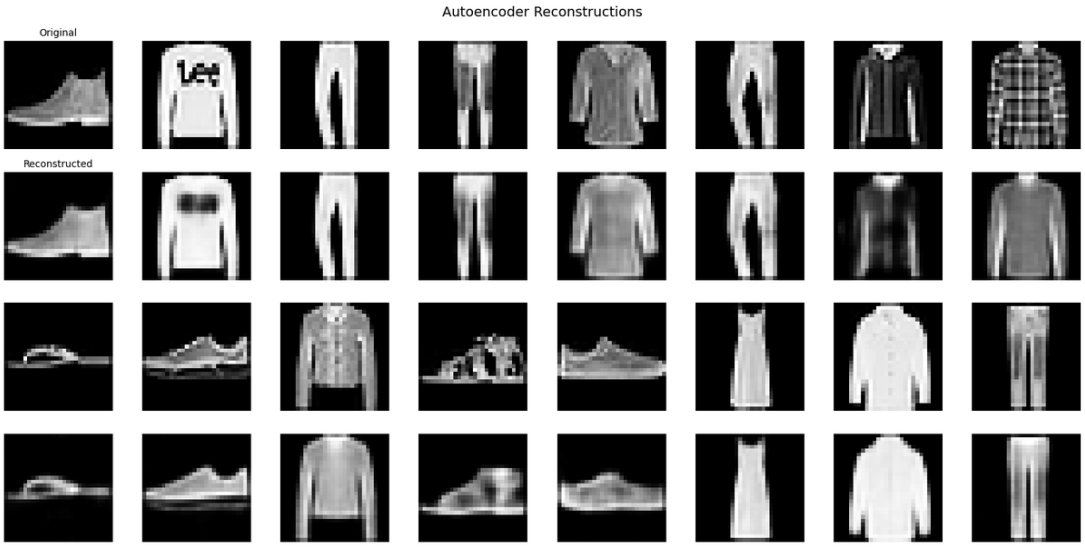
latent_vectors = np.concatenate(latent_vectors)
all_labels = np.concatenate(all_labels)

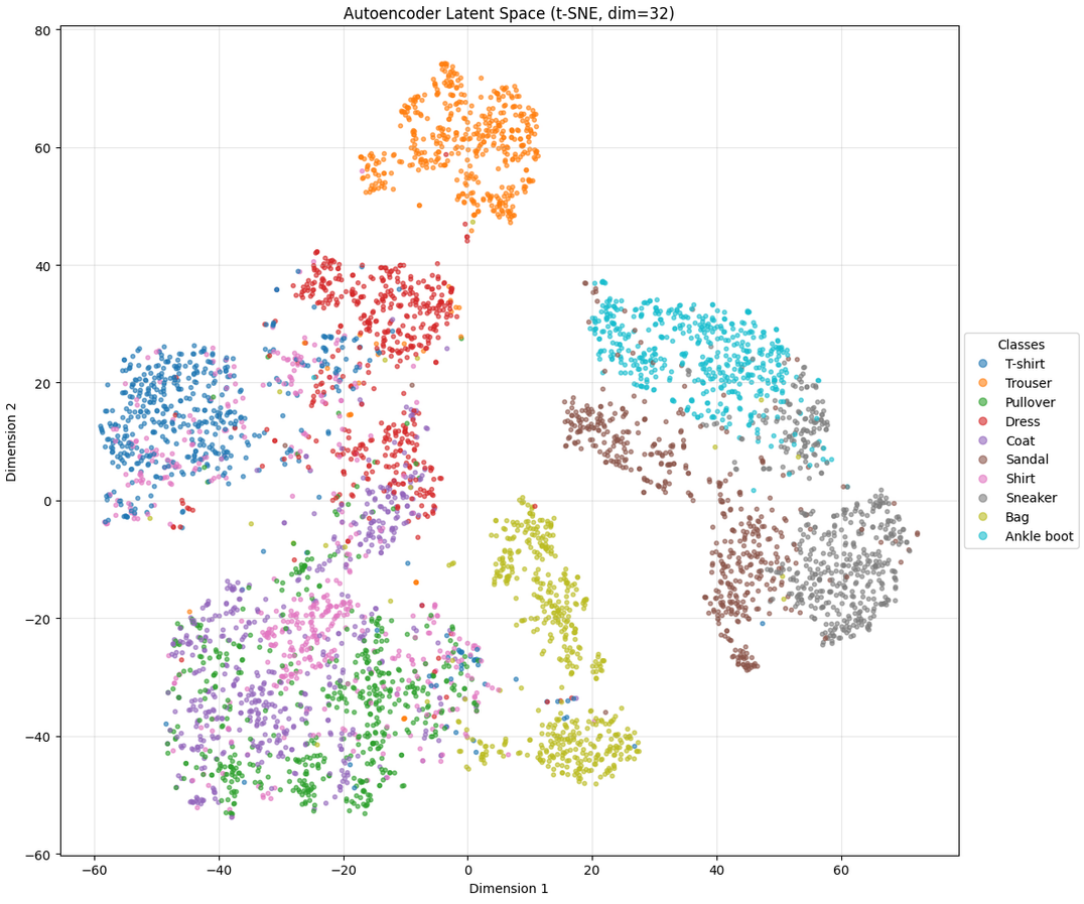
# Apply t-SNE for high-dimensional latent space
if latent_vectors.shape[1] > 2:
    tsne = TSNE(n_components=2, random_state=42, perplexity=30)
    latent_2d = tsne.fit_transform(latent_vectors[:5000])
    labels_2d = all_labels[:5000]
else:
    latent_2d = latent_vectors
    labels_2d = all_labels

plt.figure(figsize=(12, 10))
class_names = ['T-shirt', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

scatter = plt.scatter(latent_2d[:, 0], latent_2d[:, 1],
                      c=labels_2d, cmap='tab10', alpha=0.6, s=10)

handles = scatter.legend_elements()[0]
plt.legend(handles, class_names, title="Classes", loc='center left', bbox_to_anchor=(1, 0.5))
plt.title(f'Autoencoder Latent Space (t-SNE, dim={LATENT_DIM})')
plt.xlabel('Dimension 1')
plt.ylabel('Dimension 2')
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()
```





31.3.5. 2.5 Latent Space Interpolation

We demonstrate how the autoencoder's latent space captures smooth semantic transitions by interpolating between two different classes (T-shirt and Ankle boot).

```
model_ae.eval()

# Get samples from T-shirt (0) and Ankle boot (9)
idx_a = np.where(np.array(test_dataset.targets) == 0)[0][0]
idx_b = np.where(np.array(test_dataset.targets) == 9)[0][0]

x_a = test_dataset[idx_a][0].unsqueeze(0).to(device)
x_b = test_dataset[idx_b][0].unsqueeze(0).to(device)

with torch.no_grad():
    z_a = model_ae.encode(x_a)
    z_b = model_ae.encode(x_b)

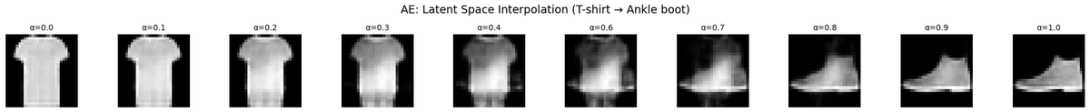
    # Interpolate
    n_steps = 10
    interpolated_images = []
    for alpha in np.linspace(0, 1, n_steps):
        z_interp = (1 - alpha) * z_a + alpha * z_b
        x_interp = model_ae.decode(z_interp)
        interpolated_images.append(x_interp)

    interpolated_images = torch.cat(interpolated_images)
    interpolated_images = interpolated_images * 0.5 + 0.5  # Denormalize

fig, axes = plt.subplots(1, n_steps, figsize=(2 * n_steps, 2))

for i, ax in enumerate(axes):
    ax.imshow(interpolated_images[i, 0].cpu().numpy(), cmap='gray')
    ax.axis('off')
    ax.set_title(f'α={i/(n_steps-1):.1f}', fontsize=10)

plt.suptitle('AE: Latent Space Interpolation (T-shirt → Ankle boot)', fontsize=14)
plt.tight_layout()
plt.show()
```



31.4. 3. Variational Autoencoders (VAE)

Variational Autoencoders are **probabilistic extensions** of autoencoders that learn a continuous latent space suitable for generation.

31.4.1. 3.1 Key Concepts

- **Probabilistic Encoder**: Maps input to a distribution (mean μ and log-variance $\log \sigma^2$) instead of a fixed vector
- **Reparameterization Trick**: Enables backpropagation through stochastic sampling using $z = \mu + \sigma \cdot \epsilon$ where $\epsilon \sim \mathcal{N}(0, 1)$

- **Loss Function**: Combines reconstruction loss with KL divergence to regularize the latent space toward a standard normal distribution

31.4.2. 3.2 VAE Implementation

The VAE encoder outputs both mean and log-variance parameters, enabling sampling from the learned distribution. The reparameterization trick allows gradients to flow through the stochastic sampling step.

```
class VAEEncoder(nn.Module):
    """Convolutional encoder for VAE with separate μ and log(σ²) heads."""

    def __init__(self, latent_dim=32, hidden_dims=(32, 64, 128)):
        super().__init__()

        # Build convolutional layers
        modules = []
        in_channels = 1

        for h_dim in hidden_dims:
            modules.append(nn.Sequential(
                nn.Conv2d(in_channels, h_dim, kernel_size=3, stride=2, padding=1),
                nn.BatchNorm2d(h_dim),
                nn.LeakyReLU(0.2, inplace=True)
            ))
            in_channels = h_dim

        self.conv_layers = nn.Sequential(*modules)

        # For 28x28 input: 28 -> 14 -> 7 -> 4
        self.flatten_size = hidden_dims[-1] * 4 * 4

        # Separate heads for μ and log(σ²)
        self.fc_mu = nn.Linear(self.flatten_size, latent_dim)
        self.fc_logvar = nn.Linear(self.flatten_size, latent_dim)

    def forward(self, x):
        h = self.conv_layers(x)
        h = h.view(h.size(0), -1)
        mu = self.fc_mu(h)
        logvar = self.fc_logvar(h)
        # Clamp logvar for numerical stability
        logvar = torch.clamp(logvar, min=-10, max=10)
        return mu, logvar

class VAEDecoder(nn.Module):
    """Convolutional decoder for VAE."""

    def __init__(self, latent_dim=32, hidden_dims=(32, 64, 128)):
        super().__init__()
        self.hidden_dims = hidden_dims

        self.fc = nn.Linear(latent_dim, hidden_dims[-1] * 4 * 4)

        modules = []
        reversed_dims = list(reversed(hidden_dims))

        for i in range(len(reversed_dims) - 1):
            modules.append(nn.Sequential(
                nn.ConvTranspose2d(reversed_dims[i], reversed_dims[i + 1],
```

```
                                    kernel_size=3, stride=2, padding=1, output_padding=1),
                nn.BatchNorm2d(reversed_dims[i + 1]),
                nn.LeakyReLU(0.2, inplace=True)
            ))

        modules.append(nn.Sequential(
            nn.ConvTranspose2d(reversed_dims[-1], 1,
                               kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.Tanh()
        ))

        self.conv_layers = nn.Sequential(*modules)

    def forward(self, z):
        h = self.fc(z)
        h = h.view(h.size(0), self.hidden_dims[-1], 4, 4)
        x_recon = self.conv_layers(h)
        x_recon = x_recon[:, :, :28, :28]
        return x_recon

class VariationalAutoencoder(nn.Module):
    """β-VAE with reparameterization trick."""

    def __init__(self, latent_dim=32, hidden_dims=(32, 64, 128)):
        super().__init__()
        self.latent_dim = latent_dim
        self.encoder = VAEEncoder(latent_dim, hidden_dims)
        self.decoder = VAEDecoder(latent_dim, hidden_dims)

        self.apply(self._init_weights)

    def _init_weights(self, m):
        if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d)):
            nn.init.xavier_uniform_(m.weight)
            if m.bias is not None:
                nn.init.constant_(m.bias, 0)
        elif isinstance(m, nn.Linear):
            nn.init.xavier_uniform_(m.weight)
            if m.bias is not None:
                nn.init.constant_(m.bias, 0)
        elif isinstance(m, nn.BatchNorm2d):
            nn.init.constant_(m.weight, 1)
            nn.init.constant_(m.bias, 0)

    def reparameterize(self, mu, logvar):
        """Reparameterization trick: z = μ + σ * ε where ε ~ N(0, I)"""
        # Clamp logvar for numerical stability
        logvar = torch.clamp(logvar, min=-10, max=10)
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def encode(self, x):
        mu, logvar = self.encoder(x)
        return mu, logvar

    def decode(self, z):
        return self.decoder(z)

    def forward(self, x):
```

```
        mu, logvar = self.encode(x)
        z = self.reparameterize(mu, logvar)
        x_recon = self.decode(z)
        return x_recon, mu, logvar

    def sample(self, n_samples, device):
        """Generate samples from the prior N(0, I)."""
        z = torch.randn(n_samples, self.latent_dim, device=device)
        return self.decode(z)

    def count_parameters(self):
        return sum(p.numel() for p in self.parameters() if p.requires_grad)

def vae_loss(x_recon, x, mu, logvar, beta=1.0):
    """
    β-VAE loss = Reconstruction Loss + β * KL Divergence

    Using MSE for reconstruction (appropriate for Tanh output with [-1, 1] normalized data)
    """
    # Reconstruction loss (MSE) - use mean reduction for stability
    recon_loss = F.mse_loss(x_recon, x, reduction='mean') * x.shape[1] * x.shape[2] * x.shape[3]

    # KL Divergence: D_KL(q(z|x) || p(z)) = -0.5 * Σ(1 + log(σ²) - μ² - σ²)
    # Clamp logvar for numerical stability
    logvar = torch.clamp(logvar, min=-10, max=10)
    kl_loss = -0.5 * torch.mean(torch.sum(1 + logvar - mu.pow(2) - logvar.exp(), dim=1))

    # Ensure losses are not NaN
    if torch.isnan(recon_loss) or torch.isnan(kl_loss):
        print("Warning: NaN detected in loss!")
        recon_loss = torch.tensor(0.0, device=x.device, requires_grad=True)
        kl_loss = torch.tensor(0.0, device=x.device, requires_grad=True)

    return recon_loss + beta * kl_loss, recon_loss, kl_loss
```

```
VAE_LATENT_DIM = 32
VAE_HIDDEN_DIMS = (32, 64, 128)

model_vae = VariationalAutoencoder(latent_dim=VAE_LATENT_DIM, hidden_dims=VAE_HIDDEN_DIMS).to(device)
print(model_vae)
print(f"\nTotal parameters: {model_vae.count_parameters():,}")
```

```
VariationalAutoencoder(
  (encoder): VAEEncoder(
    (conv_layers): Sequential(
      (0): Sequential(
        (0): Conv2d(1, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2, inplace=True)
      )
      (1): Sequential(
        (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2, inplace=True)
      )
      (2): Sequential(
```

```
        (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
        (2): LeakyReLU(negative_slope=0.2, inplace=True)
      )
    )
    (fc_mu): Linear(in_features=2048, out_features=32, bias=True)
[... 14 lines omitted ...]
      )
      (2): Sequential(
        (0): ConvTranspose2d(32, 1, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), output_padding=(1, 1))
        (1): Tanh()
      )
    )
  )
)

Total parameters: 384,577
```

31.4.3. 3.3 Training the VAE

The VAE is trained using a combined loss of reconstruction error (MSE) and KL divergence, which regularizes the latent space to follow a standard normal distribution.

```
# Hyperparameters
VAE_EPOCHS = 30
VAE_LR = 1e-3
BETA_TARGET = 4.0  # Target β value (>1 encourages disentanglement)
BETA_WARMUP = 5    # Epochs to reach target β
VAE_PATIENCE = 10
VAE_GRAD_CLIP = 1.0

# Optimizer and scheduler
vae_optimizer = optim.AdamW(model_vae.parameters(), lr=VAE_LR, weight_decay=1e-5)
vae_scheduler = optim.lr_scheduler.ReduceLROnPlateau(
    vae_optimizer, mode='min', factor=0.5, patience=5, min_lr=1e-6
)

# Early stopping
vae_early_stopper = EarlyStopping(patience=VAE_PATIENCE, min_delta=1e-4, verbose=True)

# Training history
vae_history = {
    'train_loss': [], 'val_loss': [],
    'recon_loss': [], 'kl_loss': [],
    'beta': [], 'learning_rate': []
}
best_vae_loss = float('inf')
best_vae_state = None

print(f"Model parameters: {model_vae.count_parameters():,}")
print(f"Latent dimension: {VAE_LATENT_DIM}")
print(f"Target β: {BETA_TARGET} (warmup: {BETA_WARMUP} epochs)")

for epoch in range(VAE_EPOCHS):
```

```
    print(f"\nEpoch {epoch + 1}/{VAE_EPOCHS}")
    print("-" * 40)

    # β-annealing: linearly increase β during warmup
    if epoch < BETA_WARMUP:
        current_beta = BETA_TARGET * (epoch + 1) / BETA_WARMUP
    else:
        current_beta = BETA_TARGET

    # Training phase
    model_vae.train()
    train_loss = 0.0
    train_recon = 0.0
    train_kl = 0.0

    train_pbar = tqdm(train_loader, desc="Training", leave=False)
    for img, _ in train_pbar:
        img = img.to(device)

        # Forward pass
        recon, mu, logvar = model_vae(img)
        loss, recon_l, kl_l = vae_loss(recon, img, mu, logvar, beta=current_beta)

        # Backward pass
        vae_optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model_vae.parameters(), VAE_GRAD_CLIP)
        vae_optimizer.step()

        train_loss += loss.item()
        train_recon += recon_l.item()
        train_kl += kl_l.item()

        train_pbar.set_postfix({'loss': f'{loss.item():.2f}', 'β': f'{current_beta:.2f}'})

    n_batches = len(train_loader)
    train_loss /= n_batches
    train_recon /= n_batches
    train_kl /= n_batches

    # Validation phase
    model_vae.eval()
    val_loss = 0.0
    val_recon = 0.0
    val_kl = 0.0

    with torch.no_grad():
        for img, _ in test_loader:
            img = img.to(device)
            recon, mu, logvar = model_vae(img)
            loss, recon_l, kl_l = vae_loss(recon, img, mu, logvar, beta=current_beta)
            val_loss += loss.item()
            val_recon += recon_l.item()
            val_kl += kl_l.item()

    n_val = len(test_loader)
    val_loss /= n_val
    val_recon /= n_val
    val_kl /= n_val
```

```
    current_lr = vae_optimizer.param_groups[0]['lr']

    # Update history
    vae_history['train_loss'].append(train_loss)
    vae_history['val_loss'].append(val_loss)
    vae_history['recon_loss'].append(val_recon)
    vae_history['kl_loss'].append(val_kl)
    vae_history['beta'].append(current_beta)
    vae_history['learning_rate'].append(current_lr)

    print(f"  Train Loss: {train_loss:.2f} | Val Loss: {val_loss:.2f}")
    print(f"  Recon: {val_recon:.2f} | KL: {val_kl:.2f} | β: {current_beta:.2f}")
    print(f"  LR: {current_lr:.2e}")

    # Learning rate scheduling
    vae_scheduler.step(val_loss)

    # Save best model
    if val_loss < best_vae_loss:
        best_vae_loss = val_loss
        best_vae_state = model_vae.state_dict().copy()

    # Early stopping
    if vae_early_stopper(val_loss, epoch):
        print(f"\n! Early stopping triggered at epoch {epoch + 1}")
        break

# Restore best model
if best_vae_state is not None:
    model_vae.load_state_dict(best_vae_state)

print(f"Best validation loss: {best_vae_loss:.2f}")
```

```
Model parameters: 384,577
Latent dimension: 32
Target β: 4.0 (warmup: 5 epochs)

Epoch 1/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 86.74 | Val Loss: 69.97
  Recon: 49.36 | KL: 25.77 | β: 0.80
  LR: 1.00e-03

Epoch 2/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 84.45 | Val Loss: 82.18
  Recon: 56.66 | KL: 15.95 | β: 1.60
  LR: 1.00e-03
No improvement for 1/10 epochs

Epoch 3/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 92.57 | Val Loss: 90.97
  Recon: 61.07 | KL: 12.46 | β: 2.40
  LR: 1.00e-03
No improvement for 2/10 epochs

Epoch 4/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 98.99 | Val Loss: 98.25
  Recon: 64.44 | KL: 10.57 | β: 3.20
  LR: 1.00e-03
No improvement for 3/10 epochs

Epoch 5/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 104.42 | Val Loss: 102.59
  Recon: 68.17 | KL: 8.61 | β: 4.00
  LR: 1.00e-03
No improvement for 4/10 epochs

Epoch 6/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 102.63 | Val Loss: 101.54
  Recon: 67.32 | KL: 8.56 | β: 4.00
  LR: 1.00e-03
No improvement for 5/10 epochs

Epoch 7/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 101.53 | Val Loss: 100.16
  Recon: 66.32 | KL: 8.46 | β: 4.00
  LR: 1.00e-03
No improvement for 6/10 epochs

Epoch 8/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 99.38 | Val Loss: 98.55
  Recon: 64.63 | KL: 8.48 | β: 4.00
  LR: 5.00e-04
No improvement for 7/10 epochs

Epoch 9/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 98.70 | Val Loss: 98.28
  Recon: 64.67 | KL: 8.40 | β: 4.00
  LR: 5.00e-04
No improvement for 8/10 epochs

Epoch 10/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 98.42 | Val Loss: 98.59
  Recon: 65.09 | KL: 8.38 | β: 4.00
  LR: 5.00e-04
No improvement for 9/10 epochs

Epoch 11/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Train Loss: 98.06 | Val Loss: 97.61
  Recon: 63.98 | KL: 8.41 | β: 4.00
  LR: 5.00e-04
No improvement for 10/10 epochs

! Early stopping triggered at epoch 11
Best validation loss: 69.97
```

```
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# Total loss
ax1 = axes[0, 0]
ax1.plot(vae_history['train_loss'], label='Train', linewidth=2)
ax1.plot(vae_history['val_loss'], label='Val', linewidth=2)
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Total Loss')
ax1.set_title('VAE Total Loss (Recon + β*KL)')
ax1.legend()
ax1.grid(True, alpha=0.3)

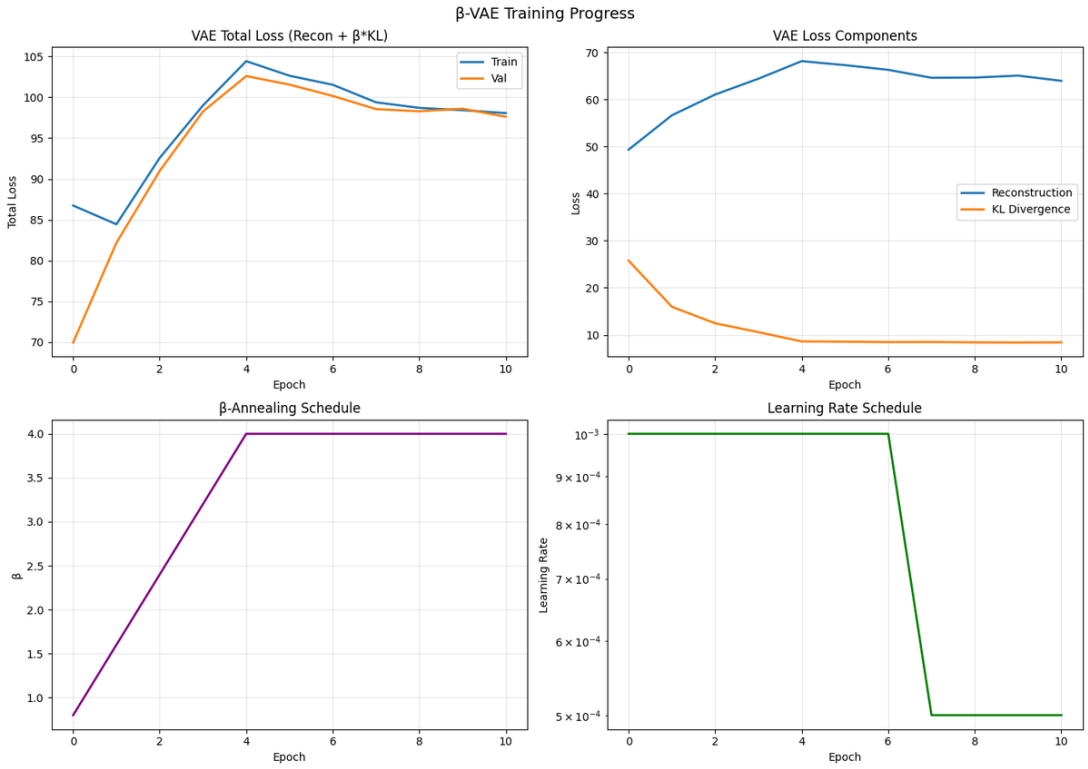
# Loss components
ax2 = axes[0, 1]
ax2.plot(vae_history['recon_loss'], label='Reconstruction', linewidth=2)
ax2.plot(vae_history['kl_loss'], label='KL Divergence', linewidth=2)
ax2.set_xlabel('Epoch')
ax2.set_ylabel('Loss')
ax2.set_title('VAE Loss Components')
ax2.legend()
ax2.grid(True, alpha=0.3)

# Beta annealing
ax3 = axes[1, 0]
ax3.plot(vae_history['beta'], linewidth=2, color='purple')
ax3.set_xlabel('Epoch')
ax3.set_ylabel('β')
ax3.set_title('β-Annealing Schedule')
ax3.grid(True, alpha=0.3)

# Learning rate
ax4 = axes[1, 1]
```

```
ax4.plot(vae_history['learning_rate'], linewidth=2, color='green')
ax4.set_xlabel('Epoch')
ax4.set_ylabel('Learning Rate')
ax4.set_title('Learning Rate Schedule')
ax4.set_yscale('log')
ax4.grid(True, alpha=0.3)

plt.suptitle('β-VAE Training Progress', fontsize=14)
plt.tight_layout()
plt.show()
```



31.4.4. 3.4 VAE Visualization & Comparison

We compare the latent spaces of AE and VAE, generate new images from random latent vectors, and demonstrate smoother interpolation enabled by the VAE's regularized latent space.

```
model_ae.eval()
model_vae.eval()

# Compare latent spaces
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(20, 9))

# Collect latent vectors from both models
with torch.no_grad():
    ae_latents = []
    vae_latents = []
    all_labels = []

    for images, labels in test_loader:
        images = images.to(device)
```

```
        # AE latent
        z_ae = model_ae.encode(images)
        ae_latents.append(z_ae.cpu().numpy())

        # VAE latent (use mean)
        mu, _ = model_vae.encode(images)
        vae_latents.append(mu.cpu().numpy())

        all_labels.append(labels.numpy())

    ae_latents = np.concatenate(ae_latents)
    vae_latents = np.concatenate(vae_latents)
    all_labels = np.concatenate(all_labels)

tsne = TSNE(n_components=2, random_state=42, perplexity=30)
ae_2d = tsne.fit_transform(ae_latents[:3000])
vae_2d = tsne.fit_transform(vae_latents[:3000])
labels_2d = all_labels[:3000]

class_names = ['T-shirt', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

scatter1 = ax1.scatter(ae_2d[:, 0], ae_2d[:, 1], c=labels_2d, cmap='tab10', alpha=0.6, s=10)
ax1.set_title('AE Latent Space (t-SNE)', fontsize=14)
ax1.set_xlabel('Dimension 1')
ax1.set_ylabel('Dimension 2')
ax1.grid(True, alpha=0.3)

scatter2 = ax2.scatter(vae_2d[:, 0], vae_2d[:, 1], c=labels_2d, cmap='tab10', alpha=0.6, s=10)
ax2.set_title('VAE Latent Space (t-SNE)', fontsize=14)
ax2.set_xlabel('Dimension 1')
ax2.grid(True, alpha=0.3)

fig.legend(handles=scatter1.legend_elements()[0], labels=class_names,
           title="Classes", loc='center right')
plt.suptitle('Latent Space Comparison: AE vs VAE', fontsize=16)
plt.tight_layout(rect=[0, 0, 0.88, 0.96])
plt.show()

# Generate new samples from VAE prior
print("\nGenerating new samples from VAE prior...")
with torch.no_grad():
    samples = model_vae.sample(16, device)
    samples = samples * 0.5 + 0.5  # Denormalize

grid = make_grid(samples, nrow=8, padding=2)
plt.figure(figsize=(12, 3))
plt.imshow(grid.permute(1, 2, 0).cpu().numpy(), cmap='gray')
plt.axis('off')
plt.title('VAE: Generated Samples from Prior N(0, I)', fontsize=14)
plt.show()

# VAE Interpolation
print("\nVAE Latent Interpolation...")
idx_a = np.where(np.array(test_dataset.targets) == 0)[0][0]
idx_b = np.where(np.array(test_dataset.targets) == 9)[0][0]

x_a = test_dataset[idx_a][0].unsqueeze(0).to(device)
x_b = test_dataset[idx_b][0].unsqueeze(0).to(device)
```

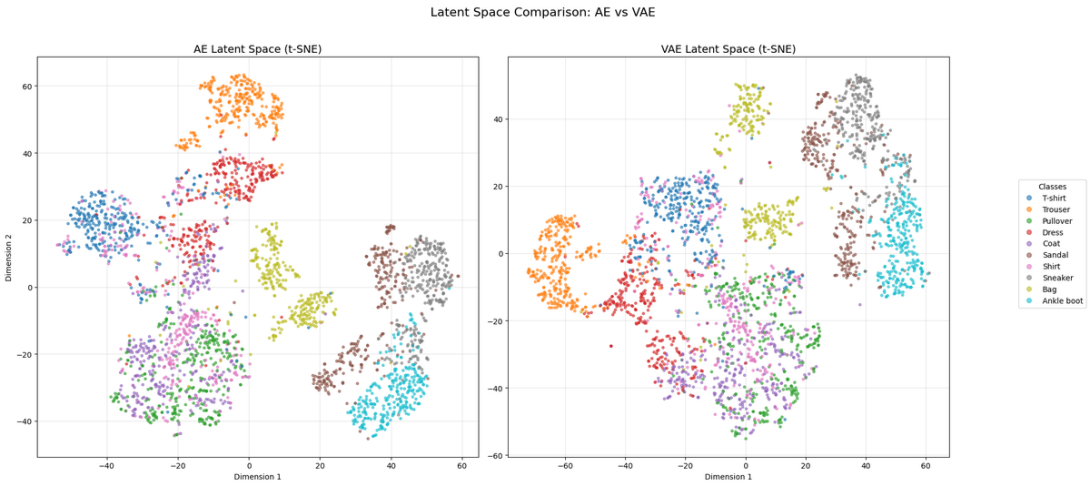
```
with torch.no_grad():
    mu_a, _ = model_vae.encode(x_a)
    mu_b, _ = model_vae.encode(x_b)

    n_steps = 10
    interpolated = []
    for alpha in np.linspace(0, 1, n_steps):
        z = (1 - alpha) * mu_a + alpha * mu_b
        img = model_vae.decode(z)
        interpolated.append(img)

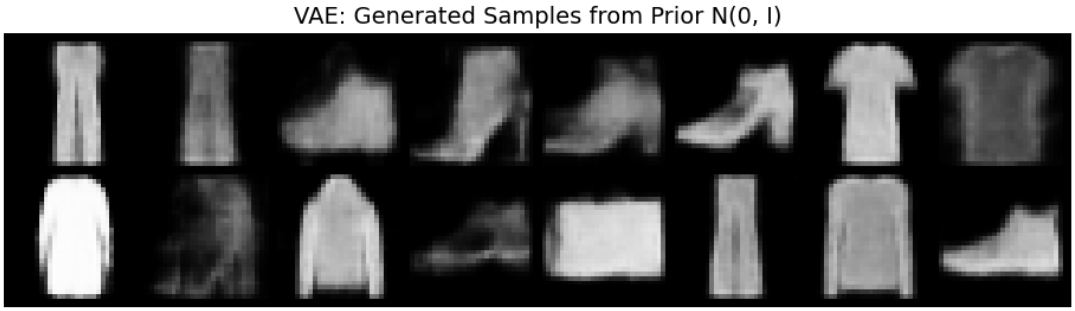
    interpolated = torch.cat(interpolated) * 0.5 + 0.5

fig, axes = plt.subplots(1, n_steps, figsize=(2 * n_steps, 2))
for i, ax in enumerate(axes):
    ax.imshow(interpolated[i, 0].cpu().numpy(), cmap='gray')
    ax.axis('off')
    ax.set_title(f'α={i/(n_steps-1):.1f}', fontsize=10)

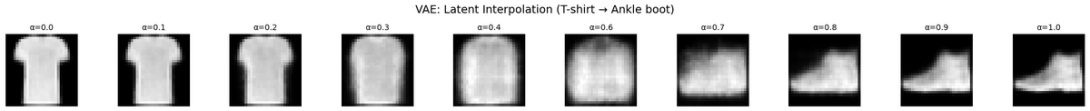
plt.suptitle('VAE: Latent Interpolation (T-shirt → Ankle boot)', fontsize=14)
plt.tight_layout()
plt.show()
```




```
Generating new samples from VAE prior...
```



```
VAE Latent Interpolation...
```



31.5. 4. Generative Adversarial Networks (GANs)

A **GAN** consists of two neural networks competing in a zero-sum game:

- **Generator (G)**: Takes random noise and generates synthetic data to fool the discriminator
- **Discriminator (D)**: Distinguishes between real and generated samples

Training continues until the discriminator can no longer distinguish real from fake, meaning $D(G(z)) \approx 0.5$.

31.5.1. 4.1 DCGAN Architecture

We implement a **Deep Convolutional GAN (DCGAN)** with: - **Generator**: Uses transposed convolutions to upsample from latent vector to 28×28 images - **Discriminator**: Uses strided convolutions to downsample and classify real vs. fake - **Training Techniques**: Label smoothing, separate learning rates (TTUR), and BCEWithLogitsLoss for stability

```python
class Generator(nn.Module):
    """DCGAN Generator with improved architecture."""

    def __init__(self, latent_dim=100, g_features=64):
        super().__init__()
        self.latent_dim = latent_dim
        nf = g_features

        # Project and reshape
        self.project = nn.Sequential(
            nn.Linear(latent_dim, nf * 8 * 7 * 7),
            nn.BatchNorm1d(nf * 8 * 7 * 7),
            nn.ReLU(True)
        )

        # Transposed convolutions
        self.conv_blocks = nn.Sequential(
            # 7x7 -> 14x14
            nn.ConvTranspose2d(nf * 8, nf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(nf * 4),
            nn.ReLU(True),

            # 14x14 -> 28x28
            nn.ConvTranspose2d(nf * 4, nf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(nf * 2),
            nn.ReLU(True),
```

```
            # Final output
            nn.Conv2d(nf * 2, 1, 3, 1, 1, bias=False),
            nn.Tanh()
        )

        self.apply(self._init_weights)

    def _init_weights(self, m):
        if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d)):
            nn.init.normal_(m.weight, 0.0, 0.02)
        elif isinstance(m, nn.BatchNorm2d):
            nn.init.normal_(m.weight, 1.0, 0.02)
            nn.init.constant_(m.bias, 0)
        elif isinstance(m, nn.Linear):
            nn.init.normal_(m.weight, 0.0, 0.02)

    def forward(self, z):
        x = self.project(z)
        x = x.view(x.size(0), -1, 7, 7)
        return self.conv_blocks(x)

class Discriminator(nn.Module):
    """DCGAN Discriminator with spectral normalization."""

    def __init__(self, d_features=64):
        super().__init__()
        nf = d_features

        self.conv_blocks = nn.Sequential(
            # 28x28 -> 14x14
            nn.utils.spectral_norm(nn.Conv2d(1, nf, 4, 2, 1, bias=False)),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout2d(0.25),

            # 14x14 -> 7x7
            nn.utils.spectral_norm(nn.Conv2d(nf, nf * 2, 4, 2, 1, bias=False)),
            nn.BatchNorm2d(nf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout2d(0.25),

            # 7x7 -> 3x3
            nn.utils.spectral_norm(nn.Conv2d(nf * 2, nf * 4, 4, 2, 1, bias=False)),
            nn.BatchNorm2d(nf * 4),
            nn.LeakyReLU(0.2, inplace=True),
        )

        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(nf * 4 * 3 * 3, 1)
        )

        self.apply(self._init_weights)

    def _init_weights(self, m):
        if isinstance(m, nn.Conv2d):
            nn.init.normal_(m.weight, 0.0, 0.02)
        elif isinstance(m, nn.BatchNorm2d):
            nn.init.normal_(m.weight, 1.0, 0.02)
            nn.init.constant_(m.bias, 0)
```

```
    def forward(self, x):
        features = self.conv_blocks(x)
        return self.classifier(features)

def compute_diversity(generator, latent_dim, n_samples, device):
    """Compute diversity of generated samples."""
    generator.eval()
    with torch.no_grad():
        z = torch.randn(n_samples, latent_dim, device=device)
        samples = generator(z).view(n_samples, -1)
        distances = torch.cdist(samples, samples)
        mask = ~torch.eye(n_samples, dtype=bool, device=device)
        diversity = distances[mask].mean().item()
    generator.train()
    return diversity
```

31.5.2. 4.2 GAN Training Loop

The training alternates between: 1. **Training Discriminator**: On real images (label ~0.9 for smoothing) and fake images (label 0) 2. **Training Generator**: To produce fakes that the discriminator classifies as real (label 1)

```
GAN_LATENT_DIM = 100
G_FEATURES = 64
D_FEATURES = 64

generator = Generator(latent_dim=GAN_LATENT_DIM, g_features=G_FEATURES).to(device)
discriminator = Discriminator(d_features=D_FEATURES).to(device)

print("Generator:")
print(generator)
print(f"\nGenerator parameters: {sum(p.numel() for p in generator.parameters()):,}")

print("\nDiscriminator:")
print(discriminator)
print(f"\nDiscriminator parameters: {sum(p.numel() for p in discriminator.parameters()):,}")
```

```
Generator:
Generator(
  (project): Sequential(
    (0): Linear(in_features=100, out_features=25088, bias=True)
    (1): BatchNorm1d(25088, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (conv_blocks): Sequential(
    (0): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1), bias=False)
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1), bias=False)
    (4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (5): ReLU(inplace=True)
```

```
    (6): Conv2d(128, 1, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (7): Tanh()
  )
)

Generator parameters: 5,207,424
[... 12 lines omitted ...]
    (8): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (9): LeakyReLU(negative_slope=0.2, inplace=True)
  )
  (classifier): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=2304, out_features=1, bias=True)
  )
)

Discriminator parameters: 659,457
```

```
GAN_EPOCHS = 30
G_LR = 2e-4
D_LR = 2e-4
BETA1, BETA2 = 0.5, 0.999
LABEL_SMOOTHING = 0.1
NOISE_STD = 0.1  # Instance noise (decays)
GAN_PATIENCE = 15
DIVERSITY_THRESHOLD = 0.01

# Optimizers
g_optimizer = optim.Adam(generator.parameters(), lr=G_LR, betas=(BETA1, BETA2))
d_optimizer = optim.Adam(discriminator.parameters(), lr=D_LR, betas=(BETA1, BETA2))

# Loss
criterion = nn.BCEWithLogitsLoss()

# Fixed noise for visualization
fixed_noise = torch.randn(64, GAN_LATENT_DIM, device=device)

# Training history
gan_history = {
    'g_loss': [], 'd_loss': [],
    'd_real_acc': [], 'd_fake_acc': [],
    'diversity': []
}

# Early stopping for GAN
class GANEarlyStopping:
    def __init__(self, patience=15, min_delta=0.01, verbose=True):
        self.patience = patience
        self.min_delta = min_delta
        self.verbose = verbose
        self.counter = 0
        self.best_score = None
        self.early_stop = False
        self.best_epoch = 0
```

```
    def __call__(self, score, epoch):
        is_improvement = self.best_score is None or score > self.best_score + self.min_delta
        if is_improvement:
            if self.verbose and self.best_score is not None:
                print(f"Diversity improved: {self.best_score:.4f} → {score:.4f}")
            self.best_score = score
            self.best_epoch = epoch
            self.counter = 0
        else:
            self.counter += 1
            if self.verbose:
                print(f"No improvement for {self.counter}/{self.patience} epochs")
            if self.counter >= self.patience:
                self.early_stop = True
        return self.early_stop

gan_early_stopper = GANEarlyStopping(patience=GAN_PATIENCE)

print("GAN training configuration initialized.")
```

```
GAN training configuration initialized.
```

```
print(f"Generator parameters: {sum(p.numel() for p in generator.parameters()):,}")
print(f"Discriminator parameters: {sum(p.numel() for p in discriminator.parameters()):,}")
print(f"Device: {device}")
best_diversity = 0.0
best_g_state = None
best_d_state = None

for epoch in range(GAN_EPOCHS):
    print(f"\nEpoch {epoch + 1}/{GAN_EPOCHS}")
    print("-" * 40)

    generator.train()
    discriminator.train()

    # Instance noise decay
    current_noise_std = NOISE_STD * max(0, 1 - epoch / (GAN_EPOCHS * 0.5))

    epoch_g_loss = 0.0
    epoch_d_loss = 0.0
    epoch_d_real_acc = 0.0
    epoch_d_fake_acc = 0.0
    n_batches = 0

    pbar = tqdm(train_loader, desc="Training", leave=False)
    for real_imgs, _ in pbar:
        real_imgs = real_imgs.to(device)
        batch_size = real_imgs.size(0)

        # Labels with smoothing
        real_labels = torch.ones(batch_size, 1, device=device) * (1 - LABEL_SMOOTHING)
        fake_labels = torch.zeros(batch_size, 1, device=device)

        # Add instance noise
        if current_noise_std > 0:
            real_imgs = real_imgs + current_noise_std * torch.randn_like(real_imgs)

        # Discriminator
        d_optimizer.zero_grad()
```

```
        # Real images
        d_real = discriminator(real_imgs)
        d_real_loss = criterion(d_real, real_labels)

        # Fake images
        z = torch.randn(batch_size, GAN_LATENT_DIM, device=device)
        fake_imgs = generator(z).detach()
        if current_noise_std > 0:
            fake_imgs = fake_imgs + current_noise_std * torch.randn_like(fake_imgs)

        d_fake = discriminator(fake_imgs)
        d_fake_loss = criterion(d_fake, fake_labels)

        d_loss = d_real_loss + d_fake_loss
        d_loss.backward()
        d_optimizer.step()

        # Generator
        g_optimizer.zero_grad()

        z = torch.randn(batch_size, GAN_LATENT_DIM, device=device)
        fake_imgs = generator(z)
        g_output = discriminator(fake_imgs)
        g_loss = criterion(g_output, torch.ones(batch_size, 1, device=device))

        g_loss.backward()
        g_optimizer.step()

        # Metrics
        epoch_g_loss += g_loss.item()
        epoch_d_loss += d_loss.item()
        epoch_d_real_acc += (torch.sigmoid(d_real) > 0.5).float().mean().item()
        epoch_d_fake_acc += (torch.sigmoid(d_fake) < 0.5).float().mean().item()
        n_batches += 1

        pbar.set_postfix({
            'G': f'{g_loss.item():.3f}',
            'D': f'{d_loss.item():.3f}'
        })

    # Epoch metrics
    epoch_g_loss /= n_batches
    epoch_d_loss /= n_batches
    epoch_d_real_acc /= n_batches
    epoch_d_fake_acc /= n_batches

    # Compute diversity
    diversity = compute_diversity(generator, GAN_LATENT_DIM, 100, device)

    # Update history
    gan_history['g_loss'].append(epoch_g_loss)
    gan_history['d_loss'].append(epoch_d_loss)
    gan_history['d_real_acc'].append(epoch_d_real_acc)
    gan_history['d_fake_acc'].append(epoch_d_fake_acc)
    gan_history['diversity'].append(diversity)

    print(f"  G Loss:      {epoch_g_loss:.4f}")
    print(f"  D Loss:      {epoch_d_loss:.4f}")
    print(f"  D Real Acc:  {epoch_d_real_acc:.2%}")
```

```
    print(f"  D Fake Acc:  {epoch_d_fake_acc:.2%}")
    print(f"  Diversity:   {diversity:.4f}")

    # Save best model
    if diversity > best_diversity:
        best_diversity = diversity
        best_g_state = generator.state_dict().copy()
        best_d_state = discriminator.state_dict().copy()

    # Show samples every 5 epochs
    if (epoch + 1) % 5 == 0:
        generator.eval()
        with torch.no_grad():
            samples = generator(fixed_noise[:16])
            samples = samples * 0.5 + 0.5

        fig, axes = plt.subplots(2, 8, figsize=(16, 4))
        for i in range(16):
            axes[i//8, i%8].imshow(samples[i, 0].cpu().numpy(), cmap='gray')
            axes[i//8, i%8].axis('off')
        plt.suptitle(f'Generated Samples - Epoch {epoch+1}')
        plt.show()
        generator.train()

    # Mode collapse warning
    if diversity < DIVERSITY_THRESHOLD:
        print(f"  ! Warning: Low diversity ({diversity:.4f}), possible mode collapse")

    # Early stopping
    if gan_early_stopper(diversity, epoch):
        print(f"\n! Early stopping triggered at epoch {epoch + 1}")
        break

# Restore best model
if best_g_state is not None:
    generator.load_state_dict(best_g_state)
    discriminator.load_state_dict(best_d_state)

print(f"Best diversity: {best_diversity:.4f}")
```

```
Generator parameters: 5,207,424
Discriminator parameters: 659,457
Device: cuda:0

Epoch 1/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      1.5362
  D Loss:      0.9383
  D Real Acc:  73.94%
  D Fake Acc:  89.80%
  Diversity:   22.6270

Epoch 2/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      1.0328
  D Loss:      1.0616
  D Real Acc:  67.23%
  D Fake Acc:  85.78%
  Diversity:   22.3196
No improvement for 1/15 epochs

Epoch 3/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      0.9641
  D Loss:      1.1059
  D Real Acc:  64.34%
  D Fake Acc:  83.99%
  Diversity:   22.5696
No improvement for 2/15 epochs

Epoch 4/30
----------------------------------------
```

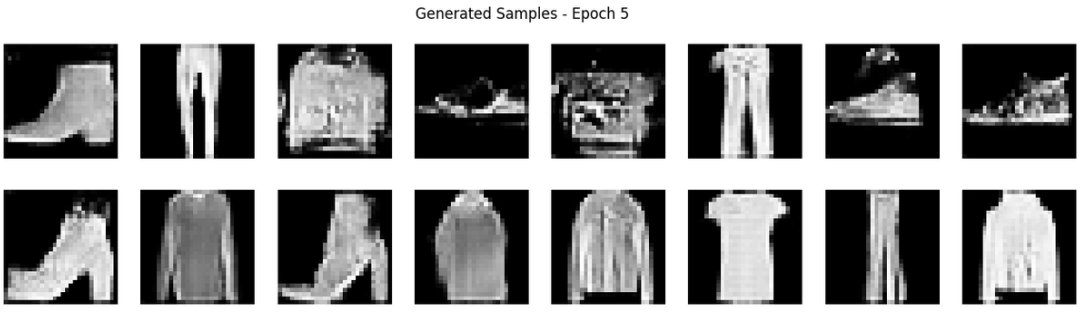
```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      0.9761
  D Loss:      1.1278
  D Real Acc:  62.26%
  D Fake Acc:  83.07%
  Diversity:   22.2241
No improvement for 3/15 epochs

Epoch 5/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      1.0017
  D Loss:      1.1307
  D Real Acc:  62.25%
  D Fake Acc:  82.99%
  Diversity:   23.6836
```




```
Diversity improved: 22.6270 -> 23.6836

Epoch 6/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      1.0255
  D Loss:      1.1278
  D Real Acc:  62.47%
  D Fake Acc:  83.12%
  Diversity:   22.4802
No improvement for 1/15 epochs

Epoch 7/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      1.0395
  D Loss:      1.1358
  D Real Acc:  62.05%
  D Fake Acc:  82.79%
  Diversity:   22.0667
No improvement for 2/15 epochs

Epoch 8/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      1.0552
  D Loss:      1.1389
  D Real Acc:  61.92%
  D Fake Acc:  82.59%
  Diversity:   21.3046
No improvement for 3/15 epochs

Epoch 9/30
----------------------------------------
```

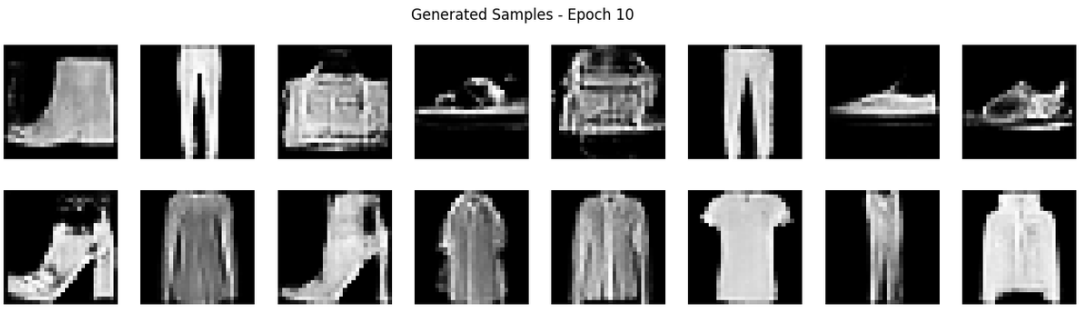
```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      1.0864
  D Loss:      1.1462
  D Real Acc:  61.36%
  D Fake Acc:  82.23%
  Diversity:   23.3660
No improvement for 4/15 epochs

Epoch 10/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      1.1162
  D Loss:      1.1441
  D Real Acc:  61.69%
  D Fake Acc:  82.49%
  Diversity:   22.4592
```




```
No improvement for 5/15 epochs

Epoch 11/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      1.1297
  D Loss:      1.1431
  D Real Acc:  61.77%
  D Fake Acc:  81.83%
  Diversity:   22.6371
No improvement for 6/15 epochs

Epoch 12/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      1.1594
  D Loss:      1.1539
  D Real Acc:  60.95%
  D Fake Acc:  81.52%
  Diversity:   22.9361
No improvement for 7/15 epochs

Epoch 13/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      1.1622
  D Loss:      1.1502
  D Real Acc:  60.98%
  D Fake Acc:  81.80%
  Diversity:   22.8388
No improvement for 8/15 epochs

Epoch 14/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

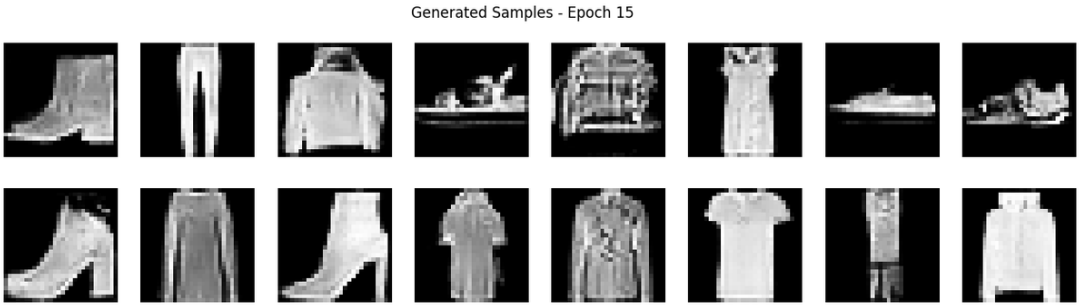
```
G Loss:      1.1892
  D Loss:      1.1535
  D Real Acc:  61.11%
  D Fake Acc:  81.50%
  Diversity:   22.5820
```

```
No improvement for 9/15 epochs

Epoch 15/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      1.1942
  D Loss:      1.1461
  D Real Acc:  61.46%
  D Fake Acc:  81.74%
  Diversity:   23.0941
```




```
No improvement for 10/15 epochs

Epoch 16/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      1.2141
  D Loss:      1.1412
  D Real Acc:  62.02%
  D Fake Acc:  82.04%
  Diversity:   23.4339
No improvement for 11/15 epochs

Epoch 17/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      1.2183
  D Loss:      1.1491
  D Real Acc:  61.32%
  D Fake Acc:  81.50%
  Diversity:   22.7151
No improvement for 12/15 epochs

Epoch 18/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      1.2108
  D Loss:      1.1450
  D Real Acc:  61.54%
  D Fake Acc:  81.88%
```

```
  Diversity:   21.8392
No improvement for 13/15 epochs

Epoch 19/30
----------------------------------------
```

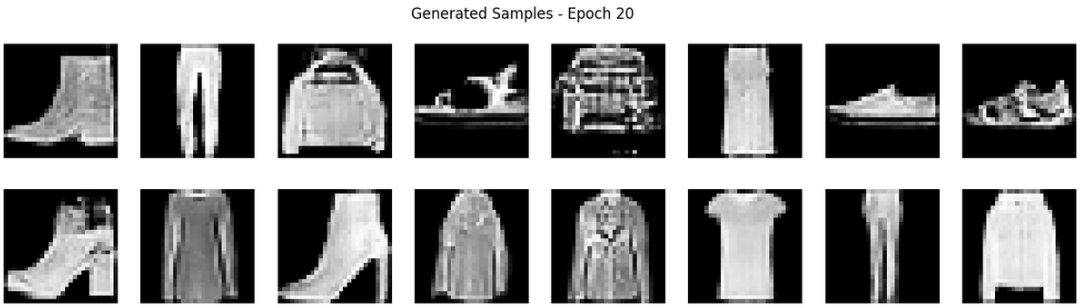
```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      1.2096
  D Loss:      1.1502
  D Real Acc:  61.25%
  D Fake Acc:  81.39%
  Diversity:   21.9829
No improvement for 14/15 epochs

Epoch 20/30
----------------------------------------
```

```
Training:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
G Loss:      1.2208
  D Loss:      1.1446
  D Real Acc:  61.85%
  D Fake Acc:  81.72%
  Diversity:   21.9658
```




```
No improvement for 15/15 epochs

! Early stopping triggered at epoch 20
Best diversity: 23.6836
```

31.5.3. 4.3 Training Visualization

We plot the loss curves to monitor the adversarial training dynamics and compare real vs. generated images.

```
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# Loss curves
ax1 = axes[0, 0]
ax1.plot(gan_history['g_loss'], label='Generator', linewidth=2)
ax1.plot(gan_history['d_loss'], label='Discriminator', linewidth=2)
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Loss')
ax1.set_title('Generator vs Discriminator Loss')
ax1.legend()
```

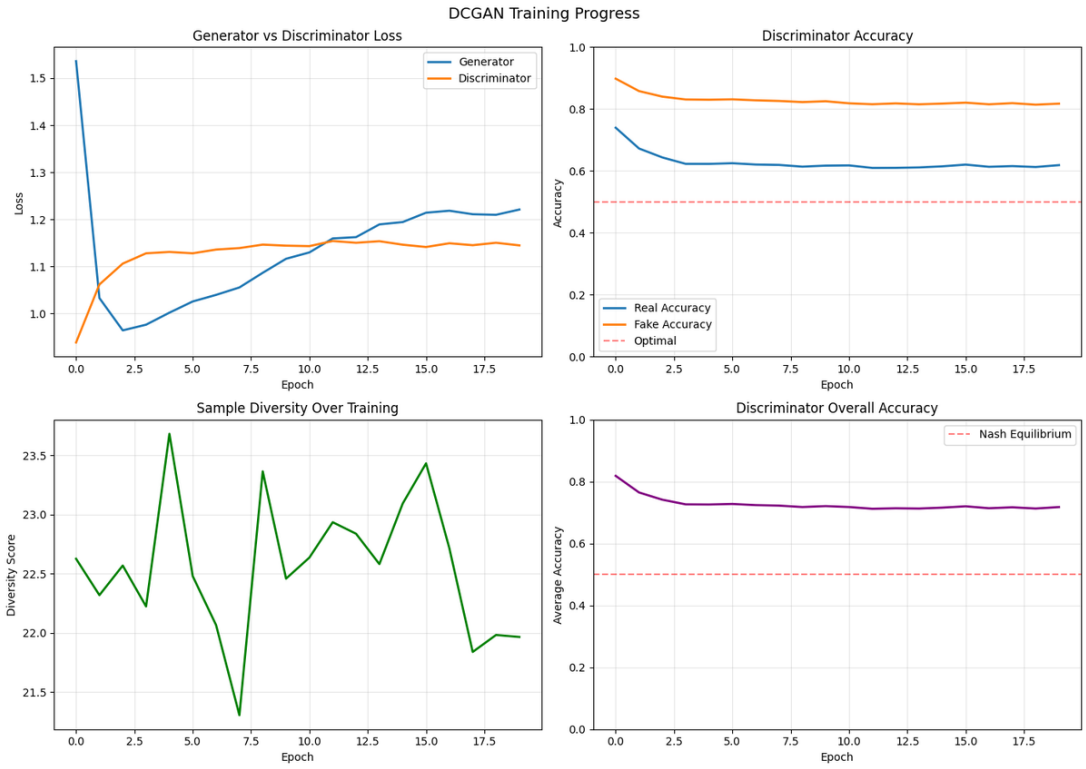
```
ax1.grid(True, alpha=0.3)

# Discriminator accuracy
ax2 = axes[0, 1]
ax2.plot(gan_history['d_real_acc'], label='Real Accuracy', linewidth=2)
ax2.plot(gan_history['d_fake_acc'], label='Fake Accuracy', linewidth=2)
ax2.axhline(y=0.5, color='r', linestyle='--', alpha=0.5, label='Optimal')
ax2.set_xlabel('Epoch')
ax2.set_ylabel('Accuracy')
ax2.set_title('Discriminator Accuracy')
ax2.legend()
ax2.grid(True, alpha=0.3)
ax2.set_ylim(0, 1)

# Diversity
ax3 = axes[1, 0]
ax3.plot(gan_history['diversity'], linewidth=2, color='green')
ax3.set_xlabel('Epoch')
ax3.set_ylabel('Diversity Score')
ax3.set_title('Sample Diversity Over Training')
ax3.grid(True, alpha=0.3)

# Combined D accuracy
ax4 = axes[1, 1]
combined_acc = [(r + f) / 2 for r, f in zip(gan_history['d_real_acc'], gan_history['d_fake_acc'])]
ax4.plot(combined_acc, linewidth=2, color='purple')
ax4.axhline(y=0.5, color='r', linestyle='--', alpha=0.5, label='Nash Equilibrium')
ax4.set_xlabel('Epoch')
ax4.set_ylabel('Average Accuracy')
ax4.set_title('Discriminator Overall Accuracy')
ax4.legend()
ax4.grid(True, alpha=0.3)
ax4.set_ylim(0, 1)

plt.suptitle('DCGAN Training Progress', fontsize=14)
plt.tight_layout()
plt.show()
```




```python
def show_real_vs_fake_comparison(generator, data_loader, latent_dim, device, n_samples=8):
    """Display real images alongside generated ones for comparison."""
    generator.eval()

    # Get real samples
    real_images, _ = next(iter(data_loader))
    real_images = real_images[:n_samples].cpu()

    # Generate fake samples
    with torch.no_grad():
        z = torch.randn(n_samples, latent_dim, device=device)
        fake_images = generator(z).cpu()

    # Create comparison figure
    fig, axes = plt.subplots(2, n_samples, figsize=(n_samples * 2, 4))

    for i in range(n_samples):
        # Real images
        axes[0, i].imshow(real_images[i].squeeze(), cmap='gray')
        axes[0, i].axis('off')
        if i == 0:
            axes[0, i].set_title('Real', fontsize=12)

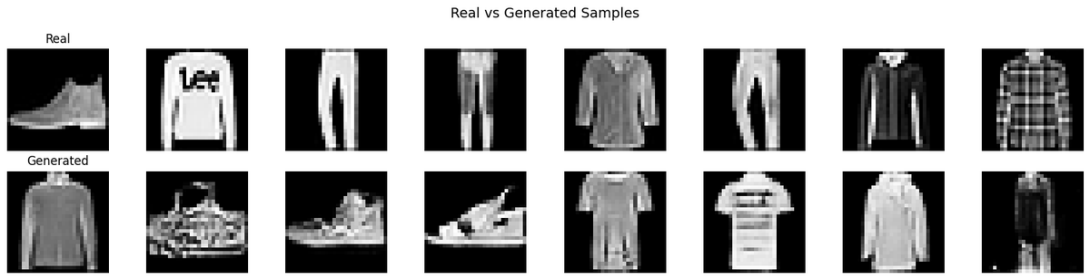
        # Fake images
        axes[1, i].imshow(fake_images[i].squeeze(), cmap='gray')
        axes[1, i].axis('off')
        if i == 0:
            axes[1, i].set_title('Generated', fontsize=12)

    plt.suptitle('Real vs Generated Samples', fontsize=14)
    plt.tight_layout()
```

```
    plt.show()

    return real_images, fake_images

# Show comparison
real_samples, fake_samples = show_real_vs_fake_comparison(
    generator, test_loader, GAN_LATENT_DIM, device
)
```



31.6. 5. Diffusion Models & Stable Diffusion (Optional)

Diffusion models learn to generate images by reversing a gradual noising process:

1. **Forward Diffusion**: Progressively adds Gaussian noise until only pure noise remains
2. **Reverse Diffusion**: The model learns to denoise step-by-step, starting from random noise

Stable Diffusion operates in a compressed **latent space** for efficiency, using a text encoder to condition generation on natural language prompts.

31.6.1. 5.1 DDPM Architecture

The **U-Net [15]** is the backbone of diffusion models, predicting noise at each timestep:

- **Time Embeddings**: Sinusoidal positional encodings inform the network which denoising step it's performing
- **Encoder Path**: Progressively downsamples while extracting features at multiple scales
- **Bottleneck Attention**: Self-attention captures global dependencies in the compressed representation
- **Decoder Path**: Upsamples with skip connections, combining local and global features for precise noise prediction

```
from dataclasses import dataclass, asdict
import math

@dataclass
class DiffusionConfig:
    """Configuration for DDPM training."""
    image_size: int = 28
    image_channels: int = 1
    model_channels: int = 64
    timesteps: int = 1000
```

```
    beta_start: float = 1e-4
    beta_end: float = 0.02
    beta_schedule: str = "linear"
    batch_size: int = 128
    learning_rate: float = 2e-4
    num_epochs: int = 30
    ema_decay: float = 0.9999
    gradient_clip: float = 1.0
    patience: int = 10
    dropout: float = 0.1
    output_dir: str = "diffusion_results"

class SinusoidalPositionEmbeddings(nn.Module):
    """Sinusoidal time step embeddings for diffusion models."""

    def __init__(self, dim: int):
        super().__init__()
        self.dim = dim

    def forward(self, time: torch.Tensor) -> torch.Tensor:
        device = time.device
        half_dim = self.dim // 2
        embeddings = math.log(10000) / (half_dim - 1)
        embeddings = torch.exp(torch.arange(half_dim, device=device) * -embeddings)
        embeddings = time[:, None] * embeddings[None, :]
        embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1)
        return embeddings

class DiffusionAttentionBlock(nn.Module):
    """Self-attention block for U-Net."""

    def __init__(self, channels: int, num_heads: int = 4):
        super().__init__()
        self.num_heads = num_heads
        self.norm = nn.GroupNorm(8, channels)
        self.attention = nn.MultiheadAttention(channels, num_heads, batch_first=True)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        b, c, h, w = x.shape
        x_norm = self.norm(x)
        x_flat = x_norm.view(b, c, h * w).transpose(1, 2)
        attn_out, _ = self.attention(x_flat, x_flat, x_flat)
        attn_out = attn_out.transpose(1, 2).view(b, c, h, w)
        return x + attn_out

class DiffusionUNet(nn.Module):
    """U-Net architecture for noise prediction in DDPM."""

    def __init__(self, config: DiffusionConfig):
        super().__init__()
        self.config = config
        time_emb_dim = config.model_channels * 4

        # Time embedding network
        self.time_mlp = nn.Sequential(
            SinusoidalPositionEmbeddings(config.model_channels),
            nn.Linear(config.model_channels, time_emb_dim),
            nn.SiLU(),
            nn.Linear(time_emb_dim, time_emb_dim)
        )
```

```
        # Encoder path
        self.conv1 = nn.Conv2d(config.image_channels, 64, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1)  # 14x14
        self.conv3 = nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=1)  # 7x7

        # Time projections for each level
        self.time_fc1 = nn.Linear(time_emb_dim, 64)
        self.time_fc2 = nn.Linear(time_emb_dim, 128)
        self.time_fc3 = nn.Linear(time_emb_dim, 256)

        # Bottleneck with attention
        self.bottleneck1 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        self.bottleneck_attn = DiffusionAttentionBlock(256)
        self.bottleneck2 = nn.Conv2d(256, 256, kernel_size=3, padding=1)
        self.time_fc_bn = nn.Linear(time_emb_dim, 256)

        # Decoder path with skip connections
        self.up1 = nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1)  # 14x14
        self.conv4 = nn.Conv2d(256, 128, kernel_size=3, padding=1)
        self.time_fc4 = nn.Linear(time_emb_dim, 128)

        self.up2 = nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1)  # 28x28
        self.conv5 = nn.Conv2d(128, 64, kernel_size=3, padding=1)
        self.time_fc5 = nn.Linear(time_emb_dim, 64)

        # Output projection
        self.final_conv = nn.Conv2d(64, config.image_channels, kernel_size=3, padding=1)

        # Normalization layers
        self.norm1 = nn.GroupNorm(8, 64)
        self.norm2 = nn.GroupNorm(8, 128)
        self.norm3 = nn.GroupNorm(8, 256)
        self.norm4 = nn.GroupNorm(8, 128)
        self.norm5 = nn.GroupNorm(8, 64)
        self.norm_bn = nn.GroupNorm(8, 256)

        self.dropout = nn.Dropout(config.dropout)

    def forward(self, x: torch.Tensor, t: torch.Tensor) -> torch.Tensor:
        # Time embedding
        t_emb = self.time_mlp(t)

        # Encoder with time conditioning
        h1 = F.silu(self.norm1(self.conv1(x)) + self.time_fc1(t_emb)[:, :, None, None])
        h2 = F.silu(self.norm2(self.conv2(h1)) + self.time_fc2(t_emb)[:, :, None, None])
        h3 = F.silu(self.norm3(self.conv3(h2)) + self.time_fc3(t_emb)[:, :, None, None])

        # Bottleneck
        h = F.silu(self.norm_bn(self.bottleneck1(h3)) + self.time_fc_bn(t_emb)[:, :, None, None])
        h = self.bottleneck_attn(h)
        h = self.dropout(h)
        h = F.silu(self.bottleneck2(h))

        # Decoder with skip connections
        h = torch.cat([self.up1(h), h2], dim=1)
        h = F.silu(self.norm4(self.conv4(h)) + self.time_fc4(t_emb)[:, :, None, None])

        h = torch.cat([self.up2(h), h1], dim=1)
        h = F.silu(self.norm5(self.conv5(h)) + self.time_fc5(t_emb)[:, :, None, None])
```

```
        return self.final_conv(h)
```

31.6.2. 5.2 Gaussian Diffusion Process

The diffusion process consists of two Markov chains:

- **Forward Process (q)**: Gradually adds Gaussian noise over T timesteps until the data becomes pure noise
- **Reverse Process (p)**: Learns to denoise step-by-step, recovering the original data distribution
- **DDIM Sampling**: An accelerated deterministic sampler that skips timesteps for faster generation (50 steps instead of 1000)

```
class GaussianDiffusion:
    """Gaussian diffusion process for DDPM with forward and reverse sampling."""

    def __init__(self, config: DiffusionConfig, device: torch.device):
        self.config = config
        self.device = device
        self.timesteps = config.timesteps

        # Create noise schedule
        if config.beta_schedule == "linear":
            self.betas = torch.linspace(config.beta_start, config.beta_end, config.timesteps)
        elif config.beta_schedule == "cosine":
            steps = config.timesteps + 1
            x = torch.linspace(0, config.timesteps, steps)
            alphas_cumprod = torch.cos(((x / config.timesteps) + 0.008) / 1.008 * math.pi * 0.5) ** 2
            alphas_cumprod = alphas_cumprod / alphas_cumprod[0]
            betas = 1 - (alphas_cumprod[1:] / alphas_cumprod[:-1])
            self.betas = torch.clamp(betas, 0.0001, 0.9999)

        self.betas = self.betas.to(device)

        # Precompute diffusion quantities
        self.alphas = 1.0 - self.betas
        self.alphas_cumprod = torch.cumprod(self.alphas, dim=0)
        self.alphas_cumprod_prev = F.pad(self.alphas_cumprod[:-1], (1, 0), value=1.0)

        self.sqrt_alphas_cumprod = torch.sqrt(self.alphas_cumprod)
        self.sqrt_one_minus_alphas_cumprod = torch.sqrt(1.0 - self.alphas_cumprod)

        # Posterior variance for reverse process
        self.posterior_variance = (
            self.betas * (1.0 - self.alphas_cumprod_prev) / (1.0 - self.alphas_cumprod)
        )

    def _extract(self, a: torch.Tensor, t: torch.Tensor, x_shape: tuple) -> torch.Tensor:
        """Extract values from tensor a at indices t."""
        batch_size = t.shape[0]
        out = a.gather(-1, t)
        return out.reshape(batch_size, *((1,) * (len(x_shape) - 1)))

    def q_sample(self, x_start: torch.Tensor, t: torch.Tensor, noise=None):
        """Forward diffusion: sample x_t from q(x_t | x_0)."""
        if noise is None:
            noise = torch.randn_like(x_start)
```

```
        sqrt_alphas_cumprod_t = self._extract(self.sqrt_alphas_cumprod, t, x_start.shape)
        sqrt_one_minus_alphas_cumprod_t = self._extract(self.sqrt_one_minus_alphas_cumprod, t, x_start.shape)

        return sqrt_alphas_cumprod_t * x_start + sqrt_one_minus_alphas_cumprod_t * noise, noise

    def p_losses(self, model: nn.Module, x_start: torch.Tensor, t: torch.Tensor, noise=None):
        """Compute training loss (predict noise)."""
        if noise is None:
            noise = torch.randn_like(x_start)

        x_noisy, noise = self.q_sample(x_start, t, noise)
        predicted_noise = model(x_noisy, t.float())

        return F.mse_loss(predicted_noise, noise)

    @torch.no_grad()
    def p_sample(self, model: nn.Module, x: torch.Tensor, t: torch.Tensor, t_index: int):
        """Sample x_{t-1} from p(x_{t-1} | x_t) - reverse diffusion step."""
        betas_t = self._extract(self.betas, t, x.shape)
        sqrt_one_minus_alphas_cumprod_t = self._extract(self.sqrt_one_minus_alphas_cumprod, t, x.shape)
        sqrt_recip_alphas_t = self._extract(1.0 / torch.sqrt(self.alphas), t, x.shape)

        # Predict mean
        model_mean = sqrt_recip_alphas_t * (
            x - betas_t * model(x, t.float()) / sqrt_one_minus_alphas_cumprod_t
        )

        if t_index == 0:
            return model_mean
        else:
            posterior_variance_t = self._extract(self.posterior_variance, t, x.shape)
            noise = torch.randn_like(x)
            return model_mean + torch.sqrt(posterior_variance_t) * noise

    @torch.no_grad()
    def sample(self, model: nn.Module, batch_size: int, channels: int, image_size: int,
               return_all_timesteps: bool = False):
        """Generate samples using full DDPM reverse process."""
        model.eval()
        shape = (batch_size, channels, image_size, image_size)
        x = torch.randn(shape, device=self.device)

        imgs = [x] if return_all_timesteps else None

        for i in tqdm(reversed(range(0, self.timesteps)), desc='DDPM Sampling', total=self.timesteps):
            t = torch.full((batch_size,), i, device=self.device, dtype=torch.long)
            x = self.p_sample(model, x, t, i)
            if return_all_timesteps:
                imgs.append(x)

        if return_all_timesteps:
            return torch.stack(imgs, dim=1)
        return x

    @torch.no_grad()
    def ddim_sample(self, model: nn.Module, batch_size: int, channels: int, image_size: int,
                    ddim_steps: int = 50, eta: float = 0.0):
        """Generate samples using DDIM (faster deterministic sampling)."""
        model.eval()
        shape = (batch_size, channels, image_size, image_size)
```

```
        x = torch.randn(shape, device=self.device)

        # Select timesteps for DDIM
        c = self.timesteps // ddim_steps
        timesteps = list(range(0, self.timesteps, c))

        for i in tqdm(reversed(range(len(timesteps))), desc='DDIM Sampling', total=len(timesteps)):
            t = torch.full((batch_size,), timesteps[i], device=self.device, dtype=torch.long)

            predicted_noise = model(x, t.float())
            alpha_t = self._extract(self.alphas_cumprod, t, x.shape)

            if i > 0:
                t_prev = torch.full((batch_size,), timesteps[i - 1], device=self.device, dtype=torch.long)
                alpha_t_prev = self._extract(self.alphas_cumprod, t_prev, x.shape)
            else:
                alpha_t_prev = torch.ones_like(alpha_t)

            # Predict x_0
            pred_x0 = (x - torch.sqrt(1 - alpha_t) * predicted_noise) / torch.sqrt(alpha_t)
            pred_x0 = torch.clamp(pred_x0, -1, 1)

            # DDIM update
            sigma = eta * torch.sqrt((1 - alpha_t_prev) / (1 - alpha_t)) * torch.sqrt(1 - alpha_t /
   alpha_t_prev)
            dir_xt = torch.sqrt(1 - alpha_t_prev - sigma ** 2) * predicted_noise
            noise = sigma * torch.randn_like(x) if i > 0 else 0

            x = torch.sqrt(alpha_t_prev) * pred_x0 + dir_xt + noise

        return x

# Initialize diffusion config and process
DIFFUSION_CONFIG = DiffusionConfig(
    timesteps=1000,
    beta_schedule="linear",
    num_epochs=30,
    learning_rate=2e-4,
    patience=10
)

diffusion = GaussianDiffusion(DIFFUSION_CONFIG, device)
print(f"  - Timesteps: {DIFFUSION_CONFIG.timesteps}")
print(f"  - Beta schedule: {DIFFUSION_CONFIG.beta_schedule}")
```

```
- Timesteps: 1000
  - Beta schedule: linear
```

31.6.3. 5.3 Model Initialization

We initialize the U-Net model with the diffusion configuration. The model has approximately 3M parameters - much larger than AE/VAE but necessary for learning the complex denoising function.

```
# Create U-Net model
diffusion_model = DiffusionUNet(DIFFUSION_CONFIG).to(device)

# Count parameters
total_params = sum(p.numel() for p in diffusion_model.parameters())
trainable_params = sum(p.numel() for p in diffusion_model.parameters() if p.requires_grad)
```

```
print(f"  - Total parameters: {total_params:,}")
print(f"  - Trainable parameters: {trainable_params:,}")
```

```
- Total parameters: 3,152,961
  - Trainable parameters: 3,152,961
```

31.6.4. 5.4 Training the DDPM

The training objective is surprisingly simple: predict the noise that was added at each timestep.

- **Loss**: MSE between predicted and actual noise
- **Timestep Sampling**: Random t ∈ [0, T) for each batch
- **Early Stopping**: Monitors validation loss with patience
- **Cosine Annealing**: Learning rate schedule for smooth convergence

```
class DiffusionEarlyStopping:
    """Early stopping for diffusion model training."""
    def __init__(self, patience=10, min_delta=0.0):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = None
        self.early_stop = False
        self.best_epoch = 0

    def __call__(self, val_loss, epoch):
        if self.best_loss is None:
            self.best_loss = val_loss
            self.best_epoch = epoch
        elif val_loss > self.best_loss - self.min_delta:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_loss = val_loss
            self.best_epoch = epoch
            self.counter = 0
        return self.early_stop

def train_diffusion_model(model, diffusion, train_loader, val_loader, config, device):
    """DDPM training loop with early stopping."""

    optimizer = torch.optim.AdamW(model.parameters(), lr=config.learning_rate)
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
        optimizer, T_max=config.num_epochs, eta_min=1e-6
    )
    early_stopping = DiffusionEarlyStopping(patience=config.patience)

    history = {'train_loss': [], 'val_loss': [], 'learning_rate': []}
    best_val_loss = float('inf')

    print(f"Epochs: {config.num_epochs} | Patience: {config.patience}")
    print(f"Timesteps: {config.timesteps} | LR: {config.learning_rate}")

    for epoch in range(1, config.num_epochs + 1):
        # Training
        model.train()
```

```
        train_loss = 0.0
        pbar = tqdm(train_loader, desc=f'Epoch {epoch}/{config.num_epochs}', leave=False)

        for batch_idx, (data, _) in enumerate(pbar):
            data = data.to(device)
            batch_size = data.shape[0]

            # Sample random timesteps
            t = torch.randint(0, diffusion.timesteps, (batch_size,), device=device, dtype=torch.long)

            # Compute loss
            loss = diffusion.p_losses(model, data, t)

            optimizer.zero_grad()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), config.gradient_clip)
            optimizer.step()

            train_loss += loss.item()
            pbar.set_postfix({'loss': f'{loss.item():.4f}'})

        train_loss /= len(train_loader)

        # Validation
        model.eval()
        val_loss = 0.0
        with torch.no_grad():
            for data, _ in val_loader:
                data = data.to(device)
                batch_size = data.shape[0]
                t = torch.randint(0, diffusion.timesteps, (batch_size,), device=device, dtype=torch.long)
                loss = diffusion.p_losses(model, data, t)
                val_loss += loss.item()
        val_loss /= len(val_loader)

        # Update scheduler
        scheduler.step()
        current_lr = scheduler.get_last_lr()[0]

        # Record history
        history['train_loss'].append(train_loss)
        history['val_loss'].append(val_loss)
        history['learning_rate'].append(current_lr)

        # Print progress
        improved = ""
        if val_loss < best_val_loss:
            best_val_loss = val_loss
            improved = "New best!"

        print(f"Epoch [{epoch}/{config.num_epochs}] "
              f"Train: {train_loss:.4f} | Val: {val_loss:.4f} | LR: {current_lr:.2e}{improved}")

        # Early stopping
        if early_stopping(val_loss, epoch):
            print(f"\nEarly stopping at epoch {epoch}")
            print(f"Best epoch was {early_stopping.best_epoch} with val_loss: {early_stopping.best_loss:.4f}")
            break

    print(f"Training Complete! Best Val Loss: {best_val_loss:.4f}")
```

```
    return history

# Train the model
diffusion_history = train_diffusion_model(
    diffusion_model, diffusion, train_loader, test_loader, DIFFUSION_CONFIG, device
)
```

```
Epochs: 30 | Patience: 10
Timesteps: 1000 | LR: 0.0002
```

```
Epoch 1/30:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Epoch [1/30] Train: 0.0712 | Val: 0.0503 | LR: 1.99e-04New best!
```

```
Epoch 2/30:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Epoch [2/30] Train: 0.0494 | Val: 0.0483 | LR: 1.98e-04New best!
```

```
Epoch 3/30:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Epoch [3/30] Train: 0.0462 | Val: 0.0455 | LR: 1.95e-04New best!
```

```
Epoch 4/30:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Epoch [4/30] Train: 0.0440 | Val: 0.0455 | LR: 1.91e-04
```

```
Epoch 5/30:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Epoch [5/30] Train: 0.0431 | Val: 0.0442 | LR: 1.87e-04New best!
```

```
Epoch 6/30:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Epoch [6/30] Train: 0.0418 | Val: 0.0405 | LR: 1.81e-04New best!
```

```
Epoch 7/30:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Epoch [7/30] Train: 0.0409 | Val: 0.0409 | LR: 1.74e-04
```

```
Epoch 8/30:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Epoch [8/30] Train: 0.0407 | Val: 0.0411 | LR: 1.67e-04
```

```
Epoch 9/30:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Epoch [9/30] Train: 0.0405 | Val: 0.0404 | LR: 1.59e-04New best!
```

```
Epoch 10/30:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Epoch [10/30] Train: 0.0394 | Val: 0.0401 | LR: 1.50e-04New best!
```

```
Epoch 11/30:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Epoch [11/30] Train: 0.0399 | Val: 0.0396 | LR: 1.41e-04New best!
```

```
Epoch 12/30:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Epoch [12/30] Train: 0.0388 | Val: 0.0385 | LR: 1.31e-04New best!
```

```
Epoch 13/30:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Epoch [13/30] Train: 0.0387 | Val: 0.0400 | LR: 1.21e-04
```

```
Epoch 14/30:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Epoch [14/30] Train: 0.0386 | Val: 0.0402 | LR: 1.11e-04

Epoch 15/30:   0%|          | 0/938 [00:00<?, ?it/s]

Epoch [15/30] Train: 0.0383 | Val: 0.0381 | LR: 1.01e-04New best!

Epoch 16/30:   0%|          | 0/938 [00:00<?, ?it/s]

Epoch [16/30] Train: 0.0376 | Val: 0.0371 | LR: 9.01e-05New best!

Epoch 17/30:   0%|          | 0/938 [00:00<?, ?it/s]

Epoch [17/30] Train: 0.0372 | Val: 0.0383 | LR: 7.98e-05

Epoch 18/30:   0%|          | 0/938 [00:00<?, ?it/s]

Epoch [18/30] Train: 0.0374 | Val: 0.0371 | LR: 6.98e-05

Epoch 19/30:   0%|          | 0/938 [00:00<?, ?it/s]

Epoch [19/30] Train: 0.0378 | Val: 0.0374 | LR: 6.00e-05

Epoch 20/30:   0%|          | 0/938 [00:00<?, ?it/s]

Epoch [20/30] Train: 0.0375 | Val: 0.0370 | LR: 5.08e-05New best!

Epoch 21/30:   0%|          | 0/938 [00:00<?, ?it/s]

Epoch [21/30] Train: 0.0369 | Val: 0.0383 | LR: 4.20e-05

Epoch 22/30:   0%|          | 0/938 [00:00<?, ?it/s]

Epoch [22/30] Train: 0.0372 | Val: 0.0384 | LR: 3.39e-05

Epoch 23/30:   0%|          | 0/938 [00:00<?, ?it/s]

Epoch [23/30] Train: 0.0371 | Val: 0.0384 | LR: 2.66e-05

Epoch 24/30:   0%|          | 0/938 [00:00<?, ?it/s]

Epoch [24/30] Train: 0.0364 | Val: 0.0366 | LR: 2.00e-05New best!

Epoch 25/30:   0%|          | 0/938 [00:00<?, ?it/s]

Epoch [25/30] Train: 0.0365 | Val: 0.0366 | LR: 1.43e-05

Epoch 26/30:   0%|          | 0/938 [00:00<?, ?it/s]

Epoch [26/30] Train: 0.0367 | Val: 0.0364 | LR: 9.60e-06New best!

Epoch 27/30:   0%|          | 0/938 [00:00<?, ?it/s]

Epoch [27/30] Train: 0.0369 | Val: 0.0381 | LR: 5.87e-06

Epoch 28/30:   0%|          | 0/938 [00:00<?, ?it/s]

Epoch [28/30] Train: 0.0368 | Val: 0.0377 | LR: 3.17e-06

Epoch 29/30:   0%|          | 0/938 [00:00<?, ?it/s]

Epoch [29/30] Train: 0.0366 | Val: 0.0369 | LR: 1.55e-06

Epoch 30/30:   0%|          | 0/938 [00:00<?, ?it/s]
```

```
Epoch [30/30] Train: 0.0364 | Val: 0.0370 | LR: 1.00e-06
Training Complete! Best Val Loss: 0.0364
```

31.6.5. 5.5 Visualizing the Forward Diffusion

The forward process demonstrates how clean images are progressively corrupted into pure Gaussian noise. At t=0 we have the original image, and by t=999 it becomes indistinguishable from random noise.

```python
def visualize_forward_diffusion(diffusion, image, timesteps_to_show=[0, 250, 500, 750, 999]):
    """Visualize the forward diffusion process (adding noise)."""
    fig, axes = plt.subplots(1, len(timesteps_to_show), figsize=(3 * len(timesteps_to_show), 3))

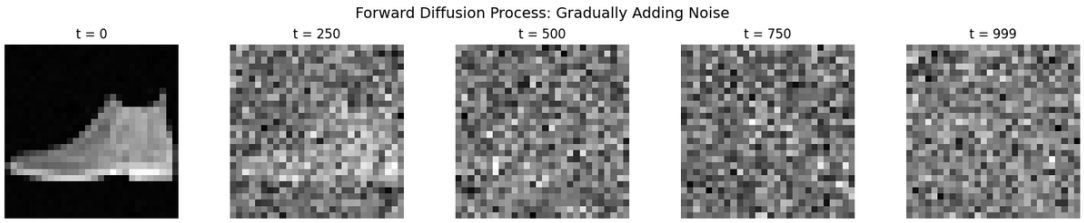
    for i, t in enumerate(timesteps_to_show):
        t_tensor = torch.tensor([t], device=diffusion.device)
        noisy_image, _ = diffusion.q_sample(image.unsqueeze(0), t_tensor)

        img = noisy_image.squeeze().cpu().numpy()
        img = (img + 1) / 2  # Denormalize from [-1, 1] to [0, 1]

        axes[i].imshow(img, cmap='gray')
        axes[i].set_title(f't = {t}')
        axes[i].axis('off')

    plt.suptitle('Forward Diffusion Process: Gradually Adding Noise', fontsize=14)
    plt.tight_layout()
    plt.show()

# Visualize forward diffusion on a test sample
sample_image = test_dataset[0][0].to(device)
visualize_forward_diffusion(diffusion, sample_image)
```



31.6.6. 5.6 Training Progress

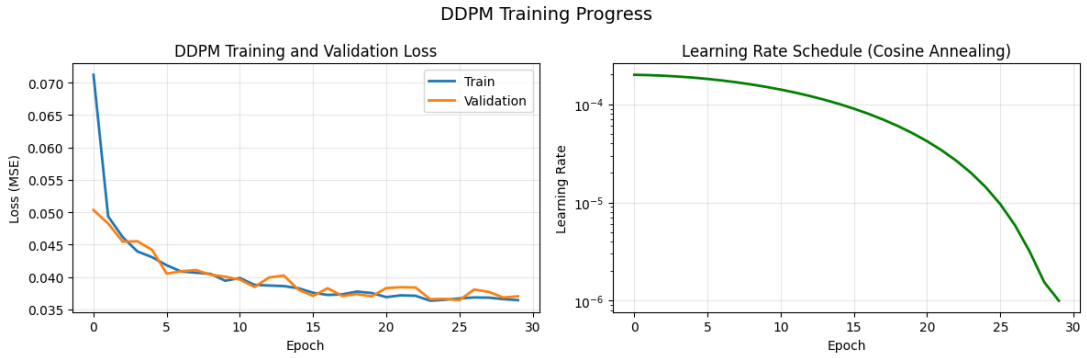
We plot the training and validation loss curves along with the learning rate schedule. The loss should decrease steadily as the model learns to predict noise more accurately.

```python
fig, axes = plt.subplots(1, 2, figsize=(12, 4))

# Loss curves
axes[0].plot(diffusion_history['train_loss'], label='Train', linewidth=2)
axes[0].plot(diffusion_history['val_loss'], label='Validation', linewidth=2)
axes[0].set_xlabel('Epoch')
axes[0].set_ylabel('Loss (MSE)')
axes[0].set_title('DDPM Training and Validation Loss')
axes[0].legend()
axes[0].grid(True, alpha=0.3)
```

```
# Learning rate
axes[1].plot(diffusion_history['learning_rate'], linewidth=2, color='green')
axes[1].set_xlabel('Epoch')
axes[1].set_ylabel('Learning Rate')
axes[1].set_title('Learning Rate Schedule (Cosine Annealing)')
axes[1].grid(True, alpha=0.3)
axes[1].set_yscale('log')

plt.suptitle('DDPM Training Progress', fontsize=14)
plt.tight_layout()
plt.show()
```



31.6.7. 5.7 Generating Samples with DDPM

DDPM generates images by starting from pure noise and iteratively denoising for 1000 steps. This is the original sampling method from the DDPM paper - high quality but slow.

```
diffusion_model.eval()
with torch.no_grad():
    ddpm_samples = diffusion.sample(
        diffusion_model,
        batch_size=64,
        channels=DIFFUSION_CONFIG.image_channels,
        image_size=DIFFUSION_CONFIG.image_size
    )

# Visualize DDPM samples
def visualize_samples(samples, nrow=8, title="Generated Samples"):
    """Visualize generated samples in a grid."""
    n_samples = samples.shape[0]
    ncol = min(nrow, n_samples)
    nrow_actual = (n_samples + ncol - 1) // ncol

    fig, axes = plt.subplots(nrow_actual, ncol, figsize=(ncol * 1.5, nrow_actual * 1.5))
    axes = np.atleast_2d(axes)

    for i in range(nrow_actual):
        for j in range(ncol):
            idx = i * ncol + j
            if idx < n_samples:
                img = samples[idx].squeeze().cpu().numpy()
                img = (img + 1) / 2  # Denormalize
```

```
                img = np.clip(img, 0, 1)
                axes[i, j].imshow(img, cmap='gray')
            axes[i, j].axis('off')

    plt.suptitle(title, fontsize=14)
    plt.tight_layout()
    plt.show()

visualize_samples(ddpm_samples, nrow=8, title="DDPM Generated Samples (1000 steps)")
```

```
DDPM Sampling:   0%|          | 0/1000 [00:00<?, ?it/s]
```



31.6.8. 5.8 Accelerated Sampling with DDIM

DDIM (Denoising Diffusion Implicit Models) provides a deterministic sampling method that can skip timesteps. With just 50 steps instead of 1000, we achieve ~20x speedup with minimal quality loss.

```
with torch.no_grad():
    ddim_samples = diffusion.ddim_sample(
        diffusion_model,
```

```
        batch_size=64,
        channels=DIFFUSION_CONFIG.image_channels,
        image_size=DIFFUSION_CONFIG.image_size,
        ddim_steps=50,
        eta=0.0  # Deterministic sampling
    )

visualize_samples(ddim_samples, nrow=8, title="DDIM Generated Samples (50 steps)")
```

```
DDIM Sampling:   0%|          | 0/50 [00:00<?, ?it/s]
```



31.6.9. 5.9 Visualizing the Reverse Diffusion

The reverse process shows how the model progressively denoises random noise into recognizable images. This is the core innovation of diffusion models - learning to reverse the corruption process.

```
with torch.no_grad():
    samples_all_steps = diffusion.sample(
        diffusion_model,
        batch_size=1,
```

```
        channels=DIFFUSION_CONFIG.image_channels,
        image_size=DIFFUSION_CONFIG.image_size,
        return_all_timesteps=True
    )

# Select timesteps to visualize
total_steps = samples_all_steps.shape[1]
n_steps_to_show = 10
indices = np.linspace(0, total_steps - 1, n_steps_to_show, dtype=int)

fig, axes = plt.subplots(1, n_steps_to_show, figsize=(2 * n_steps_to_show, 2))

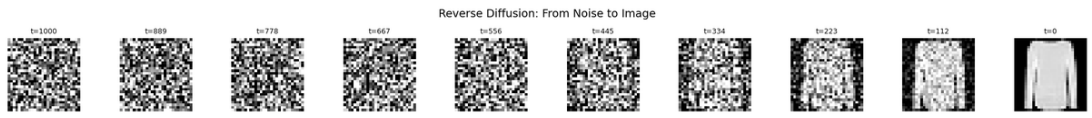
for i, idx in enumerate(indices):
    img = samples_all_steps[0, idx].squeeze().cpu().numpy()
    img = (img + 1) / 2
    img = np.clip(img, 0, 1)

    axes[i].imshow(img, cmap='gray')
    t_value = DIFFUSION_CONFIG.timesteps - idx
    axes[i].set_title(f't={t_value}', fontsize=9)
    axes[i].axis('off')

plt.suptitle('Reverse Diffusion: From Noise to Image', fontsize=14)
plt.tight_layout()
plt.show()

print(f"  - DDPM generates images through {DIFFUSION_CONFIG.timesteps} denoising steps")
print(f"  - DDIM accelerates this to just 50 steps with minimal quality loss")
```

```
DDPM Sampling:   0%|          | 0/1000 [00:00<?, ?it/s]
```




```
- DDPM generates images through 1000 denoising steps
  - DDIM accelerates this to just 50 steps with minimal quality loss
```

31.6.10. HOMEWORK (Optional)

- **Exercise 1: Compare Latent Dimensions**
  Train the Autoencoder and VAE with different latent dimensions (e.g., 2, 8, 32, 64). Compare reconstruction quality and latent space structure. Discuss the trade-off between compression and reconstruction fidelity.
- **Exercise 2: Conditional GAN (cGAN)**
  Modify the DCGAN to accept class labels as additional input to both Generator and Discriminator. Train a conditional GAN that can generate specific clothing categories on demand. Evaluate if conditioning improves sample quality.
- **Exercise 3: Implement DDPM Training**
  Extend the diffusion section by implementing a full **Denoising Diffusion Probabilistic Model**

(DDPM) training loop on Fashion MNIST. Train the model to predict noise and generate new samples through iterative denoising. Compare results with GAN-generated images.

Chapter 32

Neural Rendering

Originally developed as a Google Colab notebook:
https://colab.research.google.com/drive/1cYwiBN4mr8ZK-CCrVqYag4TdOmud0r0H?usp=sharing

Neural rendering combines classical computer graphics with deep learning to synthesize **photorealistic images from novel viewpoints**. This notebook explores the complete pipeline from **Structure from Motion (SfM)** for camera pose estimation, through **Neural Radiance Fields (NeRF [27])** for volumetric scene representation, to **3D Gaussian Splatting [28]** for real-time rendering.

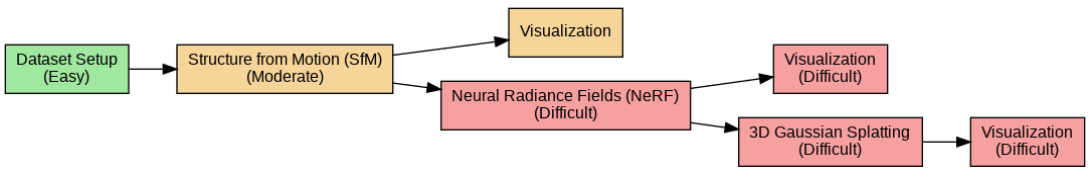
32.1. Necessary Imports

```
import os
import numpy as np
import requests
import zipfile
import io
import cv2
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
from tqdm import tqdm
from IPython.display import Image as IMG, IFrame
import imageio

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.device("cpu")
print(f"Using device: {device}")
```

```
Using device: cuda:0
```

```
IMG('neural_rendering_flowchart.png')
```



32.2. Loading Our Dataset

We'll use the **Temple Ring** dataset from Middlebury for our SfM demonstration.

```
image_dir = "templeRing"

if not os.path.exists(image_dir):
```

```
    print(f"Downloading '{image_dir}' dataset...")
    url = "https://vision.middlebury.edu/mview/data/data/templeRing.zip"
    r = requests.get(url)
    z = zipfile.ZipFile(io.BytesIO(r.content))
    z.extractall()
    print("Dataset downloaded and extracted.")
else:
    print(f"Dataset '{image_dir}' already exists.")

image_paths = sorted([os.path.join(image_dir, f) for f in os.listdir(image_dir) if f.endswith('.png')])
image_paths = image_paths[:8]

print(f"\nLoaded {len(image_paths)} images for the pipeline.")
```

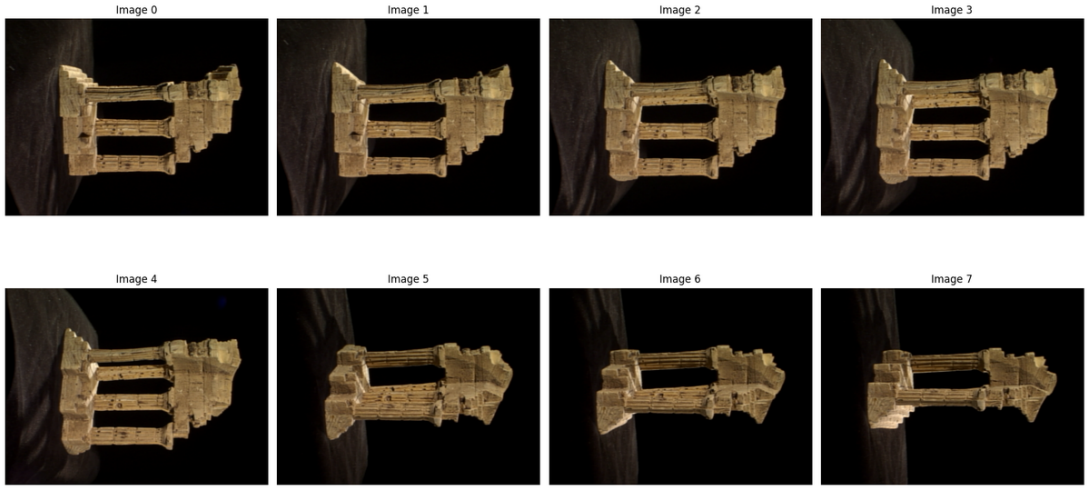
```
Downloading 'templeRing' dataset...
Dataset downloaded and extracted.

Loaded 8 images for the pipeline.
```

```
fig, axes = plt.subplots(2, 4, figsize=(18, 10))
axes = axes.ravel()

for i in range(len(image_paths)):
    img = cv2.imread(image_paths[i])
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    axes[i].imshow(img_rgb)
    axes[i].set_title(f"Image {i}")
    axes[i].axis('off')

plt.tight_layout()
plt.show()
```



32.2.1. 1.3 Core Reconstruction Functions

We implement two critical SfM functions:

1. **`match_features()`**: Detects SIFT keypoints, performs brute-force k-NN matching, and applies Lowe's ratio test to filter ambiguous matches

2. **`estimate_motion()`**: Computes the Essential matrix with RANSAC, recovers relative pose using the cheirality check (ensures points lie in front of both cameras), and triangulates valid 3D points

```
def match_features(img1_gray, img2_gray):
    """Finds and filters feature matches between two images."""
    sift = cv2.SIFT_create()
    kp1, des1 = sift.detectAndCompute(img1_gray, None)
    kp2, des2 = sift.detectAndCompute(img2_gray, None)

    bf = cv2.BFMatcher()
    matches = bf.knnMatch(des1, des2, k=2)

    good_matches = []
    for m, n in matches:
        if m.distance < 0.75 * n.distance:
            good_matches.append(m)

    pts1 = np.float32([kp1[m.queryIdx].pt for m in good_matches])
    pts2 = np.float32([kp2[m.trainIdx].pt for m in good_matches])

    return pts1, pts2

def estimate_motion(pts1, pts2, camera_matrix):
    """Estimates camera motion and triangulates 3D points with cheirality check."""
    E, inlier_mask_E = cv2.findEssentialMat(pts1, pts2, camera_matrix,
                                            method=cv2.RANSAC, prob=0.999, threshold=1.0)

    if E is None:
        print("   x Could not find Essential Matrix. Skipping pair.")
        return None, None, None, None

    pts1_inliers = pts1[inlier_mask_E.ravel() == 1]
    pts2_inliers = pts2[inlier_mask_E.ravel() == 1]

    _, R, t, pose_mask = cv2.recoverPose(E, pts1_inliers, pts2_inliers, camera_matrix)

    if pose_mask is None:
        return None, None, None, None

    final_pts1 = pts1_inliers[pose_mask.ravel() > 0]
    final_pts2 = pts2_inliers[pose_mask.ravel() > 0]

    proj_mat1 = camera_matrix @ np.hstack((np.eye(3), np.zeros((3, 1))))
    proj_mat2 = camera_matrix @ np.hstack((R, t))

    points_4d_hom = cv2.triangulatePoints(proj_mat1, proj_mat2, final_pts1.T, final_pts2.T)
    points_3d = (points_4d_hom / points_4d_hom[3])[:3, :].T

    return R, t, points_3d, final_pts2

print("Core reconstruction functions defined.")
```

```
Core reconstruction functions defined.
```

32.2.2. 1.4 Incremental SfM Pipeline

We implement a sequential SfM system that processes consecutive image pairs: 1. Start with image 0 as the world origin (R = I, t = 0) 2. For each pair: match features -> estimate relative pose -> triangulate 3D points 3. Accumulate global camera poses and transform points to world coordinates

```
print("\n--- Starting the N-View Structure from Motion Pipeline ---")

img = cv2.imread(image_paths[0])
h, w, _ = img.shape
camera_matrix = np.array([[w, 0, w/2], [0, w, h/2], [0, 0, 1]], dtype=np.float32)

R_global = np.eye(3)
t_global = np.zeros((3, 1))
all_points_3d = np.zeros((0, 3))
camera_poses = [(R_global, t_global)]

img1_gray = cv2.imread(image_paths[0], cv2.IMREAD_GRAYSCALE)

for i in range(1, len(image_paths)):
    print(f"Processing image pair ({i-1}, {i})...")
    img2_gray = cv2.imread(image_paths[i], cv2.IMREAD_GRAYSCALE)

    pts1, pts2 = match_features(img1_gray, img2_gray)
    if len(pts1) < 20:
        print("   x Not enough matches. Skipping pair.")
        continue

    R_rel, t_rel, points_3d_local, _ = estimate_motion(pts1, pts2, camera_matrix)

    if R_rel is None:
        continue

    t_global = t_global + R_global @ t_rel
    R_global = R_global @ R_rel
    camera_poses.append((R_global, t_global))

    points_3d_global = R_global @ points_3d_local.T + t_global
    all_points_3d = np.vstack((all_points_3d, points_3d_global.T))

    img1_gray = img2_gray

print("\nPipeline finished.")
print(f"Reconstructed {len(all_points_3d)} points.")
print(f"Estimated {len(camera_poses)} camera poses.")
```

```
--- Starting the N-View Structure from Motion Pipeline ---
Processing image pair (0, 1)...
Processing image pair (1, 2)...
Processing image pair (2, 3)...
Processing image pair (3, 4)...
Processing image pair (4, 5)...
Processing image pair (5, 6)...
Processing image pair (6, 7)...

Pipeline finished.
Reconstructed 2081 points.
Estimated 8 camera poses.
```

```
print("\n--- Visualizing Feature Matches for the first 3 pairs ---")

fig, axes = plt.subplots(3, 1, figsize=(20, 30))

for i in range(3):
    img1_color = cv2.imread(image_paths[i])
    img2_color = cv2.imread(image_paths[i+1])
    img1_gray = cv2.cvtColor(img1_color, cv2.COLOR_BGR2GRAY)
    img2_gray = cv2.cvtColor(img2_color, cv2.COLOR_BGR2GRAY)

    sift = cv2.SIFT_create()
    kp1, des1 = sift.detectAndCompute(img1_gray, None)
    kp2, des2 = sift.detectAndCompute(img2_gray, None)

    bf = cv2.BFMatcher()
    matches = bf.knnMatch(des1, des2, k=2)

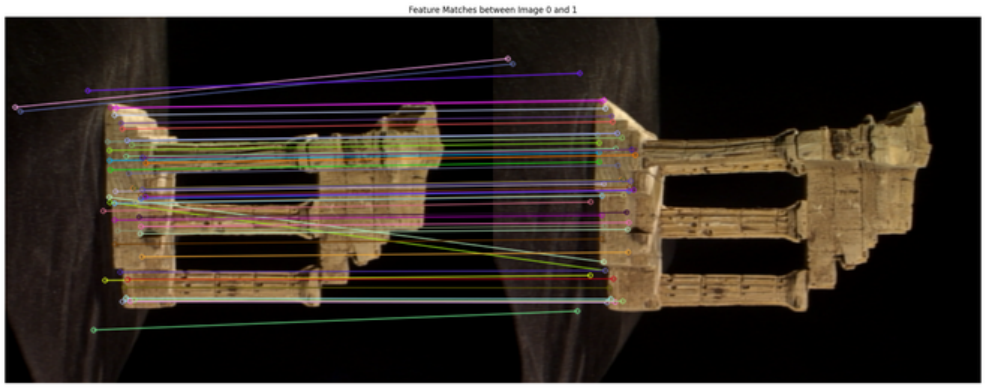
    good_matches = [m for m, n in matches if m.distance < 0.75 * n.distance]

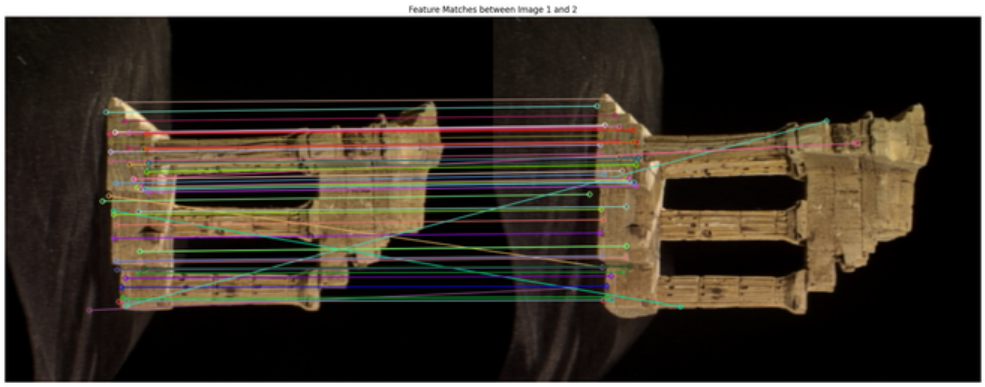
    img_matches = cv2.drawMatches(img1_color, kp1, img2_color, kp2, good_matches[:50],
                                  None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS)

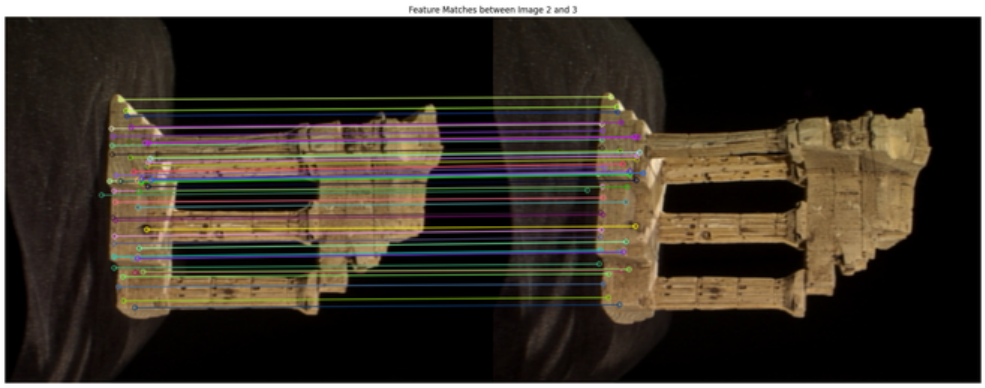
    axes[i].imshow(cv2.cvtColor(img_matches, cv2.COLOR_BGR2RGB))
    axes[i].set_title(f"Feature Matches between Image {i} and {i+1}")
    axes[i].axis('off')

plt.tight_layout()
plt.show()
```

```
--- Visualizing Feature Matches for the first 3 pairs ---
```







32.2.3. 1.5 Visualization of the Reconstructed Scene

Now that the incremental SfM pipeline has produced a sparse 3D point cloud and a set of camera poses, we visualize the results in multiple complementary ways:

- **Interactive 3D view** (Matplotlib): scatter plot of the point cloud + small colored coordinate frames (R-G-B) at each camera center

- **Standard orthographic projections** (top, side, front) for quick inspection of trajectory shape and drift

- **Camera frustum visualization** (pyramid style): gives an intuitive sense of where each camera was looking

These plots are essential for qualitatively evaluating reconstruction quality before moving on to bundle adjustment or dense reconstruction.

```
print("\n--- Visualizing the Final Scene with Matplotlib ---")

fig = plt.figure(figsize=(12, 12))
ax = fig.add_subplot(111, projection='3d')

num_points_to_plot = 5000
if len(all_points_3d) > num_points_to_plot:
    plot_points = all_points_3d[np.random.choice(len(all_points_3d), num_points_to_plot, replace=False)]
else:
    plot_points = all_points_3d

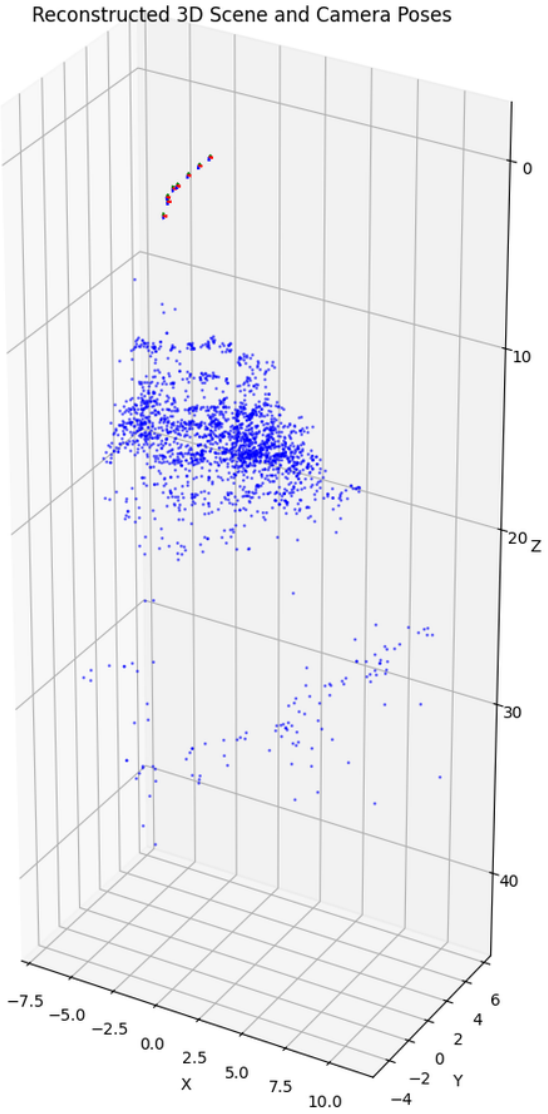
ax.scatter(plot_points[:, 0], plot_points[:, 1], plot_points[:, 2], s=1, c='b', alpha=0.5)

pose_scale = 0.2
for R, t in camera_poses:
    origin = t.ravel()
    x_axis = R[:, 0]
    y_axis = R[:, 1]
    z_axis = R[:, 2]
    ax.quiver(origin[0], origin[1], origin[2], x_axis[0], x_axis[1], x_axis[2], color='r', length=pose_scale)
    ax.quiver(origin[0], origin[1], origin[2], y_axis[0], y_axis[1], y_axis[2], color='g', length=pose_scale)
    ax.quiver(origin[0], origin[1], origin[2], z_axis[0], z_axis[1], z_axis[2], color='b', length=pose_scale)

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('Reconstructed 3D Scene and Camera Poses')
ax.set_aspect('equal', adjustable='box')
ax.invert_zaxis()

print("Displaying 3D point cloud. You can rotate the plot within this cell's output.")
plt.show()
```

```
--- Visualizing the Final Scene with Matplotlib ---
Displaying 3D point cloud. You can rotate the plot within this cell's output.
```




```
print("\n--- Visualizing the Reconstruction from Standard Angles ---")

fig = plt.figure(figsize=(24, 8))

ax1 = fig.add_subplot(131, projection='3d')
ax1.scatter(all_points_3d[:, 0], all_points_3d[:, 1], all_points_3d[:, 2], s=1, c='b', alpha=0.3)
for R, t in camera_poses:
    ax1.quiver(t[0], t[1], t[2], R[0, 2], R[1, 2], R[2, 2], color='r', length=1)
ax1.set_title('Top-Down View')
ax1.set_xlabel('X'); ax1.set_ylabel('Y'); ax1.set_zlabel('Z')
ax1.view_init(elev=90, azim=-90)
ax1.set_aspect('equal', adjustable='box')

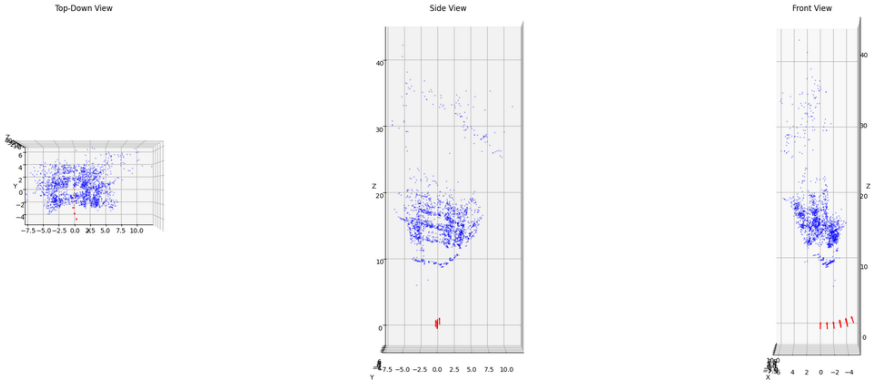
ax2 = fig.add_subplot(132, projection='3d')
ax2.scatter(all_points_3d[:, 0], all_points_3d[:, 1], all_points_3d[:, 2], s=1, c='b', alpha=0.3)
for R, t in camera_poses:
    ax2.quiver(t[0], t[1], t[2], R[0, 2], R[1, 2], R[2, 2], color='r', length=1)
ax2.set_title('Side View')
ax2.set_xlabel('X'); ax2.set_ylabel('Y'); ax2.set_zlabel('Z')
```

```
ax2.view_init(elev=0, azim=-90)
ax2.set_aspect('equal', adjustable='box')

ax3 = fig.add_subplot(133, projection='3d')
ax3.scatter(all_points_3d[:, 0], all_points_3d[:, 1], all_points_3d[:, 2], s=1, c='b', alpha=0.3)
for R, t in camera_poses:
    ax3.quiver(t[0], t[1], t[2], R[0, 2], R[1, 2], R[2, 2], color='r', length=1)
ax3.set_title('Front View')
ax3.set_xlabel('X'); ax3.set_ylabel('Y'); ax3.set_zlabel('Z')
ax3.view_init(elev=0, azim=180)
ax3.set_aspect('equal', adjustable='box')

plt.tight_layout()
plt.show()
```

```
--- Visualizing the Reconstruction from Standard Angles ---
```




```
print("\n--- Advanced Visualization: Camera Frustum Style ---")

fig = plt.figure(figsize=(15, 12))
ax = fig.add_subplot(111, projection='3d')

ax.scatter(all_points_3d[:, 0], all_points_3d[:, 1], all_points_3d[:, 2], s=1, c='b', alpha=0.1)

frustum_scale = 0.3
frustum_points = np.array([
    [0, 0, 0],
    [-1, -0.75, 1.5],
    [1, -0.75, 1.5],
    [1, 0.75, 1.5],
    [-1, 0.75, 1.5]
]) * frustum_scale

frustum_lines = [[0, 1], [0, 2], [0, 3], [0, 4],
                 [1, 2], [2, 3], [3, 4], [4, 1]]

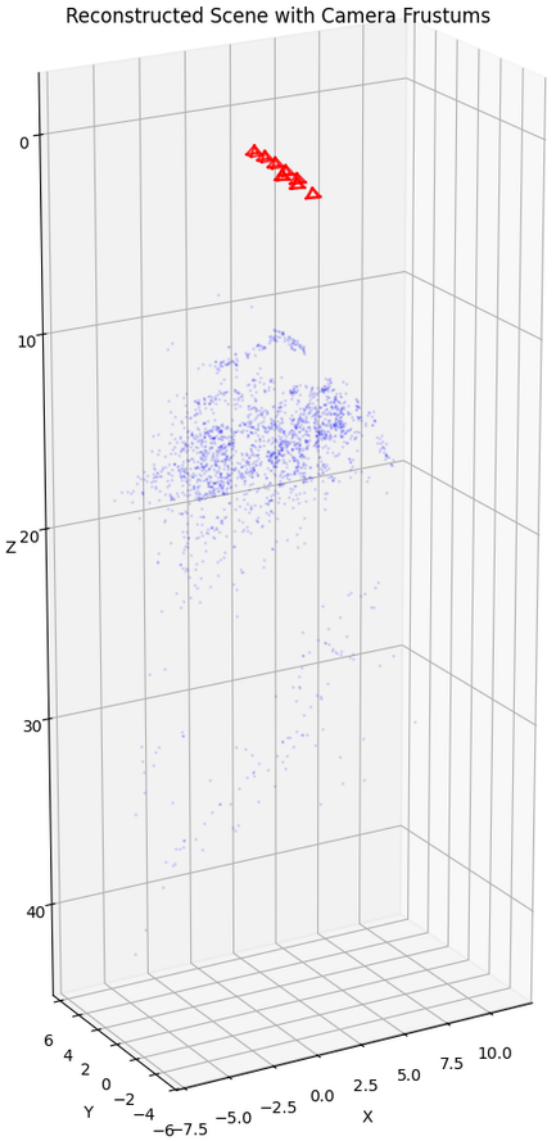
for R, t in camera_poses:
    transformed_points = (R.T @ frustum_points.T).T + t.T
    for line in frustum_lines:
        p1 = transformed_points[line[0]]
        p2 = transformed_points[line[1]]
        ax.plot([p1[0], p2[0]], [p1[1], p2[1]], [p1[2], p2[2]], color='r', alpha=0.8)

ax.set_xlabel('X')
```

```
ax.set_ylabel('Y')
ax.set_zlabel('Z')
ax.set_title('Reconstructed Scene with Camera Frustums')
ax.set_aspect('equal', adjustable='box')
ax.view_init(elev=20, azim=-120)
ax.invert_zaxis()

plt.show()
```

```
--- Advanced Visualization: Camera Frustum Style ---
```



32.3. 2. Neural Radiance Fields (NeRF)

32.3.1. 2.1 Introduction to NeRF

NeRF (Mildenhall et al., ECCV 2020) learns a continuous 3D scene representation directly from photographs using a neural network.

Core Idea: A small MLP is trained to answer: *"If I stand at 3D position* $\boldsymbol{x}$ *and look in direction* $\boldsymbol{d}$*, what color (RGB) and density (*α*) do I see?"*

Input: 5D coordinate (x, y, z, θ, ϕ), position + viewing direction
Output: RGB color + volume density σ

Rendering Process: - Shoot a virtual ray from the camera through each pixel
- Sample many points along the ray
- Query the NeRF network at each sample
- Composite color using volume rendering

The result: **photorealistic novel-view synthesis** with perfect parallax, reflections, and fine details.

Aspect	SfM + MVS	NeRF
Output	Sparse/dense point cloud + mesh	Continuous volumetric function
Novel views	Limited by geometry accuracy	Photorealistic
Reflections/shadows	Usually missing	Naturally captured
Training required	No	Yes (minutes to hours)

32.3.2. 2.2 NeRF Setup

```
print("\n--- Loading Tiny NeRF Dataset ---")

if not os.path.exists('tiny_nerf_data.npz'):
    !wget http://cseweb.ucsd.edu/~viscomp/projects/LF/papers/ECCV20/nerf/tiny_nerf_data.npz

data = np.load('tiny_nerf_data.npz')
images = data['images']
poses = data['poses']
focal = data['focal']

print(f'Images shape: {images.shape}')
print(f'Poses shape: {poses.shape}')
print(f'Focal length: {focal}')

height, width = images.shape[1:3]
near, far = 2., 6.

n_training = 100
testimg_idx = 101
testimg, testpose = images[testimg_idx], poses[testimg_idx]

plt.figure(figsize=(10, 10))
plt.imshow(testimg)
plt.title('Test Image (Index 101)')
plt.axis('off')
plt.show()

print('\nTest Pose:')
print(testpose)
print('\nTiny NeRF dataset loaded successfully.')
```

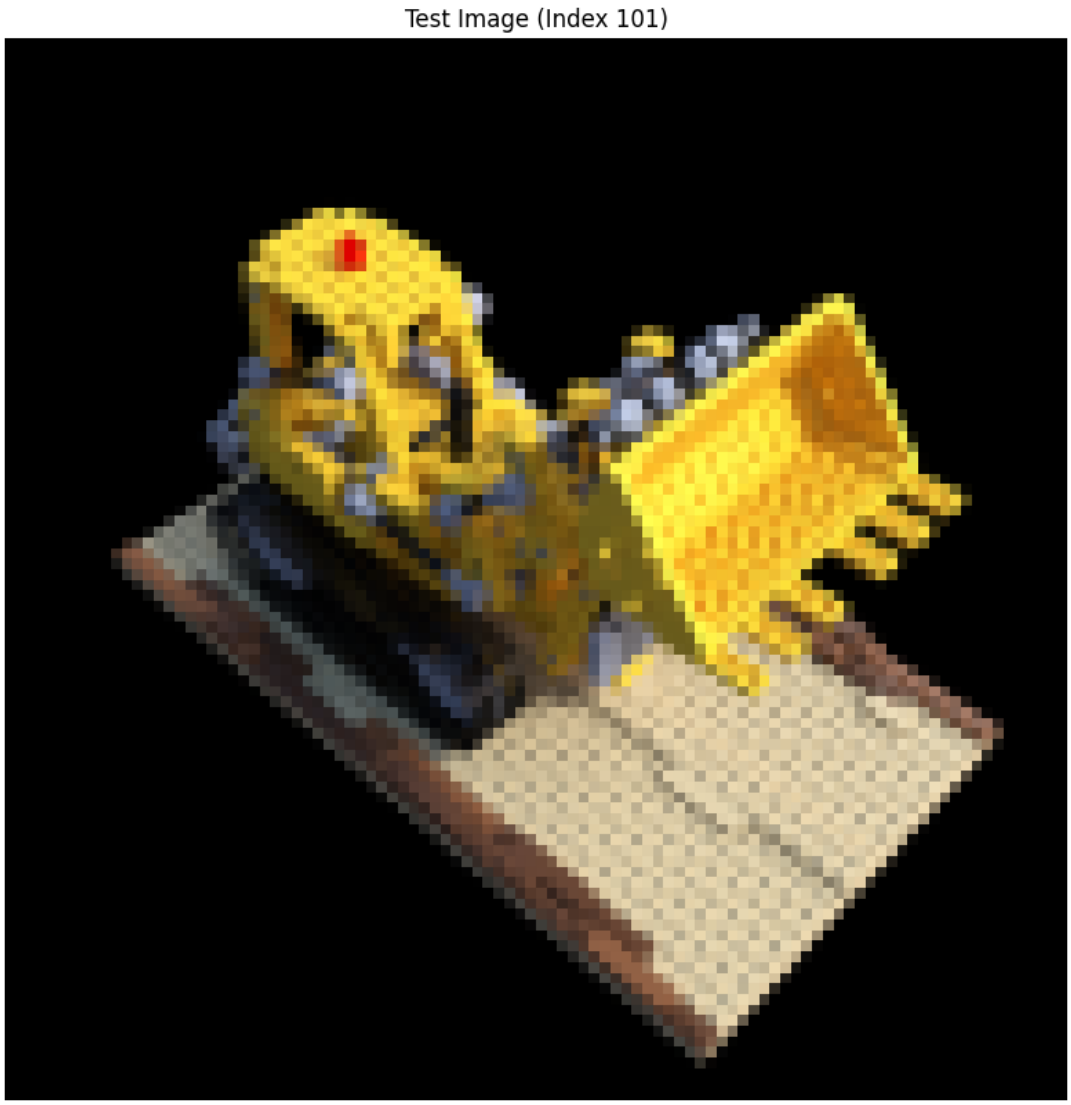
```
--- Loading Tiny NeRF Dataset ---
--2026-05-09 18:51:53--  http://cseweb.ucsd.edu/~viscomp/projects/LF/papers/ECCV
20/nerf/tiny_nerf_data.npz
```

```
Resolving cseweb.ucsd.edu (cseweb.ucsd.edu)... 132.239.8.30
Connecting to cseweb.ucsd.edu (cseweb.ucsd.edu)|132.239.8.30|:80... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://cseweb.ucsd.edu//~viscomp/projects/LF/papers/ECCV20/nerf/tiny_
nerf_data.npz [following]
--2026-05-09 18:51:53--  https://cseweb.ucsd.edu//~viscomp/projects/LF/papers/EC
CV20/nerf/tiny_nerf_data.npz
Connecting to cseweb.ucsd.edu (cseweb.ucsd.edu)|132.239.8.30|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 12727482 (12M)
Saving to: ‘tiny_nerf_data.npz’

tiny_nerf_data.npz  100%[===================>]  12.14M  31.7MB/s    in 0.4s

2026-05-09 18:51:53 (31.7 MB/s) - ‘tiny_nerf_data.npz’ saved [12727482/12727482]

Images shape: (106, 100, 100, 3)
Poses shape: (106, 4, 4)
Focal length: 138.88887889922103
```



```
Test Pose:
[[ 6.8935126e-01  5.3373039e-01 -4.8982298e-01 -1.9745398e+00]
 [-7.2442728e-01  5.0788772e-01 -4.6610624e-01 -1.8789345e+00]
 [ 1.4901163e-08  6.7615211e-01  7.3676193e-01  2.9699826e+00]
 [ 0.0000000e+00  0.0000000e+00  0.0000000e+00  1.0000000e+00]]

Tiny NeRF dataset loaded successfully.
```

32.3.3. 2.3 The Core NeRF Model + Positional Encoding

Key Components:

1. **Positional Encoding:** Maps low-frequency coordinates through sinusoids to enable high-frequency detail representation.
2. **NeRF MLP (8-layer, 256-channel):**
   - Main trunk processes positionally encoded 3D points -> predicts density σ and feature vector
   - Skip connections at layer 5 preserve high-frequency geometry
   - View-dependent branch takes feature + encoded direction -> predicts RGB color

Output per queried point: - `rgb` $\in [0,1]^3$ – emitted color - `sigma` ≥ 0 – volume density

```python
class PositionalEncoder(nn.Module):
    def __init__(self, d_input, n_freqs, log_space=False):
        super().__init__()
        self.d_input = d_input
        self.n_freqs = n_freqs
        self.log_space = log_space
        self.d_output = d_input * (1 + 2 * self.n_freqs)
        self.embed_fns = [lambda x: x]

        if self.log_space:
            freq_bands = 2.**torch.linspace(0., self.n_freqs - 1, self.n_freqs)
        else:
            freq_bands = torch.linspace(1., 2.**(self.n_freqs - 1), self.n_freqs)

        for freq in freq_bands:
            self.embed_fns.append(lambda x, freq=freq: torch.sin(x * freq))
            self.embed_fns.append(lambda x, freq=freq: torch.cos(x * freq))

    def forward(self, x):
        return torch.concat([fn(x) for fn in self.embed_fns], dim=-1)

class NeRF(nn.Module):
    def __init__(self, d_input=3, n_layers=8, d_filter=256, skip=(4,), d_viewdirs=3):
        super().__init__()
        self.d_input = d_input
        self.skip = skip
        self.act = nn.functional.relu

        self.encoder = PositionalEncoder(d_input, 10)
        self.encoder_viewdirs = PositionalEncoder(d_viewdirs, 4)

        self.layers = nn.ModuleList(
            [nn.Linear(self.encoder.d_output, d_filter)] +
            [nn.Linear(d_filter, d_filter) if i not in skip else
             nn.Linear(d_filter + self.encoder.d_output, d_filter) for i in range(n_layers - 1)]
```

```
        )

        self.sigma_out = nn.Linear(d_filter, 1)
        self.rgb_filters = nn.Linear(d_filter, d_filter)
        self.branch = nn.Linear(d_filter + self.encoder_viewdirs.d_output, d_filter // 2)
        self.rgb_out = nn.Linear(d_filter // 2, 3)

    def forward(self, x, viewdirs):
        x_encoded = self.encoder(x)
        viewdirs_encoded = self.encoder_viewdirs(viewdirs)

        y = x_encoded
        for i, layer in enumerate(self.layers):
            y = self.act(layer(y))
            if i in self.skip:
                y = torch.cat([y, x_encoded], dim=-1)

        sigma = self.sigma_out(y)

        y = self.rgb_filters(y)
        y = torch.concat([y, viewdirs_encoded], dim=-1)
        y = self.act(self.branch(y))
        rgb = self.rgb_out(y)

        return rgb, sigma

print("NeRF Model and PositionalEncoder defined.")
```

```
NeRF Model and PositionalEncoder defined.
```

32.3.4. 2.4 Ray Generation & Volumetric Rendering

Two fundamental components:

1. **`get_rays()`** – Converts pixel coordinates into 3D rays originating from camera center.
2. **`render_rays()`** – Neural volume rendering using the classic equation:
    - Sample points along each ray between `near` and `far` bounds
    - Query the NeRF model -> get RGB color and volume density (σ)
    - Compute transmittance and rendering weights
    - Accumulate weighted colors -> final pixel color

This function is **fully differentiable** -> gradients flow from rendered pixels back through the NeRF MLP.

```
def get_rays(height, width, focal_length, c2w):
    i, j = torch.meshgrid(
        torch.arange(width, dtype=torch.float32, device=c2w.device),
        torch.arange(height, dtype=torch.float32, device=c2w.device),
        indexing='xy')

    dirs = torch.stack([(i - float(width) * 0.5) / focal_length,
                        -(j - float(height) * 0.5) / focal_length,
                        -torch.ones_like(i)], dim=-1)

    rays_d = torch.sum(dirs[..., None, :] * c2w[:3, :3], dim=-1)
    rays_o = c2w[:3, -1].expand(rays_d.shape)
    return rays_o, rays_d
```

```
def render_rays(model, rays_o, rays_d, near, far, n_samples=64, rand=False):
    z_vals = torch.linspace(near, far, n_samples, device=rays_o.device)
    if rand:
        z_vals = z_vals + torch.rand(list(rays_o.shape[:-1]) + [n_samples], device=rays_o.device) * (far -
  near) / n_samples

    pts = rays_o[..., None, :] + rays_d[..., None, :] * z_vals[..., :, None]

    viewdirs = rays_d / torch.norm(rays_d, dim=-1, keepdim=True)
    viewdirs = viewdirs[..., None, :].expand(pts.shape)

    rgb, sigma = model(pts.reshape(-1, 3), viewdirs.reshape(-1, 3))
    rgb = torch.sigmoid(rgb).reshape(pts.shape)
    sigma = torch.relu(sigma).reshape(pts.shape[:-1])

    deltas = z_vals[..., 1:] - z_vals[..., :-1]
    delta_inf = torch.tensor([1e10], device=rays_o.device, dtype=deltas.dtype)
    deltas = torch.cat([deltas, delta_inf.expand(deltas[..., :1].shape)], dim=-1)

    alphas = 1.0 - torch.exp(-sigma * deltas)

    ones = torch.ones(*alphas.shape[:-1], 1, device=rays_o.device, dtype=alphas.dtype)
    T = torch.cumprod(torch.cat([ones, 1.0 - alphas + 1e-10], dim=-1), dim=-1)[..., :-1]
    weights = alphas * T

    rgb_map = torch.sum(weights[..., None] * rgb, dim=-2)

    return rgb_map

print("Ray generation and rendering functions defined.")
```

```
Ray generation and rendering functions defined.
```

32.3.5. 2.5 NeRF Training Loop

Training Strategy: - **Random ray batching:** Pick random image, sample random pixels, render rays, compare with ground truth - **Optimizer:** Adam with learning rate 5e-4 - **Live Monitoring:** Every 400 iterations, render held-out test view and compute PSNR

You'll see the image start as pure noise and gradually sharpen into a photorealistic view.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = NeRF().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=5e-4)
n_iters = 10000
batch_size = 1024
display_rate = 400

train_psnrs = []
val_psnrs = []
iternums = []

if not isinstance(poses, torch.Tensor):
    print("Converting data to PyTorch Tensors...")
    poses = torch.from_numpy(poses).float().to(device)
    images = torch.from_numpy(images).float().to(device)
    focal = torch.tensor(focal, dtype=torch.float32).to(device)

print("--- Starting NeRF Training ---")
```

```
pbar = tqdm(range(n_iters), desc="Training NeRF", unit="iter")
for i in pbar:
    model.train()

    img_idx = np.random.randint(n_training)
    target_img = images[img_idx]
    target_pose = poses[img_idx]
    rays_o, rays_d = get_rays(height, width, focal, target_pose)

    coords = torch.stack(torch.meshgrid(torch.arange(height), torch.arange(width), indexing='ij'), -1).
  reshape(-1, 2)
    select_idx = np.random.choice(coords.shape[0], batch_size, replace=False)
    select_coords = coords[select_idx]

    rays_o_batch = rays_o[select_coords[:, 0], select_coords[:, 1]]
    rays_d_batch = rays_d[select_coords[:, 0], select_coords[:, 1]]
    target_px_batch = target_img[select_coords[:, 0], select_coords[:, 1]]

    pred_px_batch = render_rays(model, rays_o_batch, rays_d_batch, near, far, rand=True)
    loss = torch.mean((pred_px_batch - target_px_batch)**2)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    psnr = -10. * torch.log10(loss)
    train_psnrs.append(psnr.item())
    pbar.set_postfix({'loss': f'{loss.item():.4f}', 'psnr': f'{psnr.item():.2f}'})

    if (i + 1) % display_rate == 0 or i == 0:
        model.eval()
        with torch.no_grad():
            testpose_tensor = torch.from_numpy(testpose).float().to(device)
            rays_o_test, rays_d_test = get_rays(height, width, focal, testpose_tensor)

            # Flatten rays for chunked rendering
            rays_o_flat = rays_o_test.reshape(-1, 3)
            rays_d_flat = rays_d_test.reshape(-1, 3)

            chunks = []
            for j in range(0, rays_o_flat.shape[0], batch_size):
                chunks.append(render_rays(model, rays_o_flat[j:j+batch_size], rays_d_flat[j:j+batch_size],
  near, far))
            test_render = torch.cat(chunks, 0).reshape(height, width, 3)

            testimg_tensor = torch.from_numpy(testimg).float().to(device)
            val_loss = torch.mean((test_render - testimg_tensor)**2)
            val_psnr = -10. * torch.log10(val_loss)

            val_psnrs.append(val_psnr.item())
            iternums.append(i + 1)

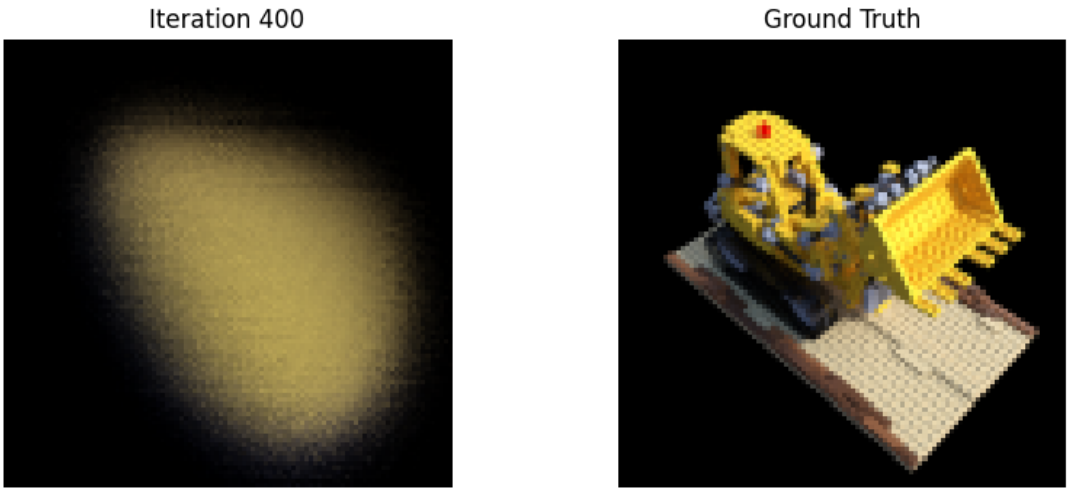
        plt.figure(figsize=(10, 4))
        plt.subplot(1, 2, 1); plt.imshow(test_render.cpu().numpy()); plt.title(f"Iteration {i+1}")
        plt.axis('off')
        plt.subplot(1, 2, 2); plt.imshow(testimg); plt.title("Ground Truth"); plt.axis('off')
        plt.show()
```

```
Converting data to PyTorch Tensors...
--- Starting NeRF Training ---
```

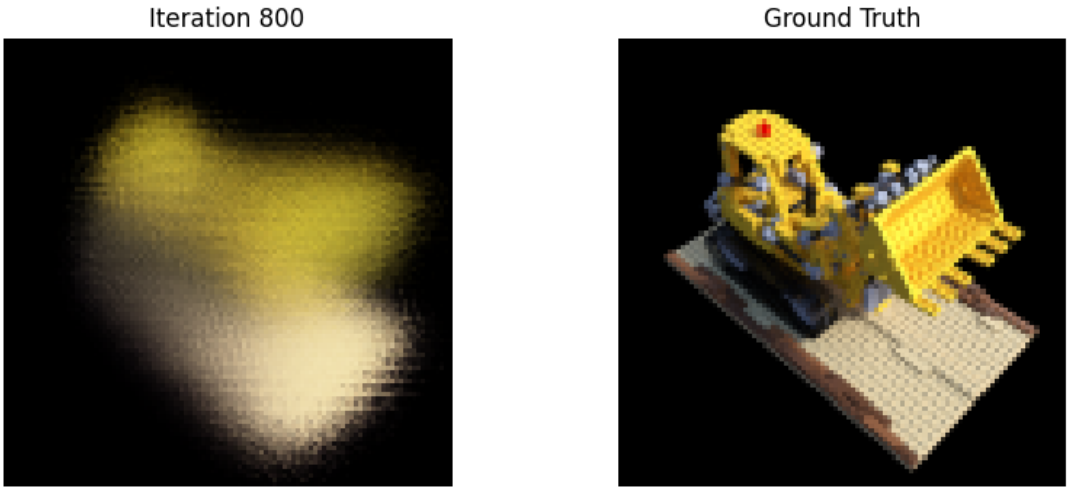
```
Training NeRF:   0%|          | 0/10000 [00:01<?, ?iter/s, loss=0.1876,
psnr=7.27]
```



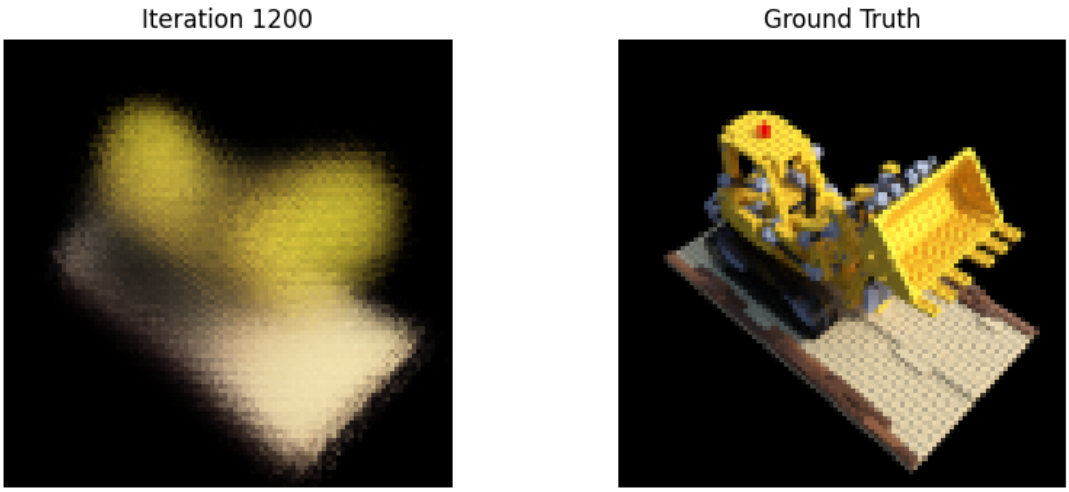

```
Training NeRF:   4%|#         | 399/10000 [00:34<13:13, 12.09iter/s,
loss=0.0235, psnr=16.29]
```



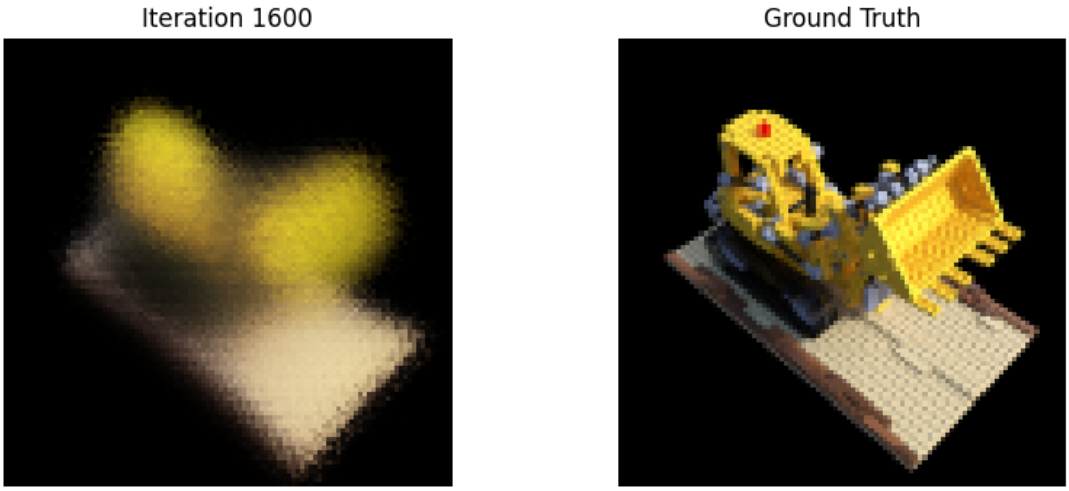

```
Training NeRF:   8%|#         | 799/10000 [01:09<13:14, 11.58iter/s,
loss=0.0182, psnr=17.39]
```



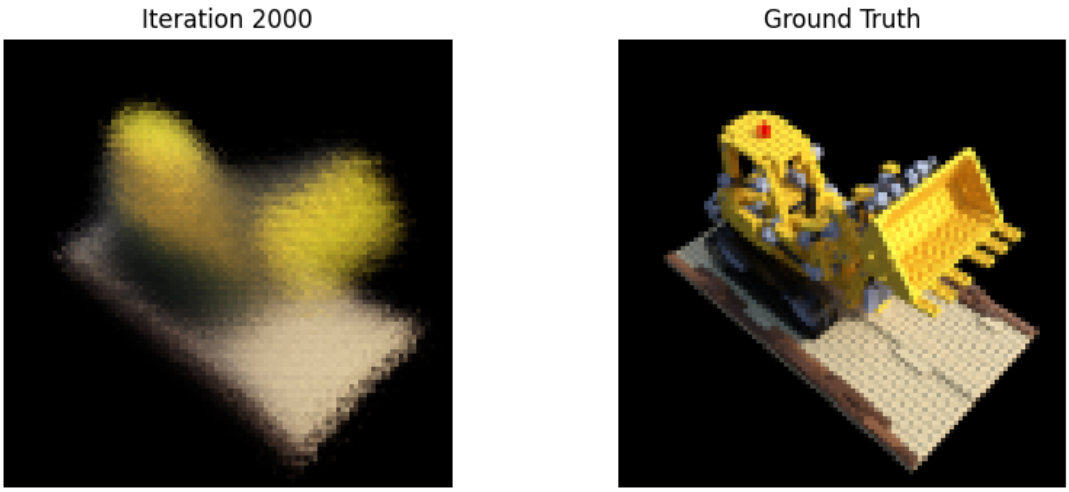

```
Training NeRF:  12%|##        | 1199/10000 [01:45<13:22, 10.96iter/s,
loss=0.0120, psnr=19.19]
```



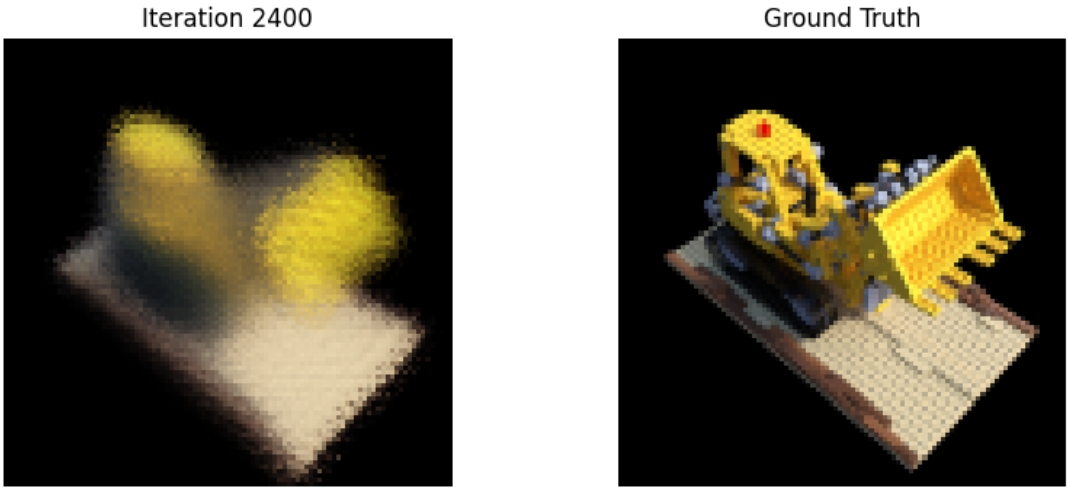

```
Training NeRF:  16%|##        | 1599/10000 [02:21<12:15, 11.43iter/s,
loss=0.0139, psnr=18.57]
```



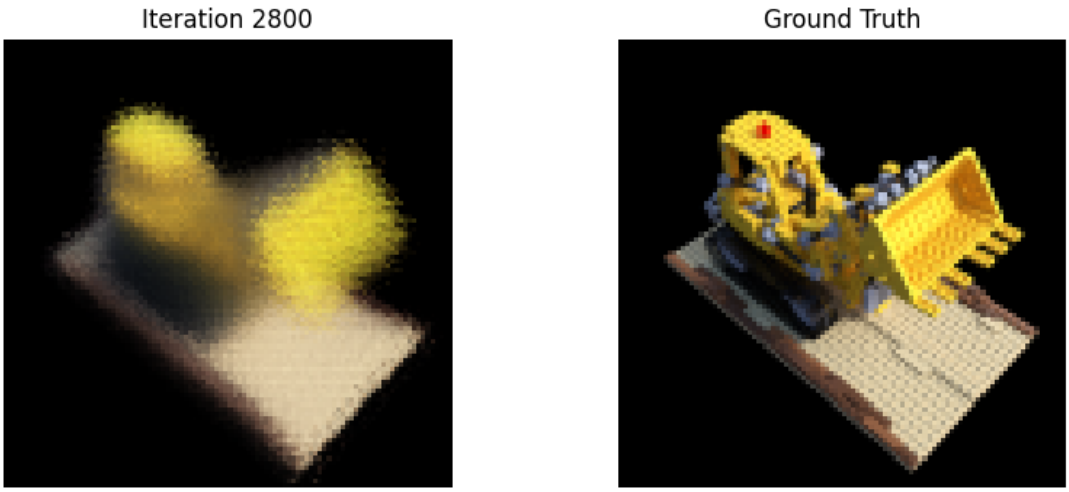

```
Training NeRF:  20%|##        | 1999/10000 [02:57<11:58, 11.14iter/s,
loss=0.0110, psnr=19.58]
```



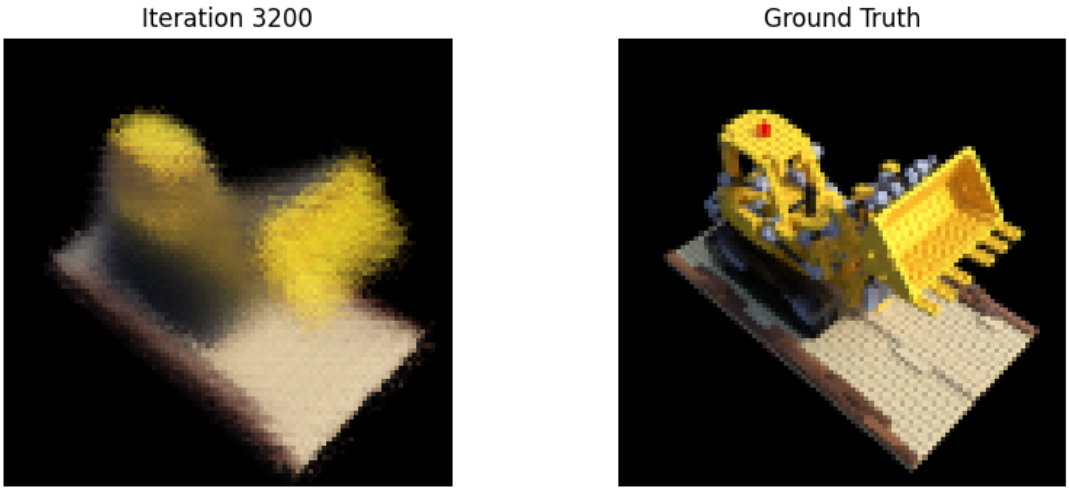

```
Training NeRF:  24%|###       | 2399/10000 [03:33<11:18, 11.21iter/s,
loss=0.0057, psnr=22.47]
```



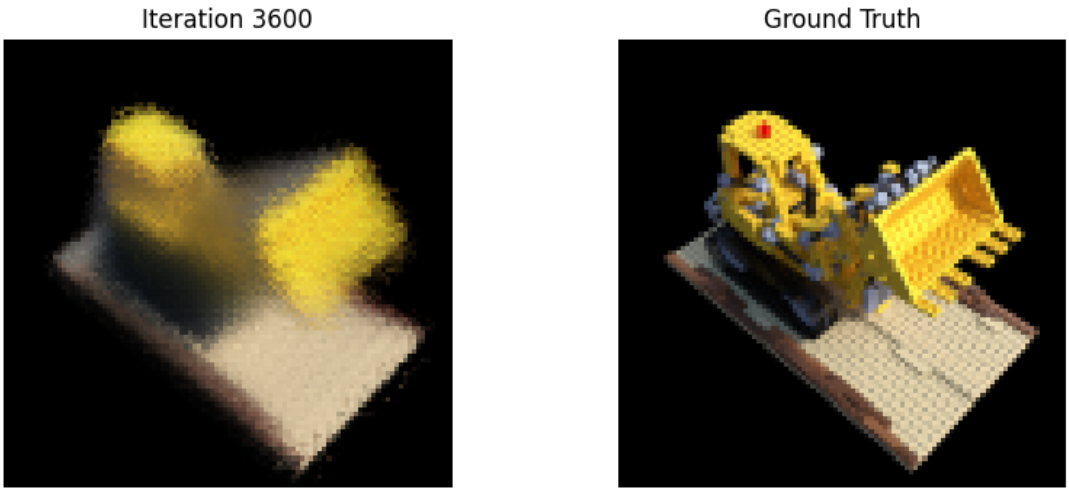

```
Training NeRF:  28%|###       | 2799/10000 [04:09<10:33, 11.37iter/s,
loss=0.0109, psnr=19.61]
```



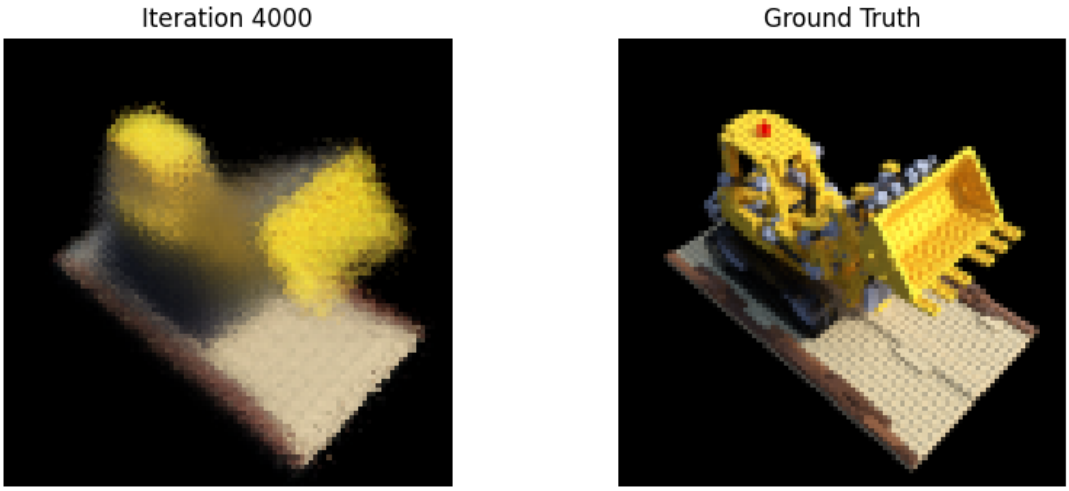

```
Training NeRF:  32%|####      | 3199/10000 [04:45<09:56, 11.40iter/s,
loss=0.0135, psnr=18.69]
```



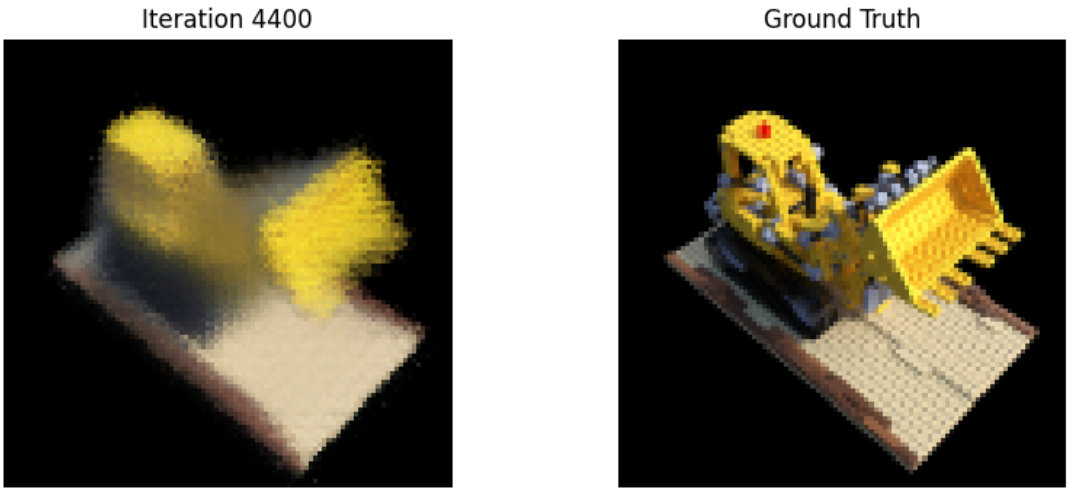

```
Training NeRF:  36%|####      | 3599/10000 [05:21<09:21, 11.39iter/s,
loss=0.0094, psnr=20.25]
```



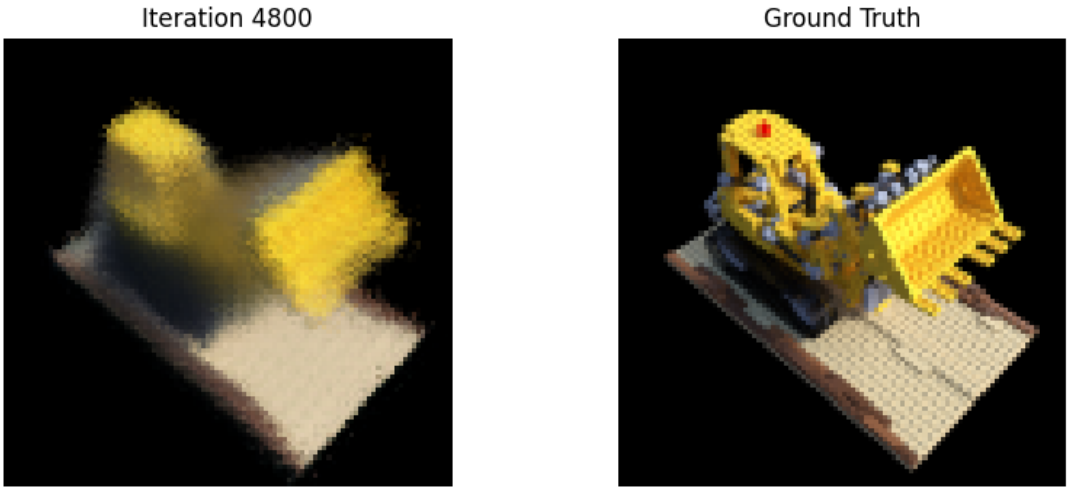

```
Training NeRF:  40%|####      | 3999/10000 [05:56<08:47, 11.38iter/s,
loss=0.0082, psnr=20.88]
```



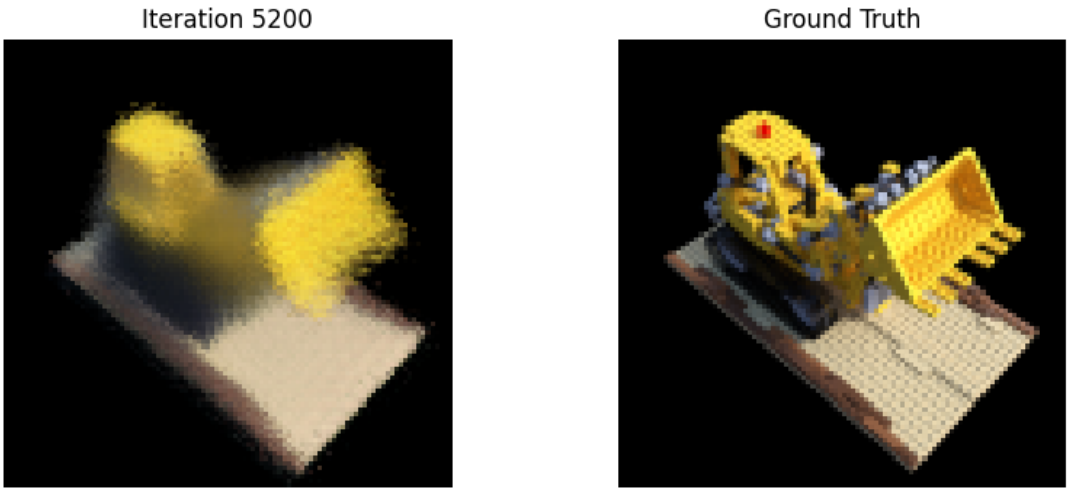

```
Training NeRF:  44%|#####     | 4399/10000 [06:32<08:11, 11.39iter/s,
loss=0.0084, psnr=20.76]
```



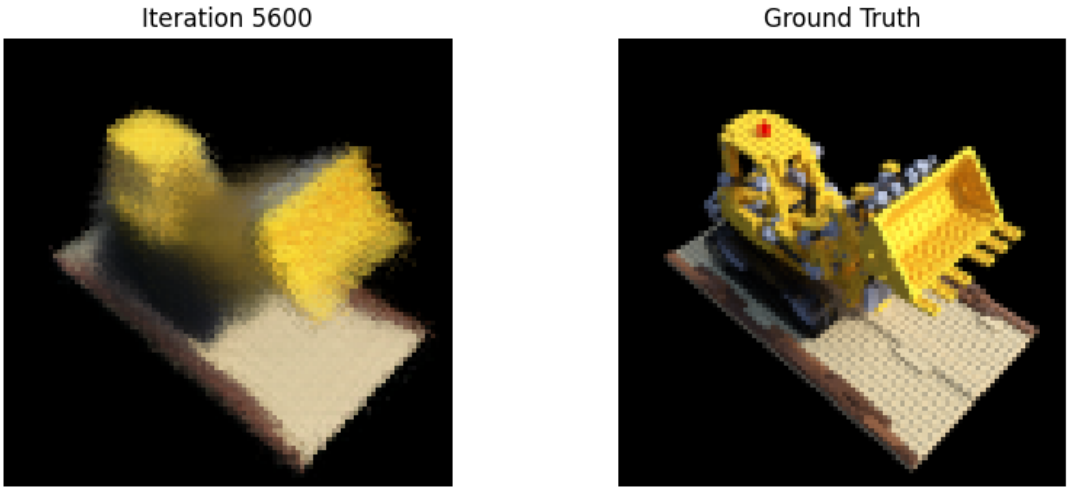

```
Training NeRF:  48%|#####     | 4799/10000 [07:07<07:36, 11.38iter/s,
loss=0.0049, psnr=23.07]
```



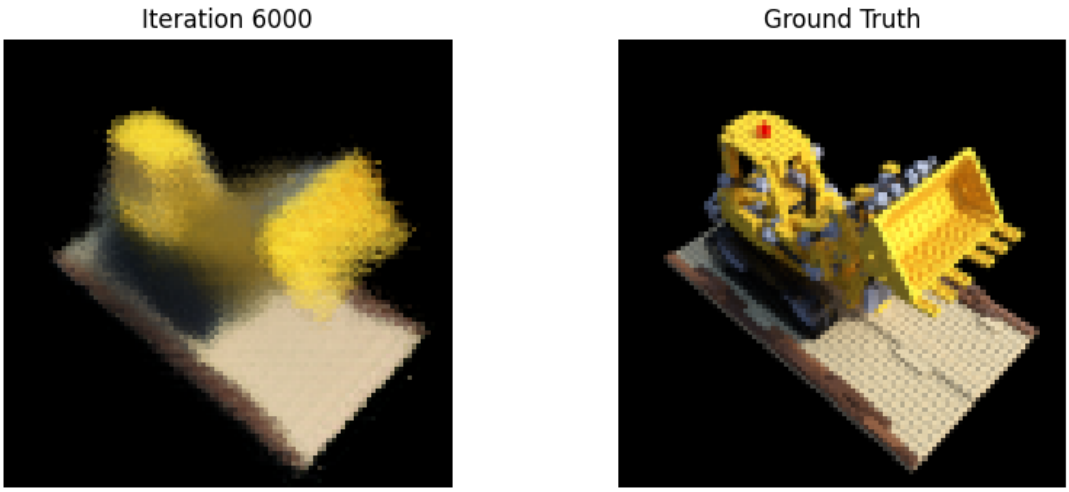

```
Training NeRF:  52%|######    | 5199/10000 [07:43<06:56, 11.52iter/s,
loss=0.0074, psnr=21.33]
```



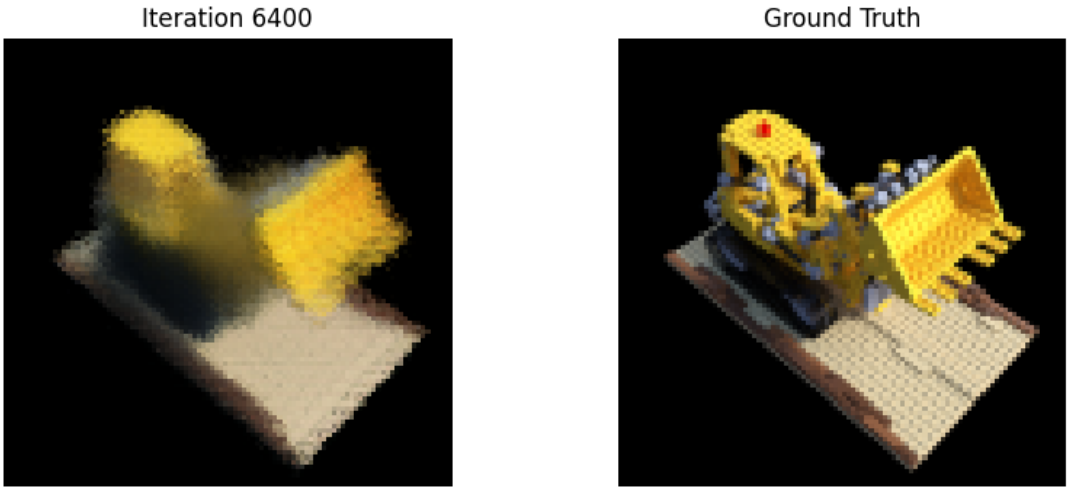

```
Training NeRF:  56%|######    | 5599/10000 [08:18<06:26, 11.39iter/s,
loss=0.0089, psnr=20.53]
```



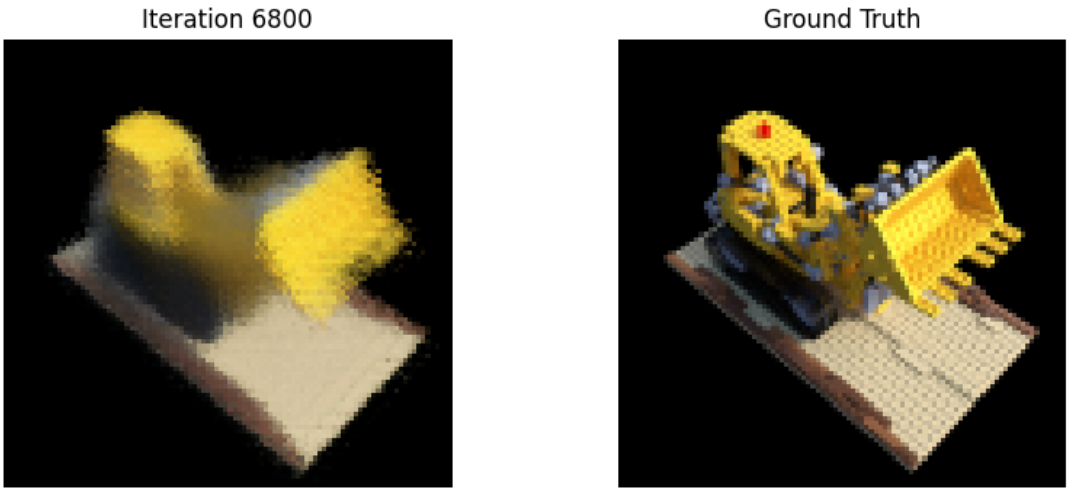

```
Training NeRF:  60%|######    | 5999/10000 [08:54<05:49, 11.46iter/s,
loss=0.0109, psnr=19.63]
```



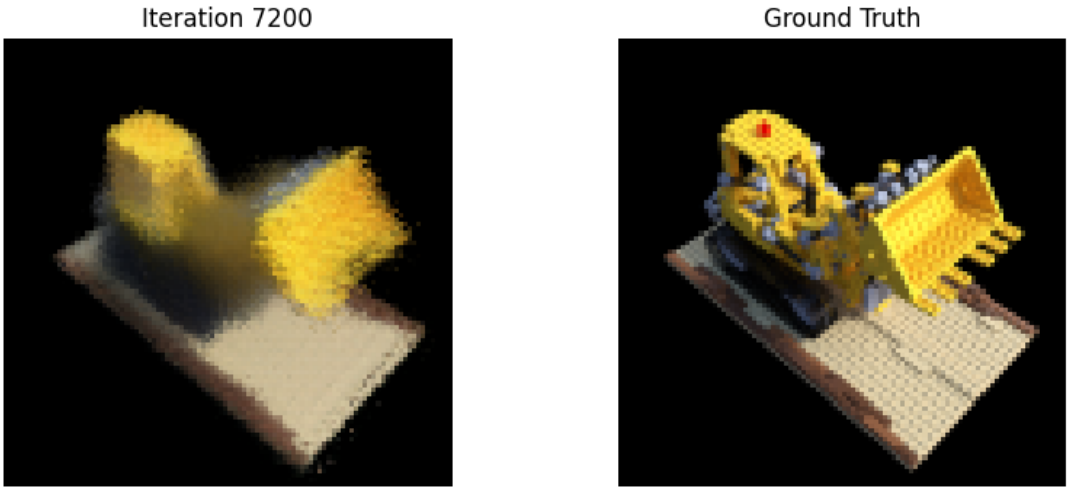

```
Training NeRF:  64%|#######   | 6399/10000 [09:29<05:16, 11.39iter/s,
loss=0.0025, psnr=25.97]
```



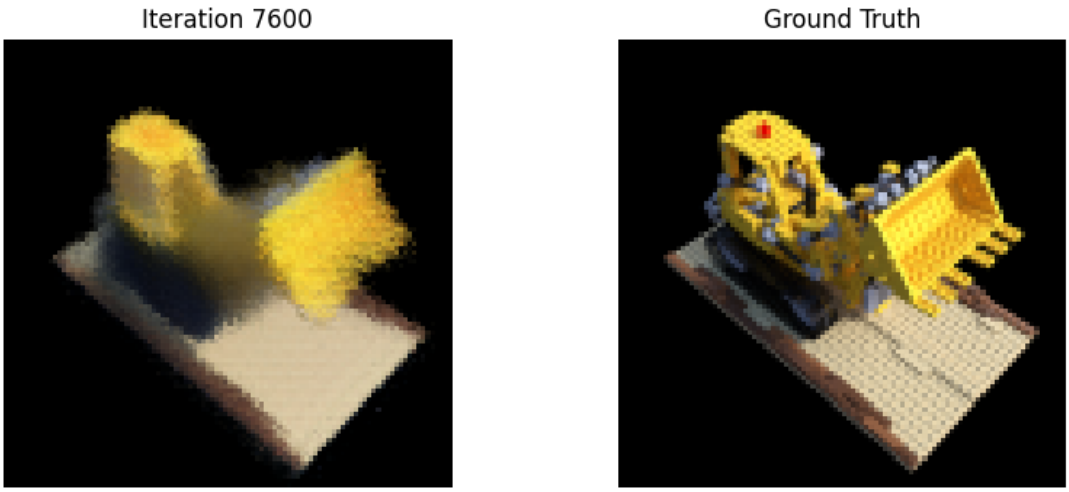

```
Training NeRF:  68%|#######   | 6799/10000 [10:05<04:37, 11.55iter/s,
loss=0.0085, psnr=20.71]
```



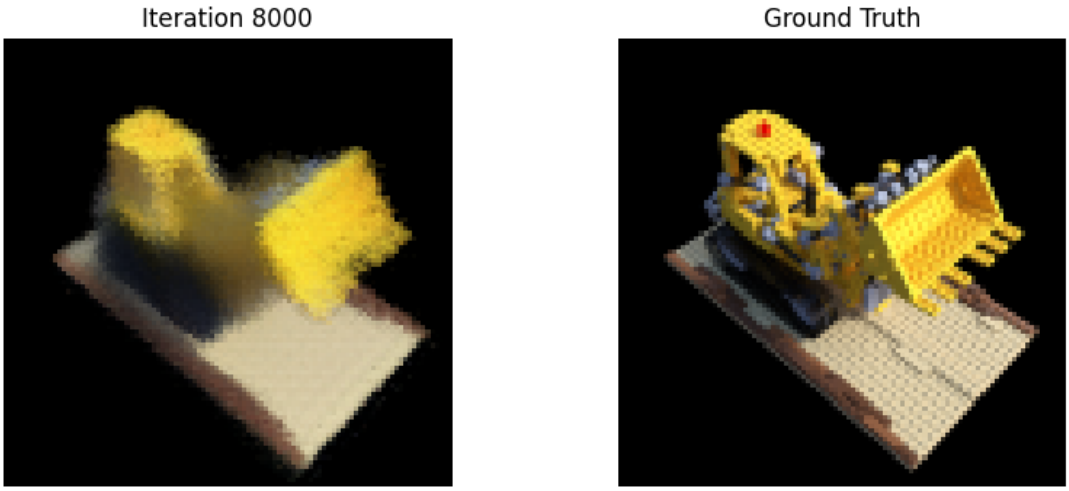

```
Training NeRF:  72%|########  | 7199/10000 [10:40<04:04, 11.44iter/s,
loss=0.0093, psnr=20.33]
```



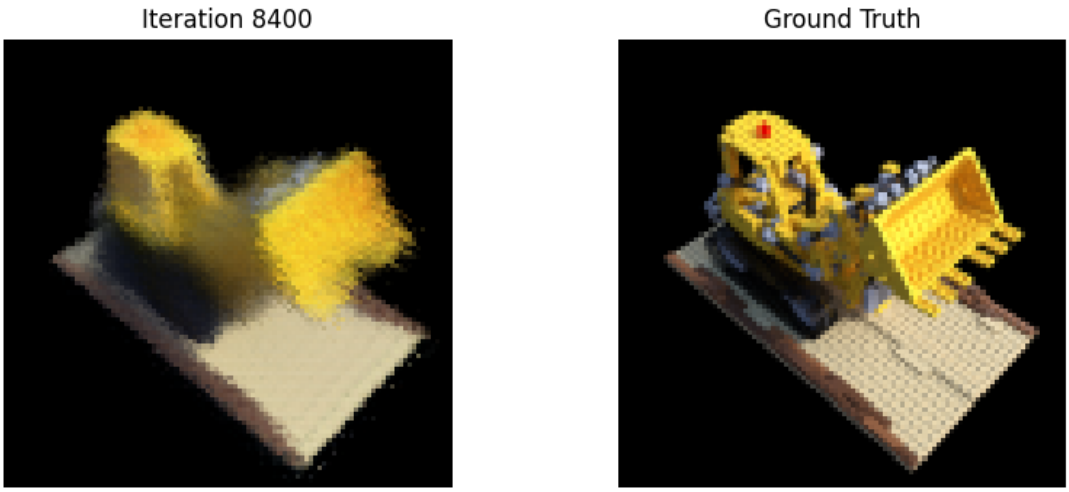

```
Training NeRF:  76%|########  | 7599/10000 [11:15<03:27, 11.56iter/s,
loss=0.0075, psnr=21.26]
```



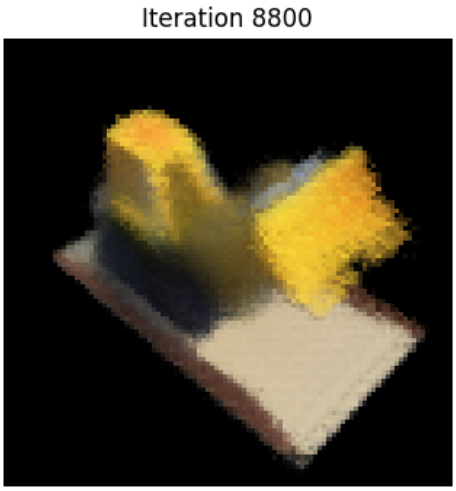

```
Training NeRF:  80%|########  | 7999/10000 [11:51<02:54, 11.49iter/s,
loss=0.0025, psnr=25.95]
```




```
Training NeRF:  84%|######### | 8399/10000 [12:26<02:19, 11.50iter/s,
loss=0.0044, psnr=23.56]
```

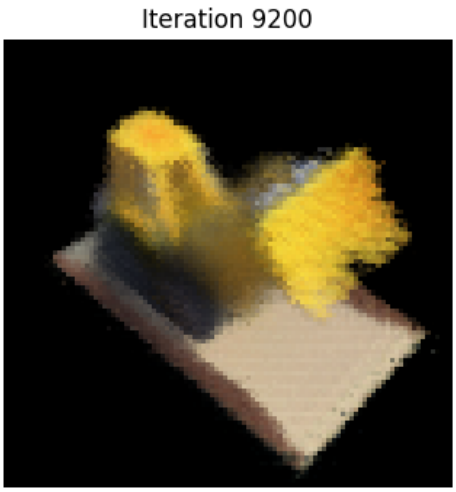



```
Training NeRF:  88%|######### | 8799/10000 [13:02<01:44, 11.52iter/s,
loss=0.0062, psnr=22.07]
```



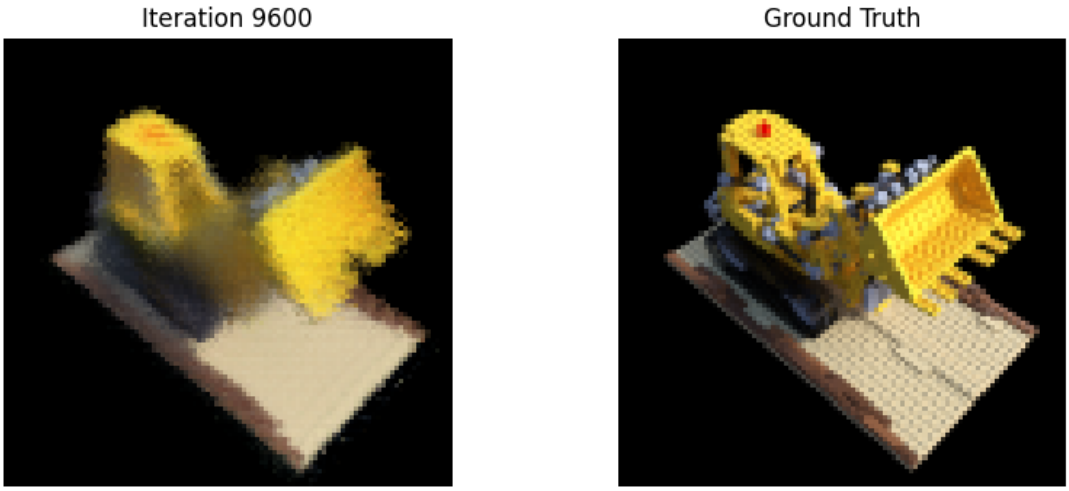



```
Training NeRF:  92%|##########| 9199/10000 [13:37<01:10, 11.43iter/s,
loss=0.0039, psnr=24.05]
```

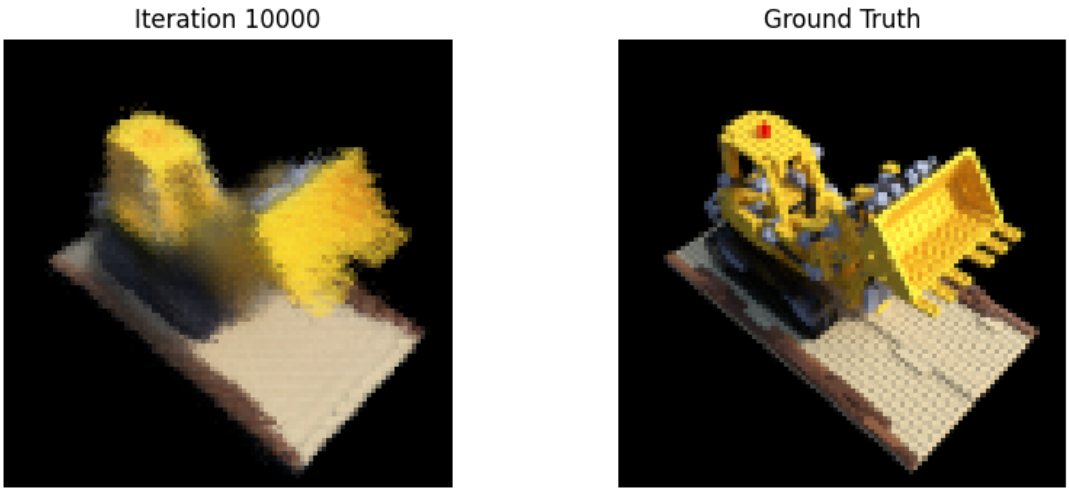





```
Training NeRF:  96%|##########| 9599/10000 [14:12<00:35, 11.35iter/s,
loss=0.0042, psnr=23.75]
```




```
Training NeRF: 100%|##########| 9999/10000 [14:48<00:00, 11.44iter/s,
loss=0.0042, psnr=23.72]
```




```
Training NeRF: 100%|##########| 10000/10000 [14:48<00:00, 11.25iter/s,
loss=0.0042, psnr=23.72]
```

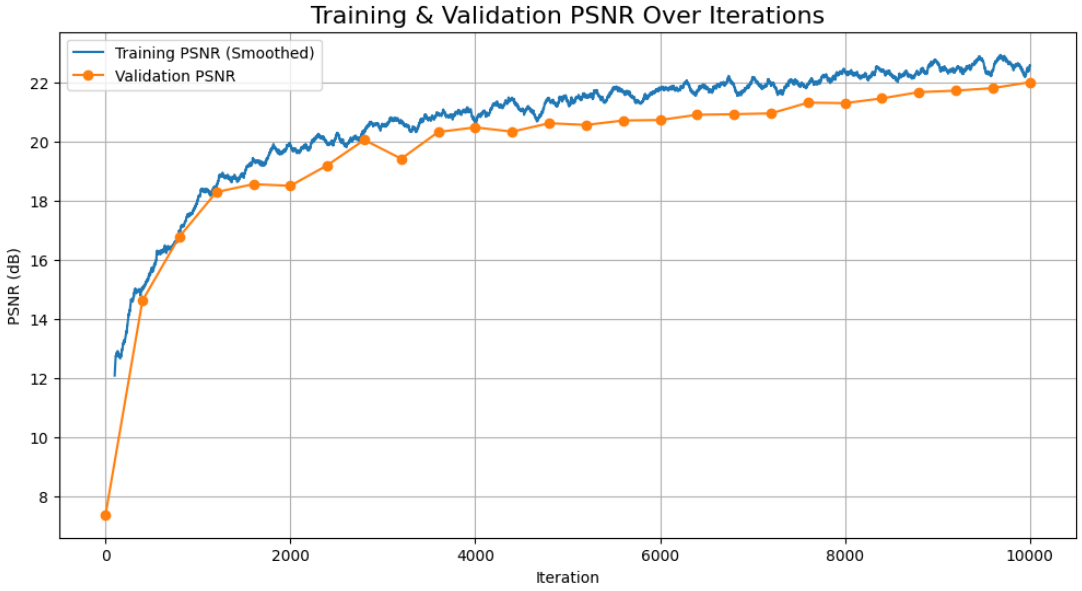
32.3.6. 2.6 Training Plots and Final Visualization

```python
def moving_average(a, n=100):
    ret = np.cumsum(a, dtype=float)
    ret[n:] = ret[n:] - ret[:-n]
    return ret[n - 1:] / n

smoothed_train_psnrs = moving_average(train_psnrs)

plt.figure(figsize=(12, 6))
plt.plot(np.arange(len(smoothed_train_psnrs)) + 100, smoothed_train_psnrs, label='Training PSNR (Smoothed)')
plt.plot(iternums, val_psnrs, 'o-', label='Validation PSNR')
plt.title('Training & Validation PSNR Over Iterations', fontsize=16)
```

```
plt.xlabel('Iteration')
plt.ylabel('PSNR (dB)')
plt.legend()
plt.grid(True)
plt.show()
```




```
print("--- Rendering novel view fly-through ---")

from IPython.display import Image as IPImage # Added this import

def pose_spherical_look_at(angle, phi, radius):
    angle_rad = np.deg2rad(angle)
    phi_rad = np.deg2rad(phi)
    camera_pos = torch.tensor([
        radius * np.sin(phi_rad) * np.cos(angle_rad),
        radius * np.sin(phi_rad) * np.sin(angle_rad),
        radius * np.cos(phi_rad)
    ], dtype=torch.float32)
    target = torch.tensor([0., 0., 0.])
    forward_vec = (target - camera_pos).float()
    forward_vec /= torch.norm(forward_vec)
    temp_up = torch.tensor([0., 0., 1.]).float()
    right_vec = torch.cross(forward_vec, temp_up)
    right_vec /= torch.norm(right_vec)
    up_vec = torch.cross(right_vec, forward_vec)
    c2w = torch.eye(4)
    c2w[:3, 0] = right_vec
    c2w[:3, 1] = up_vec
    c2w[:3, 2] = -forward_vec
    c2w[:3, 3] = camera_pos
    return c2w

render_poses = torch.stack([pose_spherical_look_at(angle, -30.0, 4.0) for angle in np.linspace(0, 360, 40,
   endpoint=False)], 0)

frames = []
model.eval()
```

```
with torch.no_grad():
    for pose in tqdm(render_poses, desc="Rendering frames"):
        pose = pose.to(device)
        rays_o, rays_d = get_rays(height, width, focal, pose)

        # Flatten rays for chunked rendering
        rays_o_flat = rays_o.reshape(-1, 3)
        rays_d_flat = rays_d.reshape(-1, 3)

        chunks = []
        for j in range(0, rays_o_flat.shape[0], batch_size):
            chunks.append(render_rays(model, rays_o_flat[j:j+batch_size], rays_d_flat[j:j+batch_size], near,
  far))

        rendered_img = torch.cat(chunks, 0).reshape(height, width, 3).cpu().numpy()
        frame = (255 * np.clip(rendered_img, 0, 1)).astype(np.uint8)
        frames.append(frame)

gif_path = 'tiny_nerf_flythrough.gif'
imageio.mimsave(gif_path, frames, fps=15, loop=0)

print(f"\nGIF created and saved to '{gif_path}'.")
IPImage(filename=gif_path)
```

```
--- Rendering novel view fly-through ---
```

```
Rendering frames: 100%|##########| 40/40 [00:14<00:00,  2.81it/s]
```

```
GIF created and saved to 'tiny_nerf_flythrough.gif'.
```

32.4. 3. Gaussian Splatting

32.4.1. 3.1 Introduction to 3D Gaussian Splatting

3D Gaussian Splatting (Kerbl et al., SIGGRAPH 2023) is the state-of-the-art for photorealistic novel-view synthesis, dramatically faster than NeRF.

Core Idea: Represent the scene as **millions of anisotropic 3D Gaussians**, each with: - Position (x, y, z) - Covariance (scale + rotation) - Color (spherical harmonics) - Opacity α

Rendering = Differentiable Rasterization: - Project Gaussians onto image plane - Sort by depth and alpha-blend using fast tile-based rasterizer - **100–1000× faster** than NeRF's ray marching

Feature	NeRF	3D Gaussian Splatting
Training speed	Hours	10–30 minutes
Rendering speed	Seconds/frame	Real-time (100+ FPS)
Memory	Small MLP	~100 MB Gaussians

32.4.2. 3.2 Setup for Gaussian Splatting

```
print("--- Downloading a pre-trained Gaussian Splatting scene ---")

ply_filename = "truck.ply"
```

```
if not os.path.exists(ply_filename):
    print(f"Downloading '{ply_filename}'...")
    !curl -L -o {ply_filename} "https://huggingface.co/cakewalk/splat-data/resolve/main/gs_FF3_lumix_4k%203.
  ply"
    print(f"\nDownload complete. File saved as '{ply_filename}'.")
else:
    print(f"Scene file '{ply_filename}' already exists.")

file_size_mb = os.path.getsize(ply_filename) / (1024 * 1024)
print(f"   File size: {file_size_mb:.2f} MB")
```

```
--- Downloading a pre-trained Gaussian Splatting scene ---
Downloading 'truck.ply'...
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  1369  100  1369    0     0   7072      0 --:--:-- --:--:-- --:--:--  7093
100  250M  100  250M    0     0  77.3M      0  0:00:03  0:00:03 --:--:-- 88.6M

Download complete. File saved as 'truck.ply'.
   File size: 250.85 MB
```

32.4.3. 3.3 Interactive Web Viewer

```
print("--- Launching Interactive Gaussian Splatting Viewer ---")
print("\nControls:")
print("- LEFT mouse: Rotate camera")
print("- RIGHT mouse: Pan camera")
print("- SCROLL wheel: Zoom")

viewer_url = "https://antimatter15.com/splat/"
scene_url = "https://huggingface.co/cakewalk/splat-data/resolve/main/gs_FF3_lumix_4k%203.ply"

IFrame(src=f"{viewer_url}?url={scene_url}", width="100%", height="600px")
```

```
--- Launching Interactive Gaussian Splatting Viewer ---

Controls:
- LEFT mouse: Rotate camera
- RIGHT mouse: Pan camera
- SCROLL wheel: Zoom
```

32.4.4. 3.4 Analysis: Real-Time vs. Offline Rendering

What You Just Saw: A complete, photorealistic 3D scene rendered in **real-time** directly in your browser at 30-60 FPS.

NeRF (Slow, Offline): For every pixel, must cast a ray and run hundreds of MLP queries. Creating our 40-frame GIF took several minutes.

Gaussian Splatting (Fast, Real-Time): Represents the scene as millions of 3D Gaussians that are projected ("splatted") onto the screen. This rasterization process is massively parallel and runs extremely fast on GPUs.

32.4.5. HOMEWORK (Optional)

Exercise 1: Bundle Adjustment Extend the SfM pipeline by implementing bundle adjustment to jointly optimize camera poses and 3D point positions. Use `scipy.optimize.least_squares` with the Levenberg-Marquardt algorithm to minimize reprojection error across all views.

Exercise 2: NeRF Improvements Experiment with the NeRF architecture: - Try different positional encoding frequencies (e.g., 6 vs 10) - Implement hierarchical sampling with coarse and fine networks - Add a depth regularization loss term

Exercise 3: Your Own Dataset Capture your own scene (30-50 photos walking around an object) and run through the complete pipeline: 1. Run COLMAP or our SfM pipeline to get camera poses 2. Train NeRF on your images 3. Export the point cloud and try training a Gaussian Splatting model using the official implementation

Part II.

Appendices

Chapter 33

Datasets and Pretrained Models

This appendix lists the datasets and pretrained models used across the practical chapters of the memorandum, together with their original sources. Reference numbers in square brackets refer to the consolidated bibliography in Appendix B.

Datasets

Dataset	Used in chapter(s)	Original source
CIFAR-10	Image Classification with CNNs; The Vision Transformer (ViT); Superpixel Tokenization for ViTs	Krizhevsky, 2009 [1]
MNIST	Generative Modelling	LeCun et al., 1998 [2]
Fashion-MNIST	Generative Modelling	Xiao et al., 2017 [3]
GTZAN (Music Genre)	Spectrogram Classification with CNNs	Tzanetakis & Cook, 2002 [4]
IMDB Movie Reviews	Sentiment Analysis	Maas et al., 2011 [5]
Hymenoptera (ants/bees)	Transfer Learning; Quantization Techniques	Chilamkurthy, n.d. [9], a small subset of ImageNet (Deng et al., 2009 [31])
PASCAL VOC 2012	Faster R-CNN for Object Detection; Object Detection from Scratch; Monocular Depth Estimation; Optical Flow; Text-Guided Image Inpainting	Everingham et al., 2010 [6]
Penn-Fudan Pedestrian	Faster R-CNN for Object Detection	Wang et al., 2007 [7]
LGG MRI Segmentation / TCGA-LGG	Image Segmentation with U-Net	Buda, Saha & Mazurowski, 2019 [32], imagery derived from the TCGA-LGG collection (Pedano et al., 2016 [33])

COCO val2017 (samples)	Faster R-CNN for Object Detection; Instance Segmentation with YOLO11; Object Detection with YOLO11; Pose Estimation with YOLO11; Segment Anything Model 2; Hugging Face Transformers and Gradio	Lin et al., 2014 [8]
WHO Life Expectancy	Linear Regression	WHO Global Health Observatory [10]
AirPassengers	Time Series Prediction	Box & Jenkins, 1976 [11]
Tatoeba sentence and link exports	Neural Machine Translation; NMT with Transformers	Tatoeba Association, n.d. [34]
HuggingFaceH4/MATH-500	Hugging Face Transformers and Gradio	HuggingFaceH4, n.d. [35], derived from Lightman et al., 2023 [36]
Middlebury Temple Ring	Neural Rendering	Seitz et al., 2006 [37]
Tiny NeRF / NeRF synthetic LEGO scene	Neural Rendering	Mildenhall et al., 2020 [27]; underlying 3D model by Heinzelnisse — see note below
SA-V	Segment Anything Model 2 (background training dataset, discussed in §18.1; see note below)	Ravi et al., 2024 [12]

Note. Flickr8k is not used in this Practice Memorandum. It belongs to the companion *Assignments and Coding Exams* memorandum and has been removed from this table accordingly. SA-V is SAM 2's training dataset and is discussed for background only; the demonstration image and video files actually downloaded and processed in Chapter 18 (`truck.jpg`, `groceries.jpg`, `bedroom.zip`) are not identified by their source as SA-V samples — see "Third-party demonstration images and media" below for their individual sources.

Note. The Chapter 32 NeRF demonstration uses `tiny_nerf_data.npz` from the original NeRF project and reproduces rendered views of the standard synthetic LEGO scene associated with NeRF [27]. The scene is based on the *Lego 856 Bulldozer* Blender model by Heinzelnisse, distributed on BlendSwap under the CC-BY-NC licence (no specific version number is stated on the source page, so none is asserted here).
Dataset source: http://cseweb.ucsd.edu/~viscomp/projects/LF/papers/ECCV20/nerf/tiny_nerf_data.npz
NeRF project: https://github.com/bmild/nerf
Original model: *Lego 856 Bulldozer* — Heinzelnisse, via BlendSwap: https://blendswap.com/blend/11490

Licence: CC-BY-NC, as stated on the BlendSwap listing.

Pretrained Models

Table 33.2 lists checkpoints that are actually downloaded and loaded (via `from_pretrained()`, `torch.hub.load()`, torchvision `weights=` arguments, or Ultralytics/Meta checkpoint downloads) somewhere in this memorandum's executed notebook code.

Model	Used in chapter(s)	Original source
ResNet-18 / ResNet-50	Transfer Learning; Quantization Techniques; Faster R-CNN for Object Detection	He et al., 2016 [13]
Faster R-CNN (torchvision, COCO-pretrained)	Faster R-CNN for Object Detection	Ren et al., 2015 [16]; COCO-pretrained weights, Lin et al., 2014 [8]
YOLO11 (Ultralytics)	Object Detection with YOLO11; Instance Segmentation with YOLO11; Pose Estimation with YOLO11	Jocher & Qiu, 2024 [17]
Segment Anything 2 (SAM 2)	Segment Anything Model 2	Ravi et al., 2024 [12]
Helsinki-NLP Opus-MT (opus-mt-en-fr, opus-mt-en-de)	Hugging Face Transformers and Gradio	Tiedemann & Thottingal, 2020 [22]
T5 (default checkpoint of the `translation_en_to_de` pipeline)	Hugging Face Transformers and Gradio	Raffel et al., 2020 [20]
Stable Diffusion v1-5 (runwayml/stable-diffusion-v1-5)	Hugging Face Transformers and Gradio	Rombach et al., 2022 [23]
Stable Diffusion x2 latent upscaler (stabilityai/sd-x2-latent-upscaler)	Hugging Face Transformers and Gradio	Rombach et al., 2022 [23]; Stability AI checkpoint
ControlNet QR-code checkpoint (DionTimmer/controlnet_qrcode-control_v1p_sd15)	Hugging Face Transformers and Gradio	Community checkpoint; base method Zhang et al., 2023 [38]
BiRefNet (ZhengPeng7/BiRefNet)	Hugging Face Transformers and Gradio	Zheng et al., 2024 [24]

MiDaS / DPT-Large	Monocular Depth Estimation	Ranftl et al., 2021 [25]
RAFT (torchvision, C+T+SKHT-pretrained)	Optical Flow	Teed & Deng, 2020 [26]
Stable Diffusion v1-4 (CompVis/stable-diffusion-v1-4)	Text-Guided Image Inpainting	Rombach et al., 2022 [23]

Note. Exact licences for each of these checkpoints are given under "Software and model licensing notes" below, with a source/licence URL for each.

Architectures Implemented in the Notebooks

Table 33.3 lists architectures that this memorandum implements and trains (or fine-tunes) itself rather than loading as a ready-made pretrained checkpoint. The cited work is the architecture or method reference, not a downloaded set of weights.

Architecture	**Used in chapter(s)**	**Reference**
U-Net	Image Segmentation with U-Net	Ronneberger et al., 2015 [15]
Vision Transformer (ViT)	The Vision Transformer (ViT)	Dosovitskiy et al., 2021 [21]
Custom diffusion model (DDPM/DDIM)	Generative Modelling	Follows the diffusion formulation discussed by Rombach et al., 2022 [23]; not a loaded Stable Diffusion checkpoint
Neural Radiance Fields (NeRF)	Neural Rendering	Mildenhall et al., 2020 [27]
3D Gaussian Splatting	Neural Rendering	Kerbl et al., 2023 [28]; see note below

Note. The Gaussian Splatting section of Chapter 32 downloads and visualizes a precomputed splat scene rather than training one with the official reference implementation; see "Third-party demonstration images and media" below. An optional end-of-chapter exercise invites the reader to train a scene with the official implementation, but this is not executed in the memorandum itself.

Note. VGG-16, DistilBERT, RoBERTa, and GPT-2 do not appear anywhere in this memorandum's executed code and have been removed from this appendix. BERT is discussed as a comparison architecture in Chapters 7 and 25 (e.g., the [CLS]-token pooling analogy) but is not instantiated as a pretrained checkpoint here. An optional homework exercise in Chapter 26 (p. 555) suggests loading `google/vit-base-patch16-224`, but this is a reader exercise, not executed notebook content, so it is not listed as an instantiated pretrained model above.

Note. Datasets fetched at run-time via standard library helpers (`torchvision.datasets`,

`datasets` from Hugging Face, Ultralytics weight downloads, `torch.hub.load()`) are downloaded from each project's canonical hosting and are governed by their respective licences. Hugging Face models are loaded via `from_pretrained()` from each model's official Hugging Face repository. Some redistributions used for course convenience (e.g., the AirPassengers CSV via the `jbrownlee/Datasets` GitHub mirror, p. 374; the GTZAN archive via an OSF mirror, p. 143) are technical hosting conveniences and do not change the underlying dataset's citation or terms.

Licensing and attribution notes

This section states what governs re-use of the material in this memorandum. Three principles apply throughout:

1. Third-party datasets and pretrained model files are generally retrieved at run time by notebook code (via `wget`/`curl`, `torchvision.datasets`, the Hugging Face `datasets`/`transformers` libraries, Ultralytics weight downloads, or `torch.hub`) and are not redistributed as complete packages with this memorandum.

2. Sample images, audio-derived representations, dataset visualizations, segmentation masks, depth maps, optical-flow visualizations, splat scenes, and other model outputs shown in this volume retain the provenance of the underlying source data. Transforming an input (for example, running it through a depth, segmentation, or diffusion model) does not remove or replace the copyright or licensing status of that input; the transformed output remains subject to the terms of the original material.

3. Where a dataset's licence permits research/educational use but does not grant a blanket republication licence for every underlying photograph or audio file, that is stated explicitly below rather than assumed. Where a licence is named below, an authoritative source or licence URL is given alongside it; a software, model, or viewer licence is never treated as covering the separate dataset, image, or scene it was used to process.

Dataset-by-dataset notes

CIFAR-10. Distributed by Krizhevsky [1] for research use. No separate blanket republication licence for the underlying images is asserted here; the terms of the original dataset apply.

MNIST. Distributed by LeCun et al. [2]. No separate blanket republication licence for the underlying images is asserted here; the terms of the original dataset apply.

Fashion-MNIST. Distributed by Zalando Research (Xiao et al. [3]) under the MIT License for the dataset repository; the licence covers the released data files as distributed by that repository. Licence: MIT License — https://github.com/zalandoresearch/fashion-mnist/blob/master/LICENSE

GTZAN. Distributed for research use under its own terms [4]. No separate blanket republication licence for the underlying audio is asserted here; the terms of the original dataset or source material apply.

Hymenoptera / ImageNet subset. PyTorch's transfer-learning tutorial describes this dataset as "a very small subset of imagenet" [9]. The tutorial's own code is BSD-3-Clause-licensed as part of the `pytorch/tutorials` repository, but that software licence covers the tutorial code, not the

underlying ant and bee photographs. The photographs derive from ImageNet (Deng et al. [31]) and remain subject to ImageNet's own terms of access and use; no separate blanket republication licence for the photographs is asserted here.
Tutorial code licence: BSD-3-Clause — https://github.com/pytorch/tutorials/blob/main/LICENSE

Penn-Fudan Pedestrian. Distributed by the original authors [7] for research use. No separate blanket republication licence for the underlying photographs is asserted here; the terms of the original dataset or source material apply.

PASCAL VOC 2012. The VOC challenge dataset [6] contains third-party photographs contributed under the original Flickr/VOC collection terms. The original image-source terms remain relevant to each photograph; no separate blanket republication licence for the underlying images is asserted here.

COCO. The COCO annotations are released under a CC BY 4.0 licence, but the underlying COCO photographs are not necessarily governed by the same rights: each photograph was sourced from Flickr and carries its own, individual Flickr licence, which is not automatically CC BY 4.0 merely because the annotation licence is. The COCO organizers have confirmed that photographer/creator attribution is not embedded in the annotation files, so per-image provenance must be traced independently.
Dataset source: https://cocodataset.org/
Annotation licence: CC BY 4.0 — https://creativecommons.org/licenses/by/4.0/

LGG MRI / TCGA-LGG. Chapter 15 trains on the LGG MRI Segmentation dataset (Buda, Saha & Mazurowski, 2019 [32]), derived from the TCGA-LGG collection of The Cancer Imaging Archive (Pedano et al., 2016 [33]). It is not redistributed with this volume; the imagery is downloaded by notebook code from its Kaggle mirror and remains subject to that collection's terms and attribution requirements.

Tatoeba. The sentence and link exports [34] are downloaded directly from Tatoeba's export server. Tatoeba's terms of use apply to redistribution and reuse of the sentence data; no separate blanket republication licence beyond those terms is asserted here.
Terms of use: https://tatoeba.org/en/terms_of_use

MATH-500. Downloaded via the Hugging Face `datasets` library from HuggingFaceH4/MATH-500 [35], a curated subset derived from the evaluation methodology of Lightman et al. [36]. The dataset card's stated terms govern reuse; no separate blanket republication licence is asserted here.

Middlebury Temple Ring. Downloaded directly from the Middlebury multi-view stereo evaluation site [37] for the structure-from-motion demonstration in Chapter 32. The data is distributed for research/benchmark use under Middlebury's own terms; no separate blanket republication licence is asserted here.

Tiny NeRF / NeRF synthetic LEGO scene. See the note under “Datasets” above for the full Heinzelnisse/BlendSwap CC-BY-NC provenance of the underlying 3D model.

Software and model licensing notes

- **Ultralytics YOLO11.** The Ultralytics software and model resources used in Chapters 12, 16, and 17 are offered under AGPL-3.0, with separate commercial/Enterprise licensing options available from Ultralytics for users who cannot satisfy AGPL-3.0's terms. This software

licence governs the Ultralytics code and the released model weights; it does not licence third-party input images, third-party datasets, or other material merely because Ultralytics software was used to process them.
Project source: https://github.com/ultralytics/ultralytics
Licence: GNU Affero General Public License v3.0 — https://www.gnu.org/licenses/agpl-3.0.html

- **SAM 2.** The SAM 2 code and released model checkpoints (Ravi et al. [12]) are distributed by Meta under the Apache License 2.0.
  Project source: https://github.com/facebookresearch/sam2
  Licence: Apache License 2.0 — https://www.apache.org/licenses/LICENSE-2.0
  This is separate from the SA-V dataset used for SAM 2's training, which has its own dataset terms; using the Apache-licensed code/checkpoints does not by itself grant any rights to SA-V or vice versa.
  SA-V dataset source: https://ai.meta.com/datasets/segment-anything-video/
  SA-V licence: the dataset page identifies SA-V as CC BY 4.0 — https://creativecommons.org/licenses/by/4.0/

- **BiRefNet.** The current BiRefNet implementation (Zheng et al. [24]) is distributed under the MIT License. BiRefNet is not non-commercial-only software.
  Project source: https://github.com/ZhengPeng7/BiRefNet
  Licence: MIT License — https://github.com/ZhengPeng7/BiRefNet/blob/main/LICENSE

- **Stable Diffusion v1-4 (CompVis/stable-diffusion-v1-4).** Used in Chapter 30 (Text-Guided Image Inpainting).
  Source: https://huggingface.co/CompVis/stable-diffusion-v1-4
  Licence: CreativeML OpenRAIL-M — https://huggingface.co/spaces/CompVis/stable-diffusion-license

- **Stable Diffusion v1-5 (runwayml/stable-diffusion-v1-5).** Used in Chapter 25 (Hugging Face Transformers and Gradio), referenced in the notebook by this repository ID. RunwayML withdrew its Hugging Face organization in August 2024, so this exact repository ID may no longer resolve; the actively maintained mirror is `stable-diffusion-v1-5/stable-diffusion-v1-5`, which states the same licence.
  Source: https://huggingface.co/runwayml/stable-diffusion-v1-5 (mirror: https://huggingface.co/stable-diffusion-v1-5/stable-diffusion-v1-5)
  Licence: CreativeML OpenRAIL-M — https://huggingface.co/spaces/CompVis/stable-diffusion-license

- **Stable Diffusion x2 latent upscaler (stabilityai/sd-x2-latent-upscaler).** Used in Chapter 25. This checkpoint uses a stricter licence than v1-4/v1-5 and should not be assumed to share their terms.
  Source: https://huggingface.co/stabilityai/sd-x2-latent-upscaler
  Licence: CreativeML Open RAIL++-M License — https://huggingface.co/stabilityai/stable-diffusion-2/blob/main/LICENSE-MODEL

- **ControlNet QR-code checkpoint (DionTimmer/controlnet_qrcode-control_v1p_sd15).** A community ControlNet checkpoint used in Chapter 25. Its licence is independent of the Stable Diffusion 1.5 base model it was fine-tuned on and must not be inferred from the base model's terms; the repository's own licence tag governs it.

Source: https://huggingface.co/DionTimmer/controlnet_qrcode-control_v1p_sd15
Licence: OpenRAIL++, as specified by the model repository's own licence tag (the repository does not expose a separate dedicated licence file beyond this tag).

- **3D Gaussian Splatting — official reference implementation.** The official INRIA implementation (Kerbl et al. [28]) is released under a licence with non-commercial research/evaluation terms. This restriction applies to that particular reference implementation; it is not a statement that 3D Gaussian Splatting as a scientific method, or this memorandum, is non-commercial. Chapter 32 does not train with this implementation; it downloads and displays a precomputed splat scene (see "Third-party demonstration images and media" below) using a separately licensed third-party WebGL viewer.
  Project source: https://github.com/graphdeco-inria/gaussian-splatting
  Licence: Gaussian-Splatting research/evaluation licence — https://github.com/graphdeco-inria/gaussian-splatting/blob/main/LICENSE.md

Third-party demonstration images and media

This section lists third-party photographs, video, and pre-computed scene files reproduced or processed in the memorandum's demonstration chapters, with source URLs and licence/terms where known.

- **Club Big Breakfast, The Coffee Club Riverton, 2026 (02).** Creator: Bahnfrend. Used as a YOLO11 detection/segmentation demonstration image (Chapters 12 and 16).
  Source: https://commons.wikimedia.org/wiki/File:Club_Big_Breakfast,_The_Coffee_Club_Riverton,_2026_%2802%29.jpg
  Licence: CC BY-SA 4.0 — https://creativecommons.org/licenses/by-sa/4.0/

- **EFTA00000247 – Hotel room with deep blue walls, a plaid carpet, and a large bed with a floral headboard.** U.S. Federal Bureau of Investigation / U.S. Department of Justice, via Wikimedia Commons. Used as a YOLO11 demonstration image (Chapters 12 and 16).
  Source: https://commons.wikimedia.org/wiki/File:EFTA00000247_-_Hotel_room_with_deep_blue_walls_a_plaid_carpet_and_a_large_bed_with_floral_headboard_T
  Status: public-domain U.S. federal-government work (no Creative Commons licence is asserted; the public-domain basis is the work's status as a U.S. federal-government work).

- **Basketball demonstration video.** Creator: Pavel Danilyuk, Pexels. Used for the RAFT optical-flow demonstration in Chapter 29; fetched in the notebook from a PyTorch tutorial-assets mirror (`pexelscom_pavel_danilyuk_basketball_hd.mp4`).
  Original source: https://www.pexels.com/video/a-man-playing-a-game-of-basketball-5192157/
  Notebook download mirror: https://download.pytorch.org/tutorial/pexelscom_pavel_danilyuk_basketball_hd.mp4
  Licence: Pexels License — https://www.pexels.com/license/

- **Ultralytics bus.jpg and zidane.jpg.** Standard Ultralytics demonstration images, downloaded automatically by the Ultralytics package for the YOLO11 detection, segmentation, and pose-estimation chapters (12, 16, 17). No separate Creative Commons licence is asserted for these two images; they are distributed as part of the Ultralytics project referenced above.
  Sources: https://ultralytics.com/images/bus.jpg, https://ultralytics.com/images/zidane.jpg

Project: https://github.com/ultralytics/assets

- **Meta SAM 2 demo assets (truck.jpg, groceries.jpg, bedroom.zip).** Downloaded directly from Meta's SAM 2 repository and asset host for the Chapter 18 demonstrations. These are demonstration files bundled with the SAM 2 codebase; Meta's source does not identify them as SA-V samples, so they are not described as such here (see the note under "Datasets" above).
  Sources: https://raw.githubusercontent.com/facebookresearch/sam2/main/notebooks/images/truck.jpg, https://raw.githubusercontent.com/facebookresearch/sam2/main/notebooks/images/groceries.jpg, https://dl.fbaipublicfiles.com/segment_anything_2/assets/bedroom.zip

- **Transformer architecture diagram (Chapter 25, p. 516).** Creator: dvgodoy, via Wikimedia Commons.
  Source: https://commons.wikimedia.org/wiki/File:Transformer,_full_architecture.png
  Licence: CC BY 4.0 — https://creativecommons.org/licenses/by/4.0/

- **Gaussian Splatting demonstration scene (gs_FF3_lumix_4k, locally saved as "truck.ply").** Chapter 32 downloads a precomputed 3D Gaussian Splatting scene file from the `cakewalk/splat-data` Hugging Face repository. That repository states that its splat scenes "have been converted from various different sources with various different licenses" and does not provide a blanket licence covering all of its contents; the exact upstream origin and licence of this specific scene file could not be established from the available repository metadata.
  Scene file source: https://huggingface.co/cakewalk/splat-data (Hugging Face repository "cakewalk/splat-data")
  The specific upstream source and licence of this scene could not be established conclusively from the available source metadata. The scene is therefore retained with provenance explicitly marked for manual verification (see "Items requiring manual legal/source verification").

- **Gaussian Splatting WebGL viewer.** The interactive viewer used to display the scene above is a separate work from the scene data itself and carries its own, independently verified licence, which does not extend to the scene it renders.
  Project: `antimatter15/splat` — Kevin Kwok
  Source: https://github.com/antimatter15/splat
  Licence: MIT License — https://github.com/antimatter15/splat/blob/main/LICENSE
  This memorandum embeds/links to the hosted viewer rather than redistributing its source code, so the full MIT licence text is not reproduced here; it would be preserved with any redistributed copy of the viewer's own source files.

Figures created by the authors

Except where an individual figure credit identifies an external source, diagrams, plots, training curves, model-output visualizations, and typeset figures in this volume were produced by the course authors. Architecture diagrams that follow a published architecture are redrawn or adapted from the cited publication rather than reproduced as facsimiles.

References

[1] Krizhevsky, A. (2009). *Learning multiple layers of features from tiny images* (Technical Report). University of Toronto. https://www.cs.toronto.edu/~kriz/cifar.html

[2] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE, 86*(11), 2278–2324.

[3] Xiao, H., Rasul, K., & Vollgraf, R. (2017). Fashion-MNIST: A novel image dataset for benchmarking machine learning algorithms. *arXiv*. https://doi.org/10.48550/arXiv.1708.07747

[4] Tzanetakis, G., & Cook, P. (2002). Musical genre classification of audio signals. *IEEE Transactions on Speech and Audio Processing, 10*(5), 293–302.

[5] Maas, A. L., Daly, R. E., Pham, P. T., Huang, D., Ng, A. Y., & Potts, C. (2011). Learning word vectors for sentiment analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics (ACL)* (pp. 142–150).

[6] Everingham, M., Van Gool, L., Williams, C. K. I., Winn, J., & Zisserman, A. (2010). The PASCAL Visual Object Classes (VOC) challenge. *International Journal of Computer Vision, 88*(2), 303–338.

[7] Wang, L., Shi, J., Song, G., & Shen, I.-F. (2007). Object detection combining recognition and segmentation. In *Proceedings of the Asian Conference on Computer Vision (ACCV)* (pp. 189–199).

[8] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., & Zitnick, C. L. (2014). Microsoft COCO: Common objects in context. In *Proceedings of the European Conference on Computer Vision (ECCV)* (pp. 740–755).

[9] Chilamkurthy, S. (n.d.). *Transfer learning for computer vision tutorial*. PyTorch Tutorials. https://docs.pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

[10] World Health Organization, Global Health Observatory. (n.d.). *Life expectancy and healthy life expectancy* [Data set; Kaggle redistribution by K. Rajarshi]. Kaggle. https://www.kaggle.com/datasets/kumarajarshi/life-expectancy-who

[11] Box, G. E. P., & Jenkins, G. M. (1976). *Time series analysis: Forecasting and control*. Holden-Day.

[12] Ravi, N., Gabeur, V., Hu, Y.-T., Hu, R., Ryali, C., Ma, T., Khedr, H., Rädle, R., Rolland, C., Gustafson, L., Mintun, E., Pan, J., Alwala, K. V., Carion, N., Wu, C.-Y., Girshick, R., Dollár, P., & Feichtenhofer, C. (2024). SAM 2: Segment anything in images and videos. *arXiv*. https://doi.org/10.48550/arXiv.2408.00714

[13] He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 770–778).

[14] Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1409.1556

[15] Ronneberger, O., Fischer, P., & Brox, T. (2015). U-Net: Convolutional networks for biomedical image segmentation. In *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI)* (pp. 234–241).

[16] Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[17] Jocher, G., & Qiu, J. (2024). *Ultralytics YOLO11* [Computer software]. https://github.com/ultralytics/ultralytics

[18] Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)* (pp. 4171–4186). Association for Computational Linguistics. https://doi.org/10.18653/v1/N19-1423

[19] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). *Language models are unsupervised multitask learners* [Technical report]. OpenAI. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[20] Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research, 21*(140), 1–67.

[21] Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2021). An image is worth 16×16 words: Transformers for image recognition at scale. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

[22] Tiedemann, J., & Thottingal, S. (2020). OPUS-MT — Building open translation services for the world. In *Proceedings of the 22nd Annual Conference of the European Association for Machine Translation (EAMT)* (pp. 479–480).

[23] Rombach, R., Blattmann, A., Lorenz, D., Esser, P., & Ommer, B. (2022). High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 10684–10695).

[24] Zheng, P., Gao, D., Fan, D.-P., Liu, L., Laaksonen, J., Ouyang, W., & Sebe, N. (2024). Bilateral reference for high-resolution dichotomous image segmentation. *CAAI Artificial Intelligence Research, 3*, Article 9150038.

[25] Ranftl, R., Bochkovskiy, A., & Koltun, V. (2021). Vision transformers for dense prediction. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*.

[26] Teed, Z., & Deng, J. (2020). RAFT: Recurrent all-pairs field transforms for optical flow. In *Proceedings of the European Conference on Computer Vision (ECCV)* (pp. 402–419).

[27] Mildenhall, B., Srinivasan, P. P., Tancik, M., Barron, J. T., Ramamoorthi, R., & Ng, R. (2020). NeRF: Representing scenes as neural radiance fields for view synthesis. In *Proceedings of the European Conference on Computer Vision (ECCV)* (pp. 405–421).

[28] Kerbl, B., Kopanas, G., Leimkühler, T., & Drettakis, G. (2023). 3D Gaussian splatting for

real-time radiance field rendering. *ACM Transactions on Graphics, 42*(4).

[29] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[30] Park, D. S., Chan, W., Zhang, Y., Chiu, C.-C., Zoph, B., Cubuk, E. D., & Le, Q. V. (2019). SpecAugment: A simple data augmentation method for automatic speech recognition. In *Proceedings of Interspeech* (pp. 2613–2617).

[31] Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 248–255). https://doi.org/10.1109/CVPR.2009.5206848

[32] Buda, M., Saha, A., & Mazurowski, M. A. (2019). Association of genomic subtypes of lower-grade gliomas with shape features automatically extracted by a deep learning algorithm. *Computers in Biology and Medicine, 109*, 218–225. https://doi.org/10.1016/j.compbiomed.2019.05.002

[33] Pedano, N., Flanders, A. E., Scarpace, L., Mikkelsen, T., Eschbacher, J. M., Hermes, B., Sisneros, V., Barnholtz-Sloan, J., & Ostrom, Q. (2016). *The Cancer Genome Atlas Low Grade Glioma Collection (TCGA-LGG)* (Version 3) [Data set]. The Cancer Imaging Archive. https://doi.org/10.7937/K9/TCIA.2016.L4LTD3TK

[34] Tatoeba Association. (n.d.). *Tatoeba sentence and link exports* [Data set]. https://downloads.tatoeba.org/exports/

[35] HuggingFaceH4. (n.d.). *MATH-500* [Data set]. Hugging Face. https://huggingface.co/datasets/HuggingFaceH4/MATH-500

[36] Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., Lee, T., Leike, J., Schulman, J., Sutskever, I., & Cobbe, K. (2023). Let's verify step by step. *arXiv*. https://doi.org/10.48550/arXiv.2305.20050

[37] Seitz, S. M., Curless, B., Diebel, J., Scharstein, D., & Szeliski, R. (2006). A comparison and evaluation of multi-view stereo reconstruction algorithms. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 519–526). https://doi.org/10.1109/CVPR.2006.19

[38] Zhang, L., Rao, A., & Agrawala, M. (2023). Adding conditional control to text-to-image diffusion models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)* (pp. 3836–3847).