Smart Python for Machine Learning and Intelligent Systems: Deep Learning, Transfer Learning, and AI Engineering | Part 2 extends the foundations established in Part 1 by introducing the modern techniques that drive today's intelligent systems. The book provides a practical, implementation-oriented approach to deep learning with Python, covering neural networks, convolutional neural networks (CNNs), recurrent neural networks (RNNs), LSTMs, GRUs, and generative adversarial networks (GANs). It also explores transfer learning through feature reuse, fine-tuning, and domain adaptation, followed by advanced deep learning architectures including ResNet and other state-of-the-art models. The final chapters focus on AI engineering, model optimization, deployment, inference benchmarking, pruning, quantization, and the development of efficient production-ready intelligent systems. Throughout the book, theoretical concepts are reinforced with complete Python implementations, practical experiments, performance evaluation, and real-world case studies. Together with Part 1, this volume provides a comprehensive guide to modern machine learning, deep learning, and AI engineering using Python.



Prof. Dr. Alexander I. Iliev is a professor and researcher in Artificial Intelligence, Machine Learning, Big Data Analytics, and Smart Systems. He holds three patents and has authored numerous scientific publications and books. He is affiliated with UC Berkeley, SRH University of Applied Sciences Heidelberg, and the Bulgarian Academy of Sciences.

9 786630 210927

Smart Python | Applied ML | Part 2

FOR AUTHOR USE ONLY

Alexander I. Iliev

LAP LAMBERT Academic Publishing



Alexander I. Iliev

Smart Python for Machine Learning and Intelligent Systems

Deep Learning, Transfer Learning, and AI Engineering | Part 2

Alexander I. Iliev

Smart Python for Machine Learning and Intelligent Systems

FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY

Alexander I. Iliev

Smart Python for Machine Learning and Intelligent Systems

Deep Learning, Transfer Learning, and AI Engineering | Part 2

FOR AUTHOR USE ONLY

LAP LAMBERT Academic Publishing

Imprint
Any brand names and product names mentioned in this book are subject to trademark, brand or patent protection and are trademarks or registered trademarks of their respective holders. The use of brand names, product names, common names, trade names, product descriptions etc. even without a particular marking in this work is in no way to be construed to mean that such names may be regarded as unrestricted in respect of trademark and brand protection legislation and could thus be used by anyone.

Cover image: www.ingimage.com

Publisher:
LAP LAMBERT Academic Publishing
is a trademark of
Dodo Books Indian Ocean Ltd. and OmniScriptum S.R.L publishing group

120 High Road, East Finchley, London, N2 9ED, United Kingdom
Str. Armeneasca 28/1, office 1, Chisinau MD-2012, Republic of Moldova, Europe
Managing Directors: Ieva Konstantinova, Victoria Ursu
info@omniscriptum.com

Printed at: see last page
ISBN: 978-66-30-21092-7

Copyright © Alexander I. Iliev
Copyright © 2026 Dodo Books Indian Ocean Ltd. and OmniScriptum S.R.L publishing group

FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY

Smart **P**ython for **M**achine **L**earning and **I**ntelligent **S**ystems

Deep Learning, Transfer Learning, and AI Engineering

FOR AUTHOR USE ONLY

Part 2

Author:

Prof Dr Alexander Iliev Iliev

Scientific Editors:

Prof Dr Reiner Creutzburg
SRH University of Applied Sciences, Berlin

Prof Dr Daniel Arias Aranda
University of Granada, Granada

A Note from the Author:

This book, ***Smart Python for Machine Learning and Intelligent Systems: Deep Learning, Transfer Learning, and AI Engineering | Part 2***, is the second volume in a series dedicated to machine learning and intelligent systems. It builds upon the foundations established in Part 1 by introducing deep learning, transfer learning, AI engineering, deployment, and optimization through practical Python implementations and real-world examples.

FOR AUTHOR USE ONLY

Currently, this textbook is used in the accredited ***Big Data and Artificial Intelligence*** master's degree program at the ***School of Technology and Architecture, SRH University of Applied Sciences Heidelberg***. It serves as the principal textbook for the course ***Machine Learning***.

This volume complements ***Smart Python for Machine Learning and Intelligent Modeling: Foundations and Classical Machine Learning | Part 1***. While Part 1 introduces the principles of classical machine learning, this volume focuses on deep learning, transfer learning, AI engineering, deployment, and optimization, completing the two-volume series on modern machine learning with Python.

Table of Contents

PART I - Deep Learning Foundations **13**
Chapter 1 - Neural Networks **13**
Introduction and Literature Review 13
1.1.1. Architecture 15
1.1.2. Model 17
1.1.3. Methods 18
1.1.4. Formulas 19
1.1.5. Results 21
1.1.6. Discussion 23
1.2. Conclusion 24
Chapter 2 - Convolutional Neural Networks (CNN) **25**
Introduction 25
2.1. Basic Structure of a Neural Network 26
2.2. Introduction to Convolutional Neural Networks (CNN) 29
2.3. CNN vs Multi-Layer Perceptron (MLP) 30
2.4. Convolution Operation 31
2.4.1. Padding Types in CNN 32
2.4.2. Feature Maps 35
2.5. Pooling Layer 37
2.6. CNN Architecture Flow 38
2.7. CNN Training Process 41
2.8. MNIST Handwritten Digit Classification using CNN 43
2.9. Performance Evaluation 44
Chapter 3 — Recurrent Neural Networks (RNN) **46**
Introduction and Literature Review 46
3.1. Recurrent Neural Networks for Sequence Processing 47
3.2. The Vanishing and Exploding Gradient Problem 48
3.3. Gated Architectural Mechanisms 49
3.4. Long Short-Term Memory (LSTM) 49
3.5. Gated Recurrent Unit (GRU) 50
3.6. Foundational Sequence Modeling Concepts 54

FOR AUTHOR USE ONLY

3.7. Time Series Forecasting Frameworks 55
3.8. Hyperparameter and Optimization Control 57
3.9. Sentiment Analysis 59
3.9.1. Methodology and Dataset 59
3.9.2. Results and Discussion 65
3.10. Time Series Application 67
3.10.1. Methodology and Dataset 67
3.10.2. Results and Discussions 76
3.11. Conclusion 78
Chapter 4 — Long Short-Term Memory (LSTM) 80
4.1. Recurrent Neural Networks and the Vanishing Gradient Problem 80
4.1.1. Why Sequence Models? 80
4.1.2. The Basic RNN Architecture 80
4.1.3. The Vanishing Gradient Problem 81
4.1.4. Where Vanilla RNNs Still Work 81
4.2. Long Short-Term Memory (LSTM) 82
4.2.1. Motivation: Remembering What Matters 82
4.2.2. Inputs and Outputs of an LSTM Cell 82
4.2.3. The Full LSTM Computation 83
4.2.4. Why the Vanishing Gradient is Solved 84
4.2.5. Gate Mechanisms 84
4.2.5.1. The Role of Gating 84
4.2.5.2. The Forget Gate 85
4.2.5.3. The Input Gate 85
4.2.5.4. The Output Gate 86
4.2.5.5. Comparison 86
4.2.6. The Gated Recurrent Unit (GRU): A Streamlined Alternative 86
4.3. Memory Cell Architecture 87
4.3.1. Inside the LSTM Cell: A Modular View 87
4.3.2. Parameter Count 88
4.3.3. Stacked (Multi-Layer) LSTMs 88
4.3.4. Dropout in LSTMs 89
4.3.5. Bidirectional LSTMs 89
Code Example: LSTM for Time-Series Forecasting 90

FOR AUTHOR USE ONLY

4.3.6. Common Architecture Choices and Hyperparameters 92
4.3.7. Summary: Sequential Models at a Glance 93
Chapter 5 — Generative Adversarial Networks (GANs) 94
Introduction 94
5.1. Core Concept 95
5.1.1. Generator vs Discriminator 95
5.1.2. Adversarial Loss 96
5.2. Methodology 97
5.2.1. Dataset 97
5.2.2. Generator Architecture 98
5.2.3. Discriminator Architecture 98
5.2.4. Training Configuration 98
5.2.5. Evaluation Strategy 98
5.3. Implementation 99
5.3.1. Environment Setup and Data Loading 99
5.3.2. Generator Network 100
5.3.3. Discriminator Network 101
5.3.4. Loss Functions and Optimizers 102
5.3.5. Training Loop 103
5.3.6. Image Generation and Visualization 105
5.3.7. Loss Curve Visualization 105
5.4. Evaluation 106
5.4.1. Visual Quality of Generated Digits 106
5.4.2. Training Loss Analysis 108
5.5. Discussion 108
5.6. Conclusion 109
PART II — Transfer Learning & Advanced Architectures 110
Chapter 6 — Transfer Learning Principles 110
6.1 Introduction to Transfer Learning 110
6.1.1 What problem does it solve? 110
6.1.2 Three pillars covered in this guide 110
6.1.3 Mini-case: a 600-image plant-disease classifier 110
6.2 Feature Reuse 111
6.2.1 The Feature Hierarchy 111

FOR AUTHOR USE ONLY

6.2.2 The Frozen Feature Extractor Pattern 111
6.2.3 PyTorch implementation 112
6.2.4 Caching Backbone Embeddings 113
6.2.5 TensorFlow / Keras Implementation 113
6.3 Fine-Tuning Strategies 114
6.3.1 Full vs. Partial Fine-Tuning 114
6.3.2 Progressive Unfreezing 114
6.3.3 Discriminative Learning Rates 115
6.3.4 PyTorch: Progressive Unfreezing with Per-Layer LR 115
6.3.5 TensorFlow: two-stage fine-tuning 117
6.4 Domain Adaptation 119
6.4.1 Types of Distribution Shift 119
6.4.2 Method Families 119
6.4.3 PyTorch: DANN with Gradient Reversal 120
6.4.4 TensorFlow: deep CORAL alignment 122
6.5 Practical Cheat Sheet & References 123
Chapter 7 — ResNet Architecture 125
Introduction to ResNet 125
Problem of Very Deep Neural Networks 126
Residual Learning Concept 128
7.1 Residual Blocks 130
7.1.1 ResNet Architecture Flow 133
7.2 Bottleneck Design 135
7.2.1 ResNet34 Architecture 139
7.2.2 ResNet50 Architecture 141
7.3 ResNet34 vs ResNet50 143
7.3.1 Experimental Benchmark Example 146
7.3.2 Practical Implementation using Keras 146
Chapter 8 - Case Study: ResNet50 Transfer Learning for Medical Imaging 150
8.1 Introduction 150
8.2 Core Concept 151
8.2.1 Transfer Learning and Fine-Tuning 151
8.2.2 ResNet50 Architecture 152
8.2.3 Evaluation Metrics for Medical Classification 153

FOR AUTHOR USE ONLY

8.2.4 Comparative Study of Pretrained Weights 153
8.3 Methodology 154
8.3.1 Dataset 154
8.3.2 Dataset Preparation and Preprocessing 154
8.3.3 Model Architecture and Fine-Tuning Strategy 154
8.3.4 Training Configuration 155
8.3.5 Evaluation Strategy 155
8.4 Implementation 155
8.4.1 Environment Setup and Dataset Download 156
8.4.2 Dataset Preparation and Data Loaders 156
8.4.3 Model Construction - Pretrained ResNet50 158
8.4.4 Model Construction - ResNet50 from Scratch 158
8.4.5 Training Loop 159
8.4.6 Model Evaluation 160
8.4.7 Visualization - Training Curves and Comparative Results 161
8.4.8 Individual Training Curves - Pretrained Model 162
8.4.9 Individual Training Curves -Scratch Model 162
8.5. Evaluation 163
8.5.1 Dataset Preparation Results 163
8.5.2 ResNet50-Pretrained Performance 164
8.5.3 ResNet50-Scratch Performance 165
8.5.4 Comparative Study of Pretrained vs. Scratch 166
8.6 Discussion 167
8.7 Conclusion 168
PART III — Python Implementation & Engineering 170
Chapter — 9 The Scikit-Learn Ecosystem 170
9.1 Introduction 170
9.2 The Scikit-Learn Object-Oriented API Architecture 170
Architecture Overview: 171
9.2.1 The Estimator Interface 171
9.2.2 The Transformer Interface 171
9.2.3 The Predictor Interface 172
9.3 Pipelines: Structural Integrity and Preventing Data Leakage 172
9.3.1 The Mechanics of Leakage via Empirical Scaling 172

FOR AUTHOR USE ONLY

9.3.2 The Pipeline Architecture Solution 173
9.4 Cross-Validation: Statistical Resampling Theory 173
9.4.1 K-Fold Cross-Validation Mechanics 174
9.4.2 Stratified K-Fold Cross-Validation 174
9.4.3 Advanced Cross-Validation Strategies 174
9.5 Grid Search: Systematic Hyperparameter Optimization 175
9.5.1 Optimization Mechanics 175
9.5.2 Computational Optimization Tactics 175
9.6 Python Implementation: Unified Engineering Pipeline 176
9.7 Comprehensive Code Metrics and Analysis 178
9.7.1 Cross-Validation Performance Topography 178
9.7.2 Generalization Performance on Unseen Test Data 178
9.8. Discussion and Methodological Interpretation 179
9.8.1 Resolving the Independence Fallacy 179
9.8.2 The Necessity of Automated Cross-Validation 180
9.9. Conclusion and Future Research Directions 180
Chapter – 10 TensorFlow & Keras 181
10.1 Introduction to TensorFlow & Keras 181
10.1.1 What problem does it solve? 181
10.1.2 Three pillars covered in this guide 181
10.1.3 Mini-case: refactoring a 200-line training script 181
10.2 Functional API 182
10.2.1 Sequential, Functional, and subclassing 182
10.2.2 Building a multi-input, multi-output model 183
10.2.3 Inspecting and visualizing the graph 184
10.2.4 Shared layers and model composition 185
10.3 Transfer Learning Workflow 185
10.3.1 Selecting a pre-trained backbone 185
10.3.2 Frozen feature extraction 186
10.3.3 Two-stage fine-tuning 187
10.3.4 Data pipelines with tf.data 187
10.4 Early Stopping and Dropout 188
10.4.1 The Regularization Budget 188
10.4.2 EarlyStopping configured correctly 188

FOR AUTHOR USE ONLY

10.4.3 Dropout placement and rate selection 189
10.4.4 Combining with weight decay and label smoothing 190
10.5 Practical Cheat Sheet & References 191
Decision flow 191
Chapter 11 — Smart Python with PyTorch 192
Introduction 192
Setting Up PyTorch 193
11.1. PyTorch Overview 195
11.2. Understanding the PyTorch Workflow 197
11.3. Understanding Tensors 197
11.3.1. Tensor Dimensions 198
11.3.2. Creating Tensors 199
11.3.3. Generating Tensors Automatically 199
11.3.4. Tensor Data Types 200
11.3.5. Tensor Shape and Dimensions 200
11.3.6. Tensor Indexing and Slicing 200
11.3.7. Mathematical Operations on Tensors 201
11.3.8. Reshaping Tensors 201
11.3.9. CPU and GPU Tensors 202
11.4. Autograd in PyTorch 202
11.4.1. Computational Graphs 203
11.4.2. Enabling Gradient Tracking 204
11.4.3. Performing Operations with Autograd 204
11.4.4. Computing Gradients Using backward() 205
11.4.5. Gradient Accumulation 205
11.4.6. Clearing Gradients 206
11.4.7. Disabling Gradient Tracking 207
11.5. Training Loops in PyTorch 207
11.5.1. Understanding the Training Process 208
11.5.2. Forward Pass 208
11.5.3. Loss Calculation 209
11.5.4. Backward Pass 209
11.5.5. Parameter Update 209
11.5.6. Epochs and Iterations 209

FOR AUTHOR USE ONLY

11.5.7. Common Optimizers 211
11.5.8. Building a Simple Neural Network 211
11.5.9. Complete Training Loop 212
11.5.10. Monitoring Training Progress 213
11.5.11. Saving a Trained Model 213
11.5.12. Loading a Saved Model 213
11.6. Binary Classification Using PyTorch 214
11.6.1. Creating a Synthetic Dataset 214
11.6.2. Defining the Neural Network 215
11.6.3. Defining the Loss Function and Optimizer 216
11.6.4. Training the Model 216
11.6.5. Making Predictions 217
11.6.6. Converting Probabilities into Class Labels 218
11.6.7. Sample Results 218
Chapter — 12 Data Engineering for Machine Learning 219
Prelude 219
Introduction 219
Dataset Description 220
Data Leakage 220
Feature Engineering 221
12.1. Feature Scaling 222
12.2. Handling Imbalance 224
12.3. Data Augmentation 226
12.4. Machine Learning Pipelines 227
12.4.1. Implementation 228
12.4.2. Results and Discussion 232
12.5 Conclusion 235
Chapter 13 — Model Deployment 237
13.1 Introduction to Model Deployment 237
13.1.1. What problem does it solve? 237
13.1.2 Three pillars covered in this guide 237
13.1.3 Mini case: taking a trained classifier to production 237
13.2. Saving and Loading Models 238
13.2.1 The format landscape: SavedModel, H5, TorchScript, ONNX 238

FOR AUTHOR USE ONLY

13.2.2 Saving and restoring in Keras 239
13.2.3 PyTorch: state_dict and TorchScript 240
13.2.4 Versioning and checkpointing strategies 241
13.3. REST APIs 242
13.3.1 Why a REST layer belongs between model and client 242
13.3.2 FastAPI: the minimal serving skeleton 242
13.3.3 Input validation, batching, and async inference 244
13.3.4 TensorFlow Serving and Triton for production 245
13.4. Edge Deployment 246
13.4.1. The model-size / latency / accuracy triangle 246
13.4.2 Exporting to ONNX 247
13.4.3 Quantization with TFLite and ONNX Runtime 248
13.4.4 Running inference in the browser 249
13.5. Practical Cheat Sheet & References 251
Chapter 14 — Performance Optimization 253
14.1. Introduction 253
14.2. Core Concept 254
14.2.1 GPU vs. CPU Performance Optimization 254
14.2.2 Model Compression — Pruning 255
14.2.3 Quantization 256
14.3. Methodology 257
14.3.1 Dataset 257
14.3.2 Model Architecture 257
14.3.3 Baseline Training Configuration 258
14.3.4 Experimental Design 258
14.4. Implementation 259
14.4.1 Environment Setup and CIFAR-10 Data Loading 259
14.4.2 Baseline ResNet18 Training 261
14.4.3 GPU vs. CPU Inference Benchmarking 263
14.4.4 Magnitude-Based Pruning 264
14.4.5 Dynamic INT8 Quantization 266
14.4.6 Comparative Visualization and Summary 268
14.5. Evaluation 270
14.5.1 Baseline Model Performance 270

FOR AUTHOR USE ONLY

14.5.2 GPU vs. CPU Benchmarking Results 270
14.5.3 Pruning Results 271
14.5.4 Quantization Results 272
14.5.5 Comparative Summary 272
14.6. Discussion 273
14.7. Conclusion 274
Chapter 15 — From Algorithms to Intelligent Systems 276
15.1 Introduction and Literature Review 276
15.2 The Bias-Variance Tradeoff 277
15.3. Hyperparameter Optimization and Model Evaluation 279
15.4. Regularization Strategies in Neural Networks 279
15.5. Model Selection Strategy 282
15.5.1. Methodology and Dataset 283
15.5.2 Results and Discussion 290
15.6. Architecture Design Philosophy 294
15.6.1. Methodology and Dataset 295
15.6.2. Results and Discussion 300
15.7. Conclusion 305
References 307

FOR AUTHOR USE ONLY

PART I - Deep Learning Foundations

Chapter 1 - Neural Networks

Introduction and Literature Review

In this technological age, the start of digital evolution and upcoming constant improvements have become the center of almost every aspect of daily life. The personalized recommendations on digital platforms, facial recognition features for biometric security or categorical classification, and automated decision support systems serve as prominent examples of these applications (Zhou et al., 2017). Most of the existing smart systems and modern software architectures are shaped by Machine Learning (ML) Algorithms.

Essentially, machine learning eliminates the need for explicit and advanced programming; it autonomously learns meaningful patterns, correlations, and statistical structures from historical datasets, making highly accurate predictions for future scenarios (Jordan & Mitchell, 2015).

FOR AUTHOR USE ONLY

The neural network is one of the powerful subfields of machine learning. These systems are developed by mimicking a real human brain, neurons, and their mathematical working principles. The structure of artificial neural networks consists of an input layer, a hidden layer, and an output layer, all connected. All the inputs are received at the input layer. A hidden layer can be one or more layers in which information is abstracted, and feature inference is performed. The final output is produced in the output layer.

This multilayer structure can model nonlinear functions that traditional methods cannot handle (Goodfellow et al., 2016, p. 194). Therefore, the system can capture highly complex and hidden patterns within data. These networks have become the industry standard in many critical sectors. It is widely used in computer vision architectures, such as object and face recognition; natural language processing (NLP) applications, including machine translation; and other areas like financial risk analysis and medical diagnostic systems (LeCun et al., 2015).

The most fundamental piece of an artificial neural network is a perceptron. The perceptron is a simple mathematical calculation unit that accepts multiple inputs and assigns a weight to each. Next, a bias term is added to the weighted input sum. The non-linear activation function is then applied to this total value, and the final output is produced. One perceptron can solve only basic problems

that can be split linearly (Rosenblatt, 1958). However, these units are combined to form a Multi-Layer Perceptron (MLP) and can model complex non-linear relationships between inputs easily.

The MLPs are trained using the backpropagation algorithm, which computes the differences between the model's prediction and the target value. The algorithm propagates this error from the output layer to the input layer, that is, backwards across the network. In this process, the contribution of each weight to the total error is also calculated by using the chain rule (Rumelhart et al., 1986). This quantity is known as the gradient. To update the weights, these accumulated gradients are processed with the gradient descent algorithm. The weights are updated in stages to minimize the error (Goodfellow et al., 2016, p. 80). This optimization process is repeated thousands of times, in epochs, on the training data. As a result, the model's error rate gradually decreases, and the prediction efficiency of the system increases.

Activation functions are mathematical functions that determine the output corresponding to the inputs of an artificial neuron. These functions serve to integrate the non-linear structures into the model. Linear models fail to capture complex and high-dimensional patterns in the datasets without an activation function. Because of this structural constraint, activation functions are considered one of the most fundamental components of deep neural networks.

FOR AUTHOR USE ONLY

Currently, in hidden layers, the most used activation function is ReLU (Rectified Linear Unit). Negative inputs are set to zero in ReLU, while positive ones are retained unchanged and conveyed exactly (Nair & Hinton, 2010). In the output layer, which performs classification and prediction, different activation functions are used in correlation with the problem type. The sigmoid function is preferred for binary classification problems, and the Softmax function is preferred for multi-class problems (Bishop, 2006).

Vanishing gradients are considered one of the most challenging structural problems in the backpropagation process of deep neural networks. This problem becomes more critical as the network depth increases. The gradients calculated in the output layer are traced back to the input layer by using the chain rule. However, during this propagation process, the gradient values are consecutively multiplied together in each layer (Bengio et al., 1994). The gradients shrink exponentially and approach nearly zero by the first layers of the network. When gradients approach zero, the weights on the initial layers are updated very slowly. Consequently, the learning process in the initial layers

may collapse entirely under specific conditions. This makes it difficult for the model to learn properly.

The vanishing gradient problem is observed much more severely, especially when using saturated activation functions such as the sigmoid and hyperbolic tangent (TANH) (Glorot & Bengio, 2010). The ReLU activation function allows gradients to be transferred to the layers before the reduction. In this way, ReLU largely solves the vanishing gradient problem. ReLU has become a standard choice for hidden layers in modern deep neural networks because of this advantage.

Liver diseases are global health problems that progress without symptoms. They can lead to serious complications such as cirrhosis, liver failure, and cancer (Asrani et al., 2019). Diagnosing them in the early stages is important to increase both treatment success and survival rates. In recent years, the growth of digital data in the healthcare sector has also increased the interest in artificial intelligence solutions. Machine learning algorithms have strong potential to provide doctors with efficient decision support by generating early and reliable predictions for diagnosing liver disease.

In this study, the "Liver Patient Dataset" (LPD) from Kaggle was used to detect possible liver diseases. According to the data, an MLP was designed and trained. Using this artificial neural network model, patients' biochemical values are analyzed to detect possible liver diseases.

FOR AUTHOR USE ONLY

1.1. Methodology

Artificial neural networks can be successfully used to solve both classification and regression problems with their flexible architecture. This study aims to analyze the practical performance of the theoretical concepts underlying deep learning. In this process, single and multi-layer perceptron architectures, backpropagation, activation functions, and the vanishing gradient problem have been addressed in an applied manner. The experimental study was selected to observe and test these fundamental mechanisms in a real-world environment.

1.1.1. Architecture

Dataset: Liver Patient Dataset from Kaggle (Srivastava, n.d.)

The system consists of the following steps:

1. Data Collection

2. Data Preparation
3. Model Creation
4. Model Training
5. Model Testing
6. Baseline Comparison
7. Visualization

The dataset consists of 583 patient records, 10 features, and 1 target column, as shown in Table 1.1. The target column contains two classes: “Liver Disease” and “No Liver Disease”. With this structure, the dataset is suitable for binary classification problems.

Table 1.1. Dataset Features and Descriptions.

	Feature	Description
0	Age	Age of the person
1	Gender	Gender of the person
2	TB	Total Bilirubin
3	DB	Direct Bilirubin
4	Alkphos	Alkaline Phosphotase
5	Sgpt	Alamine Aminotransferase
6	Sgot	Aspartate Aminotransferase
7	TP	Total Proteins
8	ALB	Albumin
9	A/G Ratio	Albumin and Globulin Ratio

FOR AUTHOR USE ONLY

In the data preparation phase, categorical variables were first converted to numerical values. The Gender column was coded as Male=1, Female=0, and the Selector column as Liver Disease=1, No Liver Disease=0. Since there were no missing values in the dataset, no padding was performed.

```
# Encode categorical features: Convert Gender to binary numeric values (Male=1, Female=0)
df['Gender'] = df['Gender'].map({'Male': 1, 'Female': 0})
```

```
# Encode target variable: Convert Selector to binary numeric values (Liver Disease=1, No Liver Disease=0)
df['Selector'] = df['Selector'].map({'Liver Disease': 1, 'No Liver Disease': 0})
```

Code Snippet 1.1. Categorical Feature Encoding.

The dataset was split into 80% for training and 20% for testing. As a result of this division, 466 training and 117 test samples were obtained.

```
# Split the dataset into 80% training and 20% testing data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
    )
```

Code Snippet 1.2. Train/Test Split.

As shown in Code snippet 1.3, all features were normalized using StandardScaler. Normalization was applied only to the training data. The test data was transformed using the parameters learned from the training data. This approach is theoretically necessary to prevent information leakage from the test data.

```
# Initialize the StandardScaler to normalize the feature ranges
scaler = StandardScaler()

# Fit the scaler on training data and transform both training and testing sets
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

Code Snippet 1.3. Feature Normalization.

1.1.2. Model

The model was created using the TensorFlow and Keras libraries. A sequential architecture with two hidden layers and one output layer were used. The first hidden layer has 64 neurons, and the second has 32 neurons. In both hidden layers, ReLU was used as an activation function. In the output layer, 1 neuron with a sigmoid activation function was used to suit the binary classification problem. To prevent overfitting, a 30% dropout layer was added after each hidden layer.

FOR AUTHOR USE ONLY

```
# Model 2: Regularized MLP Architecture to Prevent Overfitting
model2 = keras.Sequential([
    layers.Dense(64, activation='relu', input_shape=(10,)),
    layers.Dropout(0.3),
    layers.Dense(32, activation='relu'),
    layers.Dropout(0.3),
    layers.Dense(1, activation='sigmoid')
])
```

Code Snippet 1.4. Regularized MLP Architecture (Model2).

For the optimizer, Adam was used, and binary cross-entropy was chosen as the loss function.

```
# Compile regularized model
model2.compile(
    optimizer="adam", loss="binary_crossentropy", metrics=["accuracy"]
)
```

Code Snippet 1.5. Model Compilation.

The model consists of 5 layers: 2 hidden (Dense) layers, 2 dropout layers, and 1 output layer. The model contains 2,817 trainable parameters. The dropout

layers contain no learnable parameters and are used solely to prevent overfitting.

Table 1.2. Model Architecture Summary.

... Model: "sequential_1"

Layer (type)	Output Shape	Param #
dense_3 (Dense)	(None, 64)	704
dropout (Dropout)	(None, 64)	0
dense_4 (Dense)	(None, 32)	2,080
dropout_1 (Dropout)	(None, 32)	0
dense_5 (Dense)	(None, 1)	33

Total params: 8,453 (33.02 KB)
Trainable params: 2,817 (11.00 KB)
Non-trainable params: 0 (0.00 B)
Optimizer params: 5,636 (22.02 KB)

FOR AUTHOR USE ONLY

1.1.3. Methods

```
# Configure EarlyStopping callback to monitor validation loss and prevent overfitting
early_stop = EarlyStopping(
    monitor='val_loss',
    patience=10, # Stops training if val_loss doesn't improve for 10 consecutive epochs
    restore_best_weights=True # Reverts model weights to the best epoch
    )

# Train the regularized model for up to 100 epochs
history2 = model2.fit(
    X_train, y_train,
    epochs=100,
    batch_size=32,
    validation_split=0.2,
    verbose=1,
    callbacks= [early_stop]
    )
```

Code Snippet 1.6. Early Stopping Configuration and Model Training.

In this study, two models were designed. In the initial model trial, training was performed for 50 epochs, which resulted in overfitting. This shows that the model is memorizing the training data. To prevent overfitting, dropout layers and an early stopping mechanism were integrated into the second model, as shown in Code snippet 1.6.

Table 1.3. Baseline Model Comparison.

Model	Accuracy	Weighted F1	Macro F1
Logistic Regression	0.7607	0.7332	0.6212
Random Forest	0.7350	0.7227	0.6219
MLP (Model 2)	0.7607	0.7300	0.6100

Logistic Regression and Random Forest models were also used to compare the performance of different machine learning models. As seen in Table 1.3, the MLP model and the Logistic Regression model have the same accuracy. Random Forest had the lowest performance with an accuracy rate of 73.50%. If it is evaluated in terms of weighted F1-score, Logistic Regression is slightly above MLP with 0.7332. These results reveal that MLP does not provide a significant advantage over traditional methods. This phenomenon is mostly due to the class imbalance and the relatively small dataset of 583 rows. With a larger and more balanced dataset, the advantage of MLP is expected to become more apparent.

FOR AUTHOR USE ONLY

```
# Train a Logistic Regression model
lr_model = LogisticRegression(max_iter=1000, random_state=42)
lr_model.fit(X_train, y_train)
lr_pred = lr_model.predict(X_test)
```

```
# Train a Random Forest model
rf_model = RandomForestClassifier(n_estimators=100, random_state=42)
rf_model.fit(X_train, y_train)
rf_pred = rf_model.predict(X_test)
```

Code Snippet 1.7. Baseline Model Training.

As shown in Code Snippet 1.7, Logistic Regression and Random Forest models were trained on the same dataset.

1.1.4. Formulas

Sokolova, M., & Lapalme, G. (2009), Goodfellow et al. (2016), Bishop (2006)

1. ReLU (Rectified $Linear$ Unit)Activation Function $= ReLU(x) = max(0, x)$, *(Eq.1.1)*

- x - the input to the neuron

The formula can also be written as:

$f(x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases}$, (*Eq.1.2*)

2. $\text{Sigmoid Activation Function} = \sigma = \frac{1}{1+e^{-x}}$, *(Eq.1.3)*

- x - the input value
- σ - output probability (between 0 and 1)
- e - Euler's number (≈ 2.718)

3. $\text{Binary CrossEntropy Loss} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(p_i) + (1 - y_i)\log(1 - p_i)]$,

(Eq.1.4)

- N - the number of observations
- y_i - actual label (0 or 1)
- p_i - probability predicted by the model

FOR AUTHOR USE ONLY

4. $\text{Gradient Descent (Backpropagation)} = W = W - \alpha.\frac{\partial L}{\partial w}$, *(Eq.1.5)*

- w - weight
- α - learning rate
- $\partial L/\partial w$ - gradient of loss with respect to weight

5. $Precision = \frac{TP}{TP+FP}$, (*Eq.1.6)*

- TP- True Positives
- FP- False Positive

6. $Recall = \frac{TP}{TP+FN}$, *(Eq.1.7)*

- TP- True Positives
- FN- False Negative

7. $F1\ \text{Score} = 2 \times \frac{Precision \times Recall}{Precision + Recall}$, (*Eq.1.8)*

- Precision- Exactness
- Recall- Completeness

1.1.5. Results

The developed deep learning model achieved an overall accuracy rate of 76.07% and a weighted F1-Score of 0.73 on the test set. Since overall accuracy can be misleading in datasets with class imbalance, model performance was evaluated using a weighted F1-score. As shown in Table 1.4, the model performed strongly in the liver disease class but with limited performance in the no liver disease class.

Table 1.4. Model Performance Summary.

Metric	Value
Test Accuracy	0.7607
Test Loss	0.4790
Liver Disease Recall	0.93
No Liver Disease Recall	0.27
Macro F1-Score	0.61
Weighted F1-Score	0.73

FOR AUTHOR USE ONLY

As shown in Table 1.5, in the first model, training accuracy reached 78% while validation accuracy remained at 74%, indicating overfitting. In the second model, with the addition of dropout and early stopping, training was automatically stopped at epoch 33. Validation accuracy exceeded training accuracy, confirming that the model generalized.

Table 1.5. Model Comparison.

	Model 1 (No Dropout)	Model 2 (With Dropout)
Epochs	50	33 (Early Stopped)
Final Train Accuracy	0.78	0.72
Final Val Accuracy	0.74	0.79
Overfitting	Yes	No

The patient classification achieved a precision of 0.79, a recall of 0.93, and an F1-score of 0.85. 81 out of 87 patients were correctly identified, which is a strong result in terms of medical diagnosis, as shown in Table 1.6. In the no liver disease class, the recall was only 0.27. Only 8 out of 30 healthy individuals

were correctly classified. This significant asymmetry is a direct result of the imbalance between 416 patients and 167 healthy individuals in the dataset.

Table 1.6. Classification Report.

Class	Precision	Recall	F1-Score	Support
No Liver Disease	0.57	0.27	0.36	30
Liver Disease	0.79	0.93	0.85	87
Accuracy			0.76	117
Macro Avg	0.68	0.60	0.61	117
Weighted Avg	0.73	0.76	0.73	117

As shown in Figure 1.1, the model demonstrated strong performance in identifying individuals with liver disease but weak performance in identifying healthy individuals. The confusion matrix shows that the model correctly classified 81 out of 87 diseased individuals. However, 22 healthy individuals were mistakenly labeled as diseased.

FOR AUTHOR USE ONLY

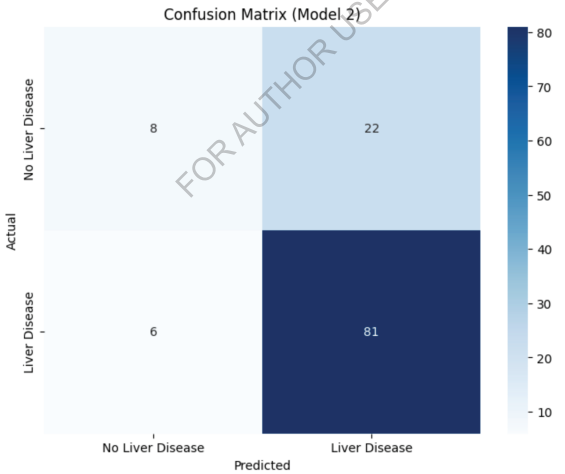


Figure 1.1. Confusion Matrix (Model 2).

The high false-positive rate indicates that the model should not be used as an independent diagnostic tool but rather as a system that supports expert clinical judgment.

As shown in Figure 1.2, the training loss decreased continuously throughout training, while the validation loss remained stable. The fact that the validation accuracy was close to the training accuracy indicates that the dropout mechanism was working effectively.

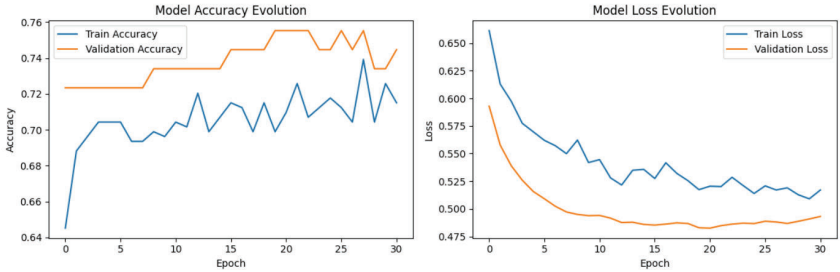


Figure 1.2. Model Accuracy and Loss Evaluation.

1.1.6. Discussion

FOR AUTHOR USE ONLY

In the developed MLP model, 76.07% accuracy and 0.73 weighted F1-score were achieved. Considering the baseline comparison, MLP has the same accuracy level as Logistic Regression and surpasses Random Forest. This result shows that MLP exhibits competitive performance compared to traditional methods under the given dataset size and class imbalance conditions. The weighted F1-score is a more reliable performance metric than overall accuracy when class imbalance is considered (Sokolova & Lapalme, 2009).

The model achieved 93% recall in the diseased class. This means that 81 out of 87 patients were correctly identified. Keeping the false negative rate low in medical diagnostic systems is critically important because a missed patient case can lead to serious clinical consequences (Asrani et al., 2019).

For the healthy class, recall was 0.27. This situation arises because the imbalance of 416 patients to 167 healthy individuals in the dataset causes the model's weight optimization to lean towards the majority class. Macro F1-Score, which is 0.61, also numerically confirms the performance gap between the classes. In future studies, synthetic data enhancement techniques such as SMOTE (Synthetic Minority Over-sampling Technique) can be applied to address this performance loss (Chawla et al., 2002). Alternatively, using cost-sensitive learning methods that increase the weights of the minority group, or a more balanced clinical dataset, can improve the model's performance.

Future studies could improve model performance by using a deeper architecture, different activation functions, or a larger class-balanced dataset. Adding batch normalization and hyperparameter optimization could also be considered (LeCun et al., 2015).

1.2. Conclusion

The method selected for this study is a Multi-Layer Perceptron (MLP), a powerful deep learning architecture, developed to diagnose liver disease at an early stage. The model was trained on the “Liver Patient Dataset”, available on the Kaggle platform and designed using the TensorFlow and Keras libraries, leading deep learning frameworks in the Python ecosystem. It achieved an accuracy of 76.07% and a weighted F1-Score of 0.73% on the test set. The dropout and early stopping mechanisms successfully prevented overfitting, and the model was automatically stopped at epoch 33. Baseline comparison also showed that the outcomes of MLP have the same accuracy performance as logistic regression, though better than random forest. Although a 93% recall rate was achieved in the patient class, class imbalance negatively affected the identification of healthy individuals. Future research aims to overcome these current limitations by using more balanced datasets, data augmentation techniques, and deeper hierarchical architectures. As a result, this model can be used as a tool for doctors to support their clinical decisions.

FOR AUTHOR USE ONLY

Chapter 2 - Convolutional Neural Networks (CNN)

Introduction

In recent years, Artificial Intelligence (AI) and Machine Learning (ML) have become important technologies in many areas of modern life. Applications such as facial recognition, self-driving cars, recommendation systems, and medical image analysis all rely on machine learning techniques. As digital image data continues to increase, the need for systems that can efficiently analyze and understand visual information has also grown significantly.

Traditional machine learning methods often require manual feature extraction before classification can be performed. In image-related tasks, this process can become difficult because images contain large amounts of visual information such as edges, textures, shapes, and colors. To solve these limitations, Deep Learning models were introduced, allowing computers to automatically learn patterns directly from raw data.

FOR JUNIOR USE ONLY

One of the most important deep learning models used for image processing is the Convolutional Neural Network (CNN). CNN is specially designed to process image data efficiently and automatically extract meaningful features from images. Due to its ability to recognize complex visual patterns, CNN is widely used in applications such as image classification, object detection, facial recognition, and medical diagnosis.

Today, CNN models play an important role in many real-world systems. In healthcare, CNN assists in detecting diseases from medical scans such as X-rays and MRI images. In transportation, self-driving vehicles use CNN models to recognize roads, traffic signs, and pedestrians. Similarly, security systems use CNN-based facial recognition technologies for identification and surveillance.

In this chapter, we will explore the fundamental concepts of Convolutional Neural Networks, including convolution operations, feature maps, pooling layers, CNN architecture, and basic CNN implementation using Python. The explanations are designed to provide a beginner-friendly understanding of how CNN models process image data in modern machine learning applications.

Before understanding how Convolutional Neural Networks operate, it is important to first understand the basic structure of a neural network and how information flows through its layers.

2.1. Basic Structure of a Neural Network

Artificial Neural Networks (ANNs) are computational models inspired by the structure and functioning of the human brain. A neural network consists of interconnected processing units called neurons, which work together to learn patterns from data. These networks form the foundation of many Deep Learning models, including Convolutional Neural Networks (CNNs).

In a basic neural network, information flows through multiple layers. Each layer performs mathematical operations on the input data and passes the processed information to the next layer. During training, the network gradually adjusts its internal parameters to improve prediction accuracy and minimize errors.

The simplest neural network structure generally consists of three main layers: the input layer, hidden layer, and output layer. Each layer has a specific role in the learning process.

The input layer is responsible for receiving the initial data. In image processing applications, the input may consist of pixel values from an image. The input layer itself does not perform computations; instead, it passes the information to the hidden layers for further processing.

FOR PAID USE ONLY

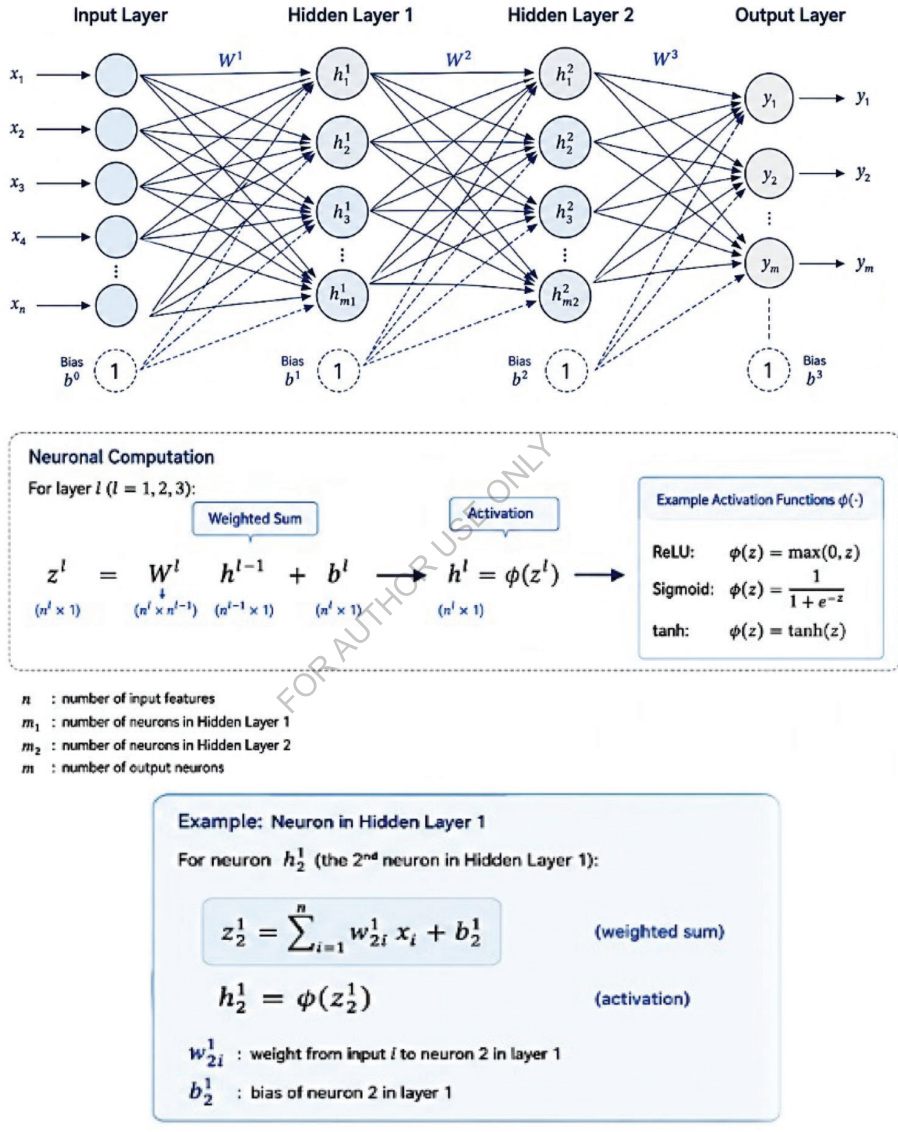
The hidden layers perform most of the computations inside the neural network. These layers extract patterns and relationships from the data by applying weights, biases, and activation functions. Depending on the complexity of the problem, a neural network may contain one or multiple hidden layers. As the number of hidden layers increases, the model becomes capable of learning more complex patterns.

The output layer produces the final prediction of the network. In classification problems, the output layer determines the predicted class label. For example, in an image classification task, the output layer may predict whether an image contains a cat, dog, car, or another object category.

Neural networks learn by adjusting weights during training using optimization algorithms. Initially, the weights are assigned random values. During the learning process, the network compares predicted outputs with actual outputs and calculates the error. This error is then minimized using optimization techniques such as gradient descent and backpropagation.

Although traditional neural networks perform well on structured data, they become inefficient when working with high-dimensional image data. Images contain thousands of pixel values, and fully connecting every neuron to all pixels results in extremely large computational requirements. This limitation led

to the development of Convolutional Neural Networks, which are specifically designed to process image data more efficiently.



FOR AUTHOR USE ONLY

Figure 2.1. Structure of a Traditional Neural Network.

Where:

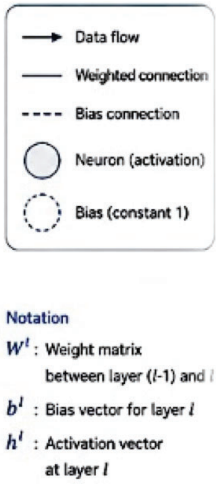


Table 2.1. Components of a Neural Network.

Component	Function
Input Layer	Receives the initial data
Hidden Layer	Extracts patterns and performs computations
Output Layer	Produces final predictions
Weights	Control the importance of connections
Bias	Helps shift activation values
Activation Function	Introduces non-linearity into the model

FOR AUTHOR USE ONLY

While traditional neural networks can learn patterns from data, they are not optimized for image processing tasks. This challenge led to the development of Convolutional Neural Networks (CNNs), which can efficiently analyze visual information using specialized layers and operations.

2.2. Introduction to Convolutional Neural Networks (CNN)

Convolutional Neural Networks (CNNs) are a specialized type of deep learning model designed mainly for processing image data. CNNs are widely used in computer vision tasks because they can automatically identify important visual patterns such as edges, shapes, textures, and objects directly from images.

Unlike traditional neural networks, CNNs do not connect every neuron to all input values. Instead, CNNs process small regions of an image at a time using mathematical filters called kernels. This reduces the number of computations and allows the model to focus on important local features within the image.

One of the major advantages of CNN is automatic feature extraction. In traditional machine learning methods, feature extraction is usually performed manually. However, CNN models learn these features automatically during training. Early layers often detect simple features such as lines and edges, while deeper layers identify more complex structures and object patterns.

FOR AUTHOR USE ONLY

CNN models are composed of several important layers that work together during image processing. The convolution layer extracts important features from the image, the pooling layer reduces the size of feature maps, and fully connected layers perform the final classification task. These layers help the model learn visual information efficiently while reducing computational complexity.

Another important characteristic of CNN is parameter sharing. The same filter is applied across different regions of the image, allowing the network to detect similar patterns regardless of their location. This improves both efficiency and learning capability compared to fully connected neural networks.

Due to these advantages, CNNs are commonly applied in many real-world applications. In healthcare, CNN models assist in medical image diagnosis. In security systems, CNNs are used for facial recognition and object detection. Similarly, autonomous vehicles use CNN models to recognize roads, traffic signs, pedestrians, and surrounding objects.

The basic workflow of a CNN generally begins with an input image. The image passes through convolution and pooling layers where features are

extracted and reduced gradually. Finally, the processed information is passed to fully connected layers that generate the final prediction or classification result.

Table 2.2. Traditional Neural Network vs CNN.

Feature	Traditional Neural Network	CNN
Input Processing	Uses all input values directly	Processes local image regions
Feature Extraction	Manual	Automatic
Computational Cost	High for image data	More efficient
Main Application	Structured data	Image and visual data
Parameter Sharing	No	Yes

FOR AUTHOR USE ONLY

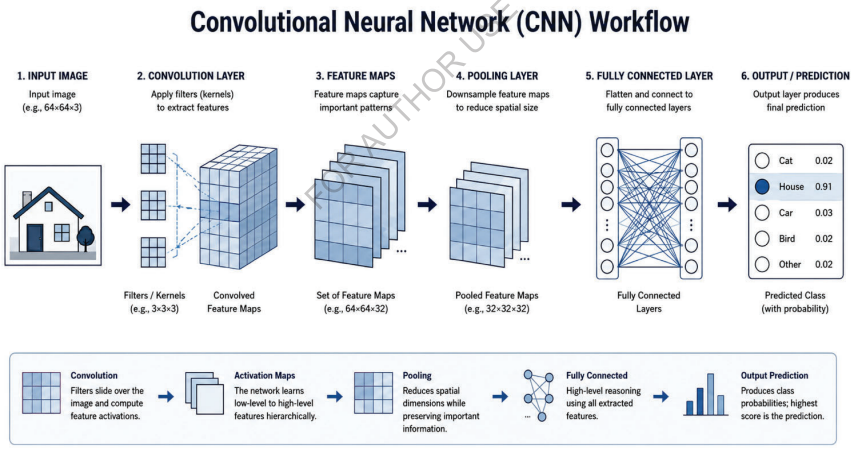


Figure 2.2. CNN Workflow Diagram.

The core operation that allows CNN models to extract important visual patterns from images is known as convolution. Understanding this operation is essential for understanding how CNN processes image data efficiently.

2.3. CNN vs Multi-Layer Perceptron (MLP)

Multi-Layer Perceptrons (MLPs) are traditional fully connected neural networks where every neuron is connected to all neurons in the next layer. Although MLP models perform well on structured numerical data, they are less efficient for image processing tasks because images contain a very large number of input pixels.

In MLP models, flattening image data into one-dimensional vectors causes loss of important spatial information such as edges, textures, and local relationships between pixels. In contrast, CNN models preserve spatial structure by processing small image regions through convolution operations.

CNNs also require fewer trainable parameters due to parameter sharing and local connectivity. This makes CNN models computationally more efficient and better suited for image classification and computer vision tasks compared to traditional MLP architectures.

Table 2.3. Comparison Between MLP and CNN.

Feature	MLP	CNN
Input Handling	Flattened input vectors	Processes image regions
Spatial Information	Mostly lost	Preserved
Parameter Count	High	Lower due to shared filters
Feature Extraction	Manual or limited	Automatic
Best Use Case	Structured numerical data	Image and visual data

FOR AUTHOR USE ONLY

2.4. Convolution Operation

The convolution operation is one of the most important processes in a Convolutional Neural Network. It is responsible for extracting useful features from an input image by applying small filters called kernels across the image. These filters help the model identify important visual patterns such as edges, lines, textures, and shapes.

A kernel is usually a small matrix, such as 3×3 or 5×5, containing numerical values. During convolution, the kernel slides across the image from left to right and top to bottom. At each position, the corresponding pixel values are

multiplied by the kernel values, and the results are added together to produce a single output value. This process generates a new matrix known as a feature map. Different kernels detect different features within the image. For example, one kernel may detect horizontal edges, while another may identify vertical patterns or textures.

The movement of the kernel across the image is controlled by a parameter called stride. A small stride allows the filter to move slowly across the image and capture more details, while a larger stride reduces the output size and computational cost.

Another important concept in convolution is padding. Padding adds extra pixels around the border of the image before convolution is applied. This helps preserve image dimensions and prevents loss of important edge information during processing.

2.4.1. Padding Types in CNN

Padding is commonly used in CNN models to control the size of the output feature map after convolution. Without padding, the dimensions of the image decrease after every convolution operation, which may lead to loss of important edge information in deeper layers.

FOR AUTHOR USE ONLY

The two most common padding methods are valid padding and same padding.

In valid padding, no extra pixels are added around the image border. As a result, the output feature map becomes smaller after convolution. This method reduces computational cost but may remove information near the edges of the image.

In same padding, additional zero-valued pixels are added around the image border before convolution is applied. This allows the output feature map to maintain approximately the same dimensions as the input image. Same padding is widely used in modern CNN architectures because it helps preserve spatial information.

The choice of padding depends on the CNN architecture and the requirements of the application. In many image classification tasks, same padding is preferred because it allows deeper networks to retain more image details throughout the learning process.

Table 2.4. Common Padding Types in CNN.

Padding Type	Description	Effect on Output Size
Valid Padding	No extra border pixels added	Output becomes smaller
Same Padding	Zero padding added around image	Output size remains similar

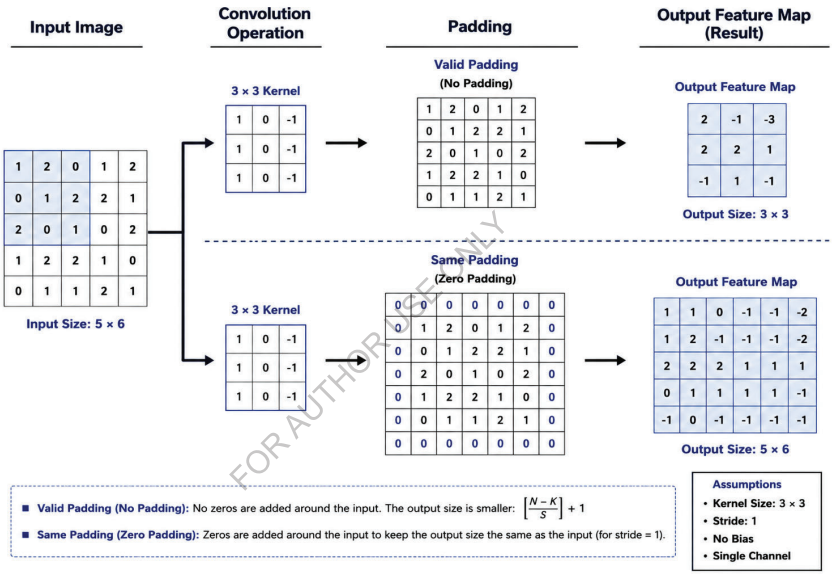


Figure 2.3. Valid Padding vs Same Padding in CNN.

One of the major advantages of convolution is that the same filter is reused across different regions of the image. This process, known as parameter sharing, reduces the number of trainable parameters and improves computational efficiency compared to fully connected neural networks.

The convolution operation allows CNN models to gradually learn hierarchical visual features. Early convolution layers often detect simple features such as edges and corners, while deeper layers identify more complex structures and object patterns.

Table 2.5. Important Parameters in Convolution.

Parameter	Description
Kernel/Filter	Small matrix used to extract features
Stride	Controls movement of the filter
Padding	Adds extra border pixels to the image
Feature Map	Output generated after convolution
Input Image	Original image provided to the CNN

The basic convolution operation can be represented as:

$$S(i,j) = \sum_{m=0}^{k_h-1} \sum_{n=0}^{k_w-1} I(i+m, j+n) \cdot K(m,n) + b, \qquad \textit{(Eq.2.1)}$$

Where:

- Irepresents the input image or input feature map
- Krepresents the convolution kernel/filter
- $k_h$and $k_w$represent the height and width of the kernel
- $b$represents the bias term
- $S(i,j)$represents the output value at position (i, j)in the feature map

FOR AUTHOR USE ONLY

The output size after convolution can be calculated as:

$$\textit{Output Size} = \frac{N-F+2P}{S} + 1, \qquad \textit{(Eq.2.2)}$$

Where:

- N= input size
- F= filter/kernel size
- P= padding
- S= stride

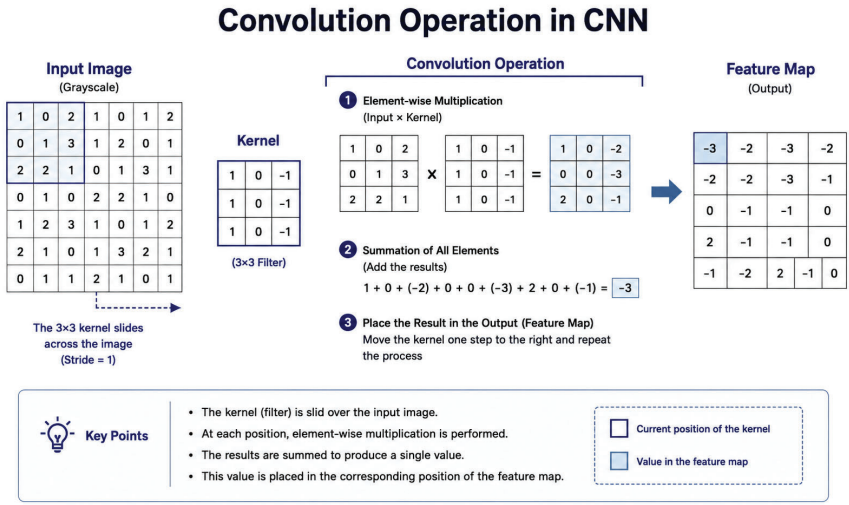


Figure 2.4. Convolution Operation Using a 3×3 Kernel.

FOR AUTHOR USE ONLY

```python
import cv2
import numpy as np
from matplotlib import pyplot as plt

image = cv2.imread('image.jpg', 0)

kernel = np.array([[1, 0, -1],
                   [1, 0, -1],
                   [1, 0, -1]])

feature_map = cv2.filter2D(image, -1, kernel)

plt.imshow(feature_map, cmap='gray')
plt.title("Feature Map")
plt.show()
```

Code Snippet 2.1. Applying Convolution using Python.

The output generated after applying convolution is known as a feature map. These feature maps contain important visual information that helps the CNN model recognize patterns within images.

2.4.2. Feature Maps

Feature maps are the outputs generated after applying convolution filters to an input image. These maps contain the important visual patterns

detected by the filters during the convolution process. Each feature map highlights a specific feature such as edges, textures, corners, or shapes within the image.

Different filters produce different feature maps. For example, one filter may detect vertical edges while another identifies horizontal patterns or texture details. As multiple filters are applied to the same image, several feature maps are generated, allowing the CNN model to learn different visual characteristics simultaneously.

In the early layers of a CNN, feature maps usually capture simple features such as lines and edges. As the network becomes deeper, the feature maps represent more complex structures and object-related patterns. This layered learning process helps CNN models recognize objects more accurately.

Feature maps also reduce the need for manual feature extraction. Instead of defining features manually, CNN models automatically learn useful visual patterns during training. This is one of the major reasons CNN performs effectively in image classification and computer vision tasks.

FOR AUTHOR USE ONLY

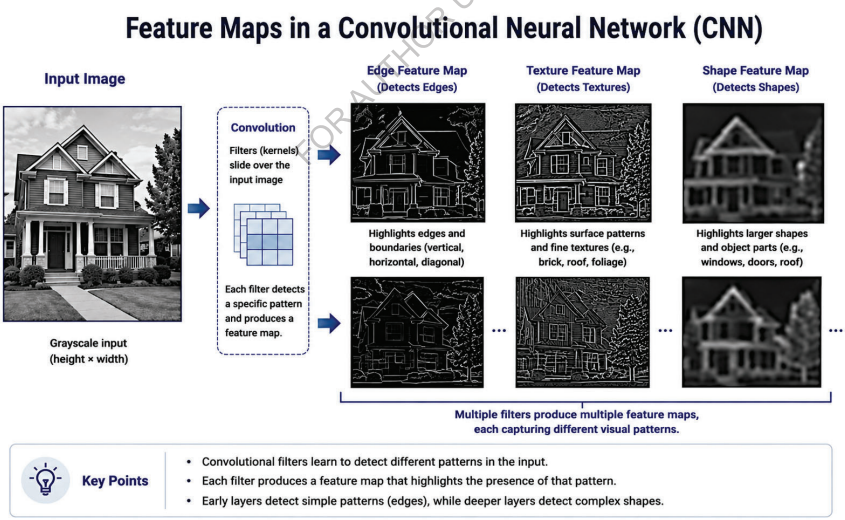


Figure 2.5. Example of Feature Maps Generated by CNN Filters.

```
plt.imshow(feature_maps[0, :, :, 0], cmap='gray')
plt.title("Feature Map")
plt.show()
```

Code Snippet 2.2. Visualizing Feature Maps.

After feature extraction, CNN models use pooling layers to reduce the size of feature maps while preserving important visual information.

2.5. Pooling Layer

The pooling layer is used in Convolutional Neural Networks to reduce the size of feature maps while keeping the most important information. This process helps decrease computational complexity and improves the efficiency of the CNN model.

Pooling works by selecting important values from small regions of the feature map. The most used type is max pooling, where the maximum value from each region is selected. Another type is average pooling, which calculates the average value of each region.

By reducing the dimensions of feature maps, pooling helps the model focus on the most significant visual features while reducing unnecessary details and noise. It also helps prevent overfitting by simplifying the learned representations.

FOR AUTHOR USE ONLY

Pooling layers are usually applied after convolution layers. As the image data moves deeper through the network, the feature maps become smaller while the important visual information is preserved.

Table 2.6. Types of Pooling in CNN.

Pooling Type	Description
Max Pooling	Selects the maximum value from a region
Average Pooling	Calculates the average value of a region
Global Max Pooling	Selects the maximum value from the entire feature map
Global Average Pooling	Computes the average value of the entire feature map

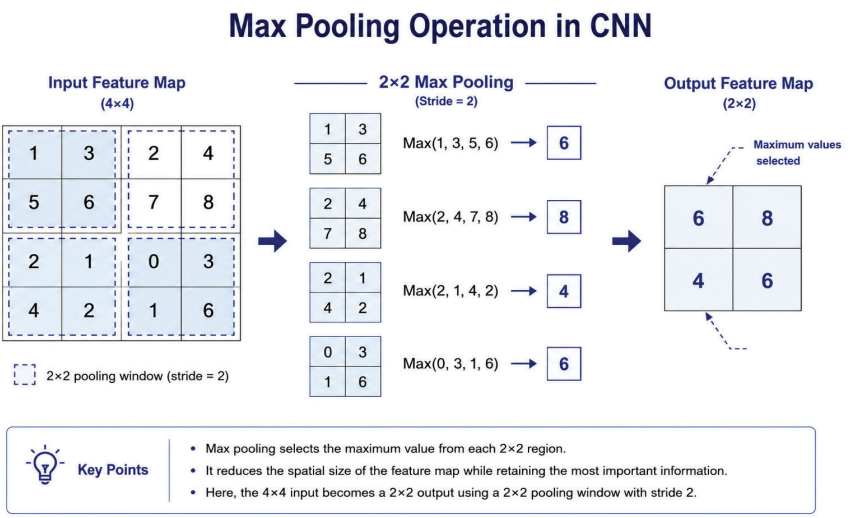


Figure 2.6. Max Pooling Operation in CNN.

```
from tensorflow.keras.layers import MaxPooling2D

pooling = MaxPooling2D(pool_size=(2,2))
```

Code Snippet 2.3. Applying Max Pooling.

FOR AUTHOR USE ONLY

2.6. CNN Architecture Flow

A Convolutional Neural Network processes image data through a sequence of specialized layers. Each layer performs a specific task that helps the network gradually learn important visual patterns from the input image. A typical CNN architecture consists of convolution layers, activation functions, pooling layers, fully connected layers, and an output layer.

The process begins with the input image, where pixel values are passed into the convolution layer. The convolution layer applies filters to detect important features such as edges, textures, and shapes. The resulting outputs are called feature maps.

After convolution, an activation function is applied to introduce non-linearity into the network. One of the most used activation functions in CNN models is the Rectified Linear Unit (ReLU). ReLU helps the network learn complex

patterns efficiently and reduces the vanishing gradient problem that commonly occurs in deep neural networks.

The ReLU activation function is mathematically represented as:

$$f(x) = \max(0, x), \qquad \textit{(Eq.2.3)}$$

One important challenge in deep neural networks is the vanishing gradient problem. During backpropagation, gradients may become extremely small as they pass through multiple layers. As a result, earlier layers learn very slowly or may stop learning completely. Traditional activation functions such as sigmoid and tanh are more affected by this problem because their gradients become very small for large input values. ReLU helps reduce this issue because it maintains stronger gradients for positive input values, allowing CNN models to train faster and more efficiently.

This function replaces all negative values with zero while keeping positive values unchanged. As a result, the model becomes computationally efficient and learns faster during training.

After activation, pooling layers reduce the size of feature maps while preserving important visual information. This helps decrease computational complexity and reduces overfitting.

FOR AUTHOR USE ONLY

The extracted features are then passed into fully connected layers. These layers combine the learned features and perform the final classification task. In image classification problems, the output layer often uses the Softmax activation function to generate probability values for different classes.

A standard CNN architecture generally follows the sequence:

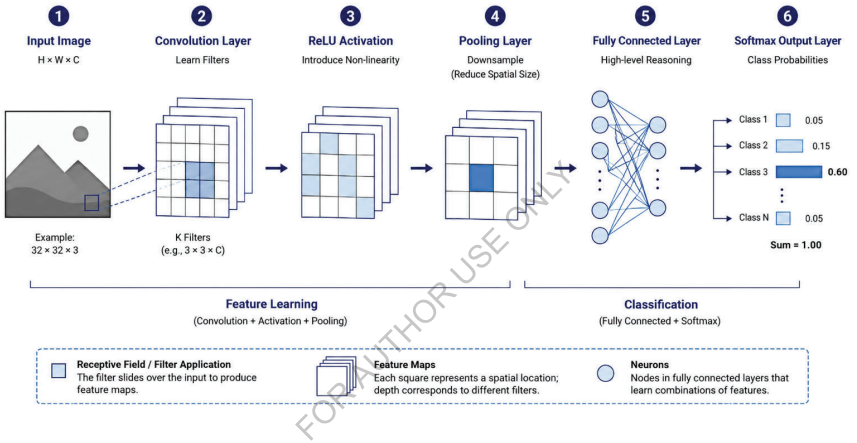
Input Image → Convolution → ReLU → Pooling → Fully Connected Layer → Softmax Output

This layered architecture allows CNN models to automatically learn hierarchical image features and perform highly accurate image classification.

Table 2.7. Main Layers in CNN Architecture.

Layer	Function
Convolution Layer	Extracts image features

ReLU Activation	Introduces non-linearity
Pooling Layer	Reduces feature map dimensions
Fully Connected Layer	Performs classification
Softmax Layer	Produces class probabilities



FOR AUTHOR USE ONLY

Figure 2.7. Basic CNN Architecture Flow.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D
from tensorflow.keras.layers import Flatten, Dense

model = Sequential([
    Conv2D(32, (3,3), activation='relu',
           input_shape=(28,28,1)),
    MaxPooling2D((2,2)),
    Flatten(),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')
])
```

Code Snippet 2.4. Basic CNN Architecture using Keras.

After defining the CNN architecture, the model must be trained using labeled data to learn meaningful patterns and improve prediction accuracy.

2.7. CNN Training Process

After defining the CNN architecture, the model must be trained using labeled image data. During training, the CNN gradually learns important visual patterns by adjusting its weights and biases to minimize prediction errors.

The training process begins with forward propagation. In this step, the input image passes through convolution, activation, pooling, and fully connected layers to generate a prediction. The predicted output is then compared with the actual label using a loss function.

A loss function measures how far the predicted output is from the correct output. In image classification tasks, categorical cross-entropy is commonly used as the loss function because it works efficiently with multi-class classification problems.

The categorical cross-entropy loss function is represented as:

$$L = -\sum_{i=1}^{C} y_i \, log \, g(\hat{y}_i) \qquad \textit{(Eq.2.4)}$$

Where:

- $y_i$represents the true label
- $\hat{y}_i$represents the predicted probability
- Crepresents the number of classes

FOR AUTHOR USE ONLY

After calculating the loss, the CNN updates its parameters using backpropagation and optimization algorithms. Backpropagation calculates gradients for each parameter, while the optimizer adjusts the weights to reduce prediction error.

One of the most widely used optimizers in CNN models is the Adam optimizer. Adam combines the advantages of momentum and adaptive learning rates, allowing faster and more stable training.

The training process is repeated for multiple epochs. An epoch represents one complete pass of the training dataset through the CNN model. During training, the model gradually improves its ability to recognize image patterns and classify images correctly.

Table 2.8. Important Components in CNN Training.

Component	Function
Forward Propagation	Generates predictions from input data
Loss Function	Measures prediction error
Backpropagation	Computes gradients for learning
Optimizer	Updates model parameters
Epoch	One full training cycle

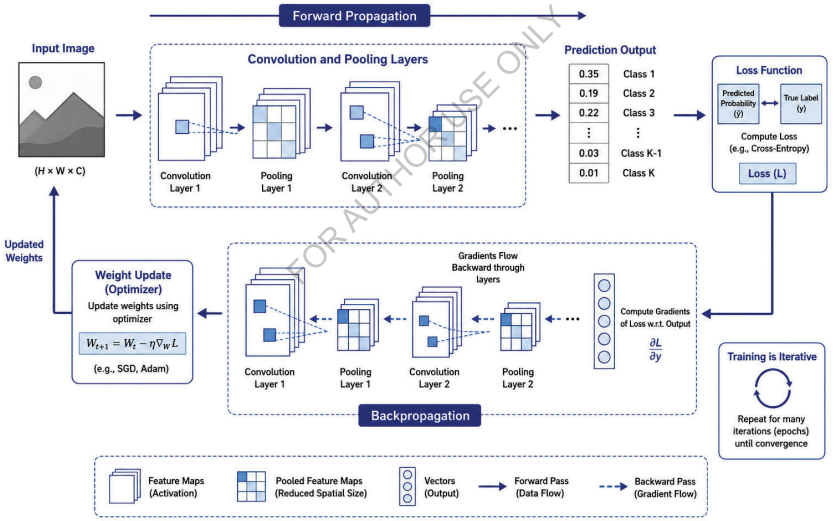


Figure 2.8. CNN Training Pipeline.

FOR AUTHOR USE ONLY

```
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(X_train, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_split=0.2)
```

Code Snippet 2.5. Compiling and training a CNN Model.

After understanding the CNN training process, a practical implementation using a real image dataset can further demonstrate how CNN models perform image classification tasks.

2.8. MNIST Handwritten Digit Classification using CNN

To demonstrate the practical application of Convolutional Neural Networks, the MNIST handwritten digit dataset is commonly used in deep learning projects. The MNIST dataset contains grayscale images of handwritten digits from 0 to 9. Each image has a size of 28×28 pixels and is labeled according to the digit it represents.

FOR AUTHOR USE ONLY

The dataset is divided into training and testing sets. The training set is used to train the CNN model, while the testing set evaluates the model's performance on unseen data. Before training, the pixel values are normalized to improve learning efficiency.

In this implementation, the CNN model consists of convolution layers, ReLU activation functions, max pooling layers, flattening, and fully connected layers. The final output layer uses the Softmax activation function to classify images into 10 different digit categories.

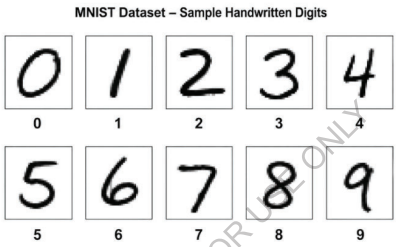
During training, the model learns important visual features such as curves, edges, and digit shapes from the handwritten images. As the number of training epochs increases, the prediction accuracy of the model gradually improves.

After training, the CNN model is evaluated using the test dataset. Performance metrics such as accuracy and loss are commonly used to measure how well the model classifies handwritten digits.

CNN models generally achieve very high accuracy on the MNIST dataset because convolution and pooling operations allow the network to efficiently extract important visual features from images.

Table 2.9. MNIST Dataset Information.

Property	Description
Dataset Name	MNIST Handwritten Digits
Number of Classes	10 (Digits 0–9)
Image Size	28 × 28 pixels
Training Images	60,000
Testing Images	10,000



FOR AUTHOR USE ONLY

Figure 2.9. Sample Images from the MNIST Dataset.

2.9. Performance Evaluation

After training the CNN model, its performance is evaluated using the testing dataset. Performance metrics help determine how accurately the model classifies unseen images and how effectively it has learned meaningful visual patterns during training.

One of the most used metrics is accuracy, which measures the percentage of correctly classified images. High accuracy indicates that the CNN model can successfully recognize handwritten digits from the MNIST dataset.

Another important metric is loss, which measures the difference between predicted outputs and actual labels. During training, the goal of the optimizer is to minimize the loss value. A lower loss generally indicates better model performance and improved learning.

In addition to training accuracy and loss, validation accuracy and validation loss are also monitored during training. Validation metrics help evaluate how well the model generalizes to unseen data and assist in detecting overfitting.

A confusion matrix can also be used to analyze classification performance in more detail. The confusion matrix shows how many images from each digit class are correctly or incorrectly classified. This helps identify which classes are more difficult for the CNN model to recognize.

For image classification tasks such as MNIST, CNN models typically achieve very high accuracy because convolution and pooling operations efficiently capture important spatial features from handwritten digit images.

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

(X_train, y_train), (X_test, y_test) = mnist.load_data()

X_train = X_train.reshape((60000, 28, 28, 1)) / 255.0
X_test = X_test.reshape((10000, 28, 28, 1)) / 255.0

y_train = to_categorical(y_train)
y_test = to_categorical(y_test)

model.fit(X_train, y_train,
          epochs=10,
          batch_size=32,
          validation_split=0.2)

test_loss, test_acc = model.evaluate(X_test, y_test)

print("Test Accuracy:", test_acc)
```

FOR AUTHOR USE ONLY

Code Snippet 2.6. CNN Implementation using the MNIST Dataset.

```
test_loss, test_accuracy = model.evaluate(X_test, y_test)

print("Test Loss:", test_loss)
print("Test Accuracy:", test_accuracy)
print("Test Accuracy Percentage:", test_accuracy * 100)
```

```
313/313 ━━━━━━━━━━━━━━━━━━━━ 2s 6ms/step - accuracy: 0.9843 - loss: 0.0691
Test Loss: 0.06908269226551056
Test Accuracy: 0.9843000173568726
Test Accuracy Percentage: 98.43000173568726
```

Code Snippet 2.7. CNN Training and Evaluation Output on the MNIST Dataset.

The CNN model achieved a test accuracy of 98.43% on the MNIST dataset after 10 training epochs, demonstrating the effectiveness of convolutional neural networks in handwritten digit classification tasks.

Chapter 3 — Recurrent Neural Networks (RNN)

Introduction and Literature Review

In today's world, machine learning has become an integral part of modern decision-making systems, powering everything from natural language comprehension to continuous engineering forecasts. While classical algorithms excel at evaluating isolated, static data points, many real-world phenomena are fundamentally tied to sequential order and chronological context. Two critical domains where sequential dependencies are evident are natural language processing (NLP) and environmental sensor analytics. In NLP, sentence sentiment cannot be deciphered as a disconnected "bag of words"; instead, syntax, negation, and context rely entirely on word order. Similarly, environmental parameters such as hourly gas concentrations or temperature levels are governed by physical thermodynamics over time, meaning that future states are deeply contingent upon historical intervals.

To resolve these structural complexities, early researchers explored architectures capable of preserving sequential memory. Rumelhart et al. (1986) established the foundational Recurrent Neural Network (RNN) framework using Backpropagation Through Time (BPTT), which allows a model to map sequential dependencies by circulating historical information through an internal hidden state. Despite establishing the theoretical framework for sequence learning, classical RNNs suffer from severe limitations. As detailed by Lipton et al. (2015) in a critical review of sequence processing, standard recurrent units struggle with the vanishing and exploding gradient problem, which exponentially diminishes or blows up error signals over extended sequences. This vulnerability renders basic RNNs practically incapable of tracking long-range dependencies, causing models to drop critical contextual cues in long text reviews or decades of hourly continuous data.

To overcome this fundamental limitation, Hochreiter and Schmidhuber (1997) introduced the Long Short-Term Memory (LSTM) network. The LSTM architecture replaces simple recurrent nodes with intricate gating cells containing an explicit cell state managed by input, forget, and output gates that regulate context retention. Seeking to optimize these recurrent workflows, Cho et al. (2014) introduced the Gated Recurrent Unit (GRU), which streamlined the gating pipeline by compounding parameters into integrated update and reset vectors. Building upon these deep learning frameworks, Lim and Zohren (2021) demonstrated that gated sequence models could be effectively applied

FOR AUTHOR USE ONLY

to complex, multi-variable time-series forecasting by constructing sliding temporal windows, capturing non-linear trends that traditional statistical methods overlook.

This project directly builds upon these sequential modeling paradigms using Python-based machine learning and deep learning tools (Pedregosa et al., 2011; Chollet, 2015). We model two separate experimental setups trained across two separate data environments: a text classification setup using the IMDb Movie Reviews dataset (Maas et al., 2011) and a multi-variable continuous forecasting pipeline using the Air Quality UCI dataset. Restructuring both problems into a supervised learning format (X, *y*), this study implements baseline linear/statistical methods and advances to deep Gated LSTMs. The primary goal is to evaluate, optimize, and compare how recurrent architectures learn sequential patterns over time, analyzing their performance metrics through classification configurations, loss trajectories, and error tracking metrics.

FOR AI TRAINING USE ONLY

3.1. Recurrent Neural Networks for Sequence Processing

In supervised machine learning, traditional feedforward neural networks operate under the assumption that all input samples and features are entirely independent of one another. While this assumption is true for tabular data structures, it fails when applied to data governed by chronological order, syntax, or temporal continuity. A Recurrent Neural Network (RNN) resolves this structural limitation by introducing cyclical feedback loops within its hidden layers, enabling information to persist across successive steps of a sequence. Rather than map an isolated input tensor directly to an output vector, an RNN processes the current sequential element while simultaneously consuming its own internal hidden state from the previous step, continuously passing an updated memory vector forward. This mathematical architecture allows the network to be conceptually unfolded across time, transforming a single looping computational node into an ordered, interconnected chain of chronological operations.

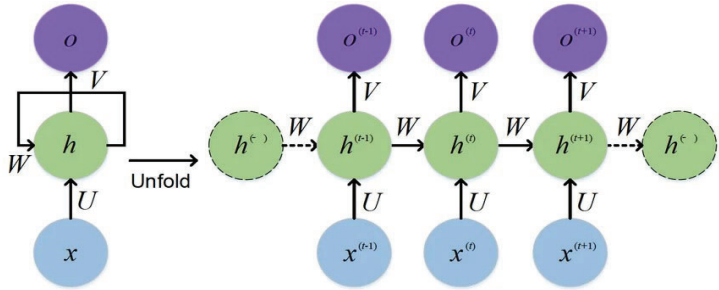


Figure 3.1. Structural layout of a standard Recurrent Neural Network unfolded across time.

In practical applications, each step in an unfolded RNN operates as a precise sequential checkpoint. For sequence modeling tasks such as natural language processing and sentiment classification, the network sequentially inputs an ordered stream of text tokens. As it loops through the text, it continuously updates its hidden state vector, allowing the model to carry early contextual cues through the narrative to compute a final sequence-to-one category prediction.

For continuous time-series applications, the network loops through uniform chronological intervals of numerical data, tracking multi-variable adjustments over a defined historical horizon. This cyclical processing matches the underlying physics of time-dependent phenomena, making the RNN framework the foundational architecture for modern sequence-based decision support systems.

FOR AUTHOR USE ONLY

3.2. The Vanishing and Exploding Gradient Problem

Despite the theoretical capacity of standard RNNs to retain long-term memory via recurrent feedback loops, their practical performance is limited by the vanishing and exploding gradient problem. This issue occurs during Backpropagation Through Time (BPTT). When calculating gradients for long sequences, the gradients must be propagated backward through earlier timesteps. This process involves repeated multiplication of recurrent weight matrices across multiple timesteps.

If the eigenvalues of the recurrent weight matrix are less than unity, the repeated multiplication causes gradients to shrink toward zero as the sequence length increases. This phenomenon is known as the vanishing gradient, and it reduces learning for earlier timesteps. Conversely, if the eigenvalues exceed

unity, the gradients grow exponentially, resulting in exploding gradients that can destabilize training.

In sentiment classification, a vanishing gradient causes the network to completely drop long-range contextual information introduced at the start of a long review. In time-series forecasting, it prevents the model from capturing long-term temporal patterns. As a result, standard RNNs may perform poorly on long sequences despite performing adequately on shorter inputs.

3.3. Gated Architectural Mechanisms

To address these limitations, advanced architectures replace basic looping nodes with gated cell structures. Gated mechanisms regulate information flow across sequences by utilizing internal operations that retain, update, or discard information.

These gates utilize element-wise multiplications combined with smooth non-linear activation functions to establish more stable gradient flow. The two primary gated variations used to address long-range dependencies are the Long Short-Term Memory (LSTM) network and the Gated Recurrent Unit (GRU).

FOR AUTHOR USE ONLY

3.4. Long Short-Term Memory (LSTM)

The Long Short-Term Memory (LSTM) network reduces the impact of the vanishing gradient problem by introducing a cell state pathway known as the **cell state** (C_t). The cell state passes information across timesteps with minimal non-linear interactions, allowing gradient signals to flow backward across hundreds of time steps with reduced gradient loss. The cell state is modified through three gating mechanisms: the forget gate, the input gate, and the output gate.

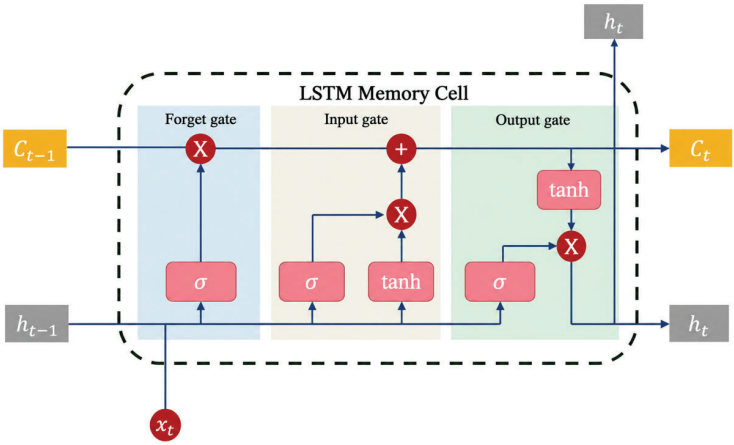


Figure 3.2. Internal mathematical gating structure of a Long Short-Term Memory cell.

The Forget Gate (f_t) determines which information should be removed from the cell state. It processes the previous hidden state (h_{t-1}) and the current input vector (x_t) through a logistic sigmoid activation function, producing values between 0 (total deletion) and 1 (complete retention).

The Input Gate (i_t) controls what new information is added to the cell state. It combines a sigmoid selection layer with a hyperbolic tangent (tanh) candidate layer ($\hat{C}_t$) to generate updated memory information.

The Output Gate (o_t) determines the next hidden state (h_t), controlling which information is passed to the subsequent layer or terminal output.

The core LSTM cell state update is governed by the following system of equations:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t, \qquad \textit{(Eq.3.1)}$$

Equation 3.1. LSTM cell state modification function.

Where:

C_t: The updated cell state vector at the current time step. This represents the network's long-term memory state after processing the current token or time step.

FOR AUTHOR USE ONLY

f_t: The forget gate activation vector at the current time step. It contains values bounded between 0 and 1 that determine what fraction of historical memory should be erased.

C_{t-1}: The historical cell state vector from the immediate previous time step. It represents the previous memory state.

i_t: The input gate activation vector at the current time step. It contains values between 0 and 1 that regulate how much new information should be allowed into the cell state.

$\tilde{C}_t$: The candidate cell state vector. This represents new memory values generated from the current input and the previous hidden state using a hyperbolic tangent (tanh) activation function.

$\odot$: The Hadamard product operator. This indicates that element-wise vector multiplication is performed.

+: Vector addition. This operation combines the filtered past memory with the newly gated incoming information to construct the updated cell state.

t: The current timestep in the sequence.

t-1: The previous timestep

FOR AUTHOR USE ONLY

By utilizing this cell state mechanism, the LSTM helps preserve long-range dependencies, improving learning across long sequences.

3.5. Gated Recurrent Unit (GRU)

The Gated Recurrent Unit (GRU) reduces the complexity of the LSTM by merging the independent cell state and hidden state into a single hidden state vector (h_t). This architectural simplification removes the dedicated cell state highway, using two integrated operations: the update gate and the reset gate.

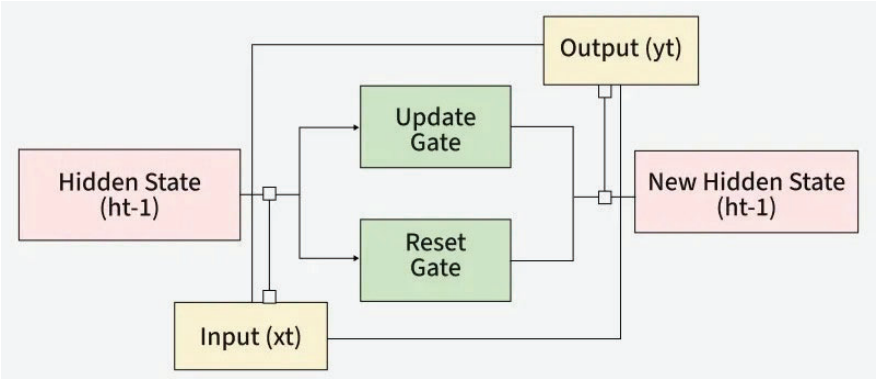


Figure 3.3. Internal mathematical gating structure of a Gated Recurrent Unit cell.

The Update Gate (z_t) combines the functions of the forget and input gates, controlling how much previous information is retained from the past alongside how much new candidate information to write into the state.

The Reset Gate (r_t) determines how much of the past hidden state the network should ignore when calculating the new candidate step, allowing the model to remove unnecessary information

FOR AUTHOR USE ONLY

The mathematical formulation for the terminal GRU hidden state configuration is given by the formula:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \quad , \qquad \textit{(Eq.3.2)}$$

Equation 3.2. GRU hidden state modification function.

Where:

h_t: The final updated hidden state vector at the current time step. In a GRU, this single vector serves as both the internal memory state and the visible output passed to the next layer.

z_t: The update gate activation vector at the current time step. It functions as a combined forget and input gate, explicitly balancing how much old context to drop versus how much new content to accept.

h_{t-1}: The hidden state vector from the immediate previous time step, carrying the network's previous hidden state information.

$\tilde{h}_t$: The current candidate hidden state vector. This represents the updated hidden representations built from the current step's input combined with the filtered previous state.

The simpler gating structure reduces the number of parameters, resulting in faster training and lower computational cost while maintaining effective sequence modeling performance.

Table 3.1. Structural Comparison of Gated Recurrent Neural Network Architectures.

Aspect	Long Short-Term Memory (LSTM)	Gated Recurrent Unit (GRU)
State Vector Topology	Dual vectors: Separated Cell State (C_t) and Hidden State (h_t)	Single vector: Unified Hidden State (h_t)
Gating Layout	Three independent gates: Forget (f_t), Input (i_t), and Output (o_t)	Two integrated gates: Update (z_t) and Reset (r_t)
Computational Footprint	Higher parameter count; increased matrix computation load	Streamlined structure; optimized training throughput
Sensitivity to Long Dependencies	Highly robust over hyper-extended chronological contexts	Adaptive and effective on short-to-medium sequence lengths
Primary Project Use Case	Complex language modeling requiring deep hierarchical tracking	Agile temporal forecasting where speed and efficiency are prioritized

FOR AUTHOR USE ONLY

3.6. Foundational Sequence Modeling Concepts

In natural language processing pipelines, text data must be converted into numerical form before it can be processed by neural networks. Sequence modeling therefore involves transforming raw text into structured numerical representations while preserving word order. The preprocessing pipeline typically includes text cleaning, tokenization, vocabulary mapping, sequence padding, and embedding generation.

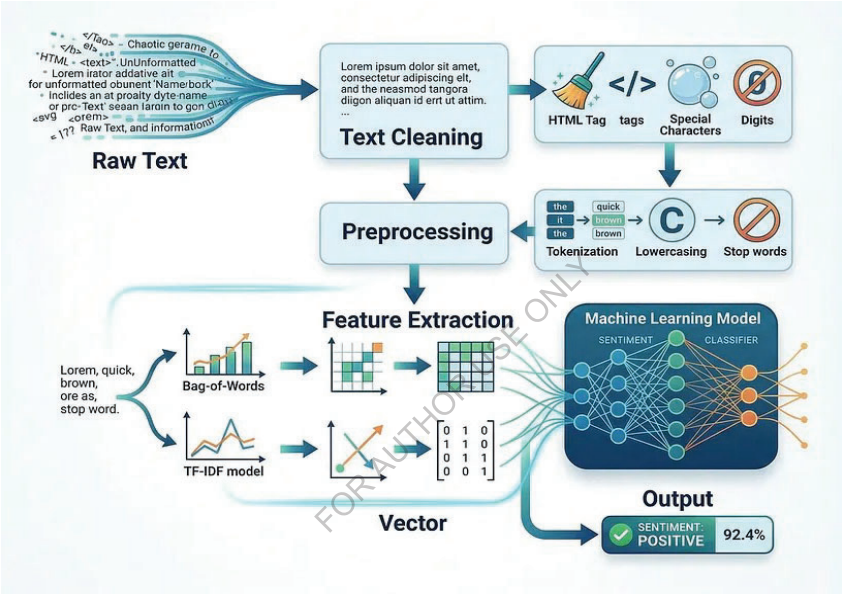


Figure 3.4. Natural language preprocessing pipeline.

The data preparation framework illustrated in Figure 3.4 transforms unrefined text corpora into optimized model inputs through an integrated execution pipeline:

- Raw Text Input: Raw text reviews are collected directly from the dataset. At this stage, the data may contain inconsistent formatting, symbols, or unnecessary elements that require preprocessing before model training.
- Text Cleaning Stage: Unnecessary elements such as HTML tags, special characters, and numerical values are removed from the text. This reduces noise and improves data consistency during training.

- Tokenization, Lowercasing, and Stopwords Filtering: The text is converted to lowercase and split into individual word tokens. Common stopwords such as “the”, “and”, and “is” may also be removed to reduce irrelevant information and improve feature quality.
- Feature Extraction (Vectorization): The processed text tokens are converted into numerical vectors so they can be used as inputs for the neural network model.
- Model Input Generation: The final vectorized sequences are used as inputs for training and testing the recurrent neural network.

The equation for embedding function is given by:

$$E : V \rightarrow \mathbb{R}^d , \qquad \textit{(Eq.3.3)}$$

Equation 3.3. Dense embedding vector mapping function.

Where,

E: The embedding function or transformation matrix. This mathematical operator maps discrete words into continuous vector representations.

V: The discrete vocabulary set. This represents the vocabulary containing all unique integer word indices extracted from the training corpus.

FOR AUTHOR USE ONLY

→: The mapping operator, indicating a mathematical transformation from a source domain into a target co-domain.

$\mathbb{R}^d$: A continuous, real-valued d-dimensional vector space. The variable d represents the hyperparameter choice for embedding dimensions (e.g., 32 or 64), defining the length of the continuous vector that tracks semantic relationships.

3.7. Time Series Forecasting Frameworks

Time series forecasting involves analyzing sequential numerical data recorded over time to predict future values. Recurrent neural networks can learn temporal patterns such as trends, seasonal behavior, and short-term dependencies from historical observations.

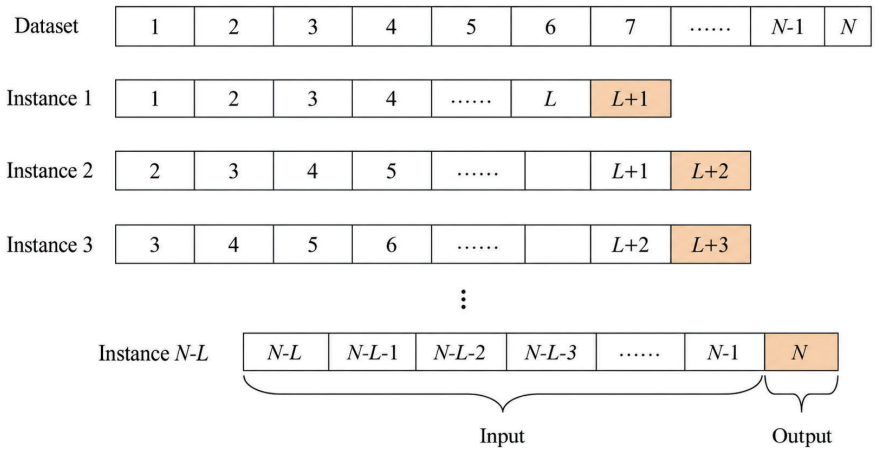


Figure 3.5. Continuous Time-Series vector conversion via sliding lookback windows.

FOR AUTHOR USE ONLY

To adapt a continuous time-series data for supervised machine learning, the data must be restructured using a Sliding Window Strategy. As detailed in the operational layout of the temporal partitioning data pipeline in Figure 3.5, a single continuous chronological data vector cannot be directly interpreted by a supervised model because it lacks independent input-output relationships. The sliding window framework explicitly overcomes this by segmenting the continuous timeline into a sequence of overlapping training rows, where a block of past observations is mapped directly to a future value.

The data sliding window mechanism illustrated in Figure 3.5 converts a continuous sequence of length *N* into pairs of supervised features and labels using the following visual steps:

- Sliding Window Strategy: A fixed size window of previous observations is used as the input sequence, while the following value becomes the prediction target.
- Feature Construction: The first *L* values in the window are used as input features *X*, representing historical observations.
- Target Generation: The value immediately following the input window is used as the target output *y*.

Because recurrent layers are highly sensitive to the scale of continuous numeric variables, Feature Scaling via Min-Max normalization or

Standardization is executed across the generated arrays to eliminate gradient anomalies and preserve optimization stability. Furthermore, sequential deep learning architectures require inputs to match a three-dimensional input structure: (Samples, Time Steps, Features)

This dimensional configuration ensures that the recurrent cell can loop through the "Time Steps" dimension (the exact sliding window length L) for every individual "Sample" row, processing one timestep at a time before emitting a continuous scalar forecast at the terminal layer.

$$X_t = [x_{t-W}, x_{t-W+1}, \ldots, x_{t-1}] \quad \longrightarrow \quad y_t = x_t \quad , \qquad \textit{(Eq.3.4)}$$

Equation 3.4. Supervised sliding lookback window transformation.

Where,

X_t: The supervised feature vector (or input feature matrix) constructed for training at chronological interval t.

x: The raw, single-variable continuous time-series data array (such as recorded hourly temperatures or gas sensor metrics).

W: The lookback window size hyperparameter. This integer defines the exact number of historical hours or intervals used as predictor features.

y_t: The supervised target scalar label associated with the feature matrix at step t.

x_t: The actual real-world value manifested at the current time step t, which serves as the target value the model is optimizing to forecast.

→: The transformation operator, indicating how a single time-series sequence is mapped into distinct pairs of input features and target labels for supervised machine learning models.

FOR AUTHOR USE ONLY

3.8. Hyperparameter and Optimization Control

To improve generalization and reduce overfitting, recurrent neural networks require appropriate hyperparameter tuning and regularization techniques. These controls help stabilize training and improve model performance on unseen data.

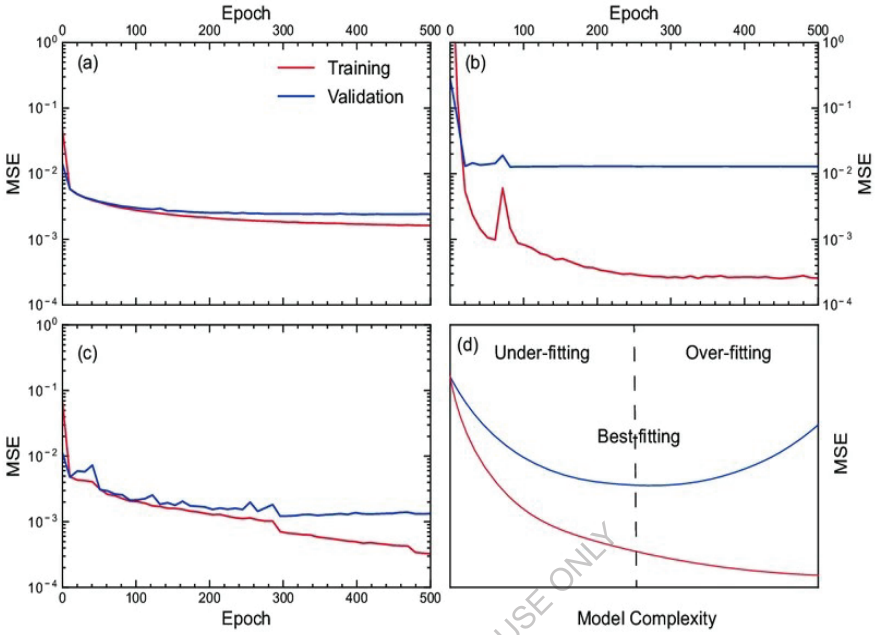


Figure 3.6. Training and validation convergence patterns.

FOR AUTHOR USE ONLY

Illustration of training loss (red curves) and validation loss (blue curves) over the number of epochs in (a) under-fitting, (b) over-fitting, and(c) the best-fit optimized model. In (d), we plot the loss as a function of model complexity, and show that when model complexity is too low, both training and validation losses are high, as shown in (a). On the other hand, when the model complexity is too high, we start to observe a huge gap between training and validation loss. An indication of an optimized model is when both the training and validation losses are low, and similar in value to each other, as shown by the dashed line in (d).

The number of hidden units affects the learning capacity of the network. Too few units may lead to underfitting, while too many may increase the risk of overfitting. To address this issue, dropout regularization is applied within the recurrent layers.

The functional distinction between these two regularizers is critical for sequence stability:

- Standard Dropout: Randomly disables a proportion of neuron connections during training to reduce over-reliance on specific features.

- Recurrent Dropout: Applies dropout to recurrent connections between timesteps, helping prevent the model from depending too heavily on specific sequential patterns.

The Adam optimizer is used to adapt learning rates during training based on gradient information. Compared to standard stochastic gradient descent, Adam generally provides faster and more stable convergence.

Early stopping monitors validation loss during training. Training is stopped once validation performance no longer improves, reducing unnecessary epochs and helping prevent overfitting.

3.9. Sentiment Analysis

The IMDB Movie Reviews dataset is a widely used benchmark for natural language processing and sentiment classification tasks. The dataset consists of 50,000 movie reviews sourced from the Internet Movie Database, equally divided into 25,000 training and 25,000 test samples. Each review is labelled as either positive (1) or negative (0), making this a binary classification problem. The dataset is preprocessed and made available through the Keras library, where each review is represented as a sequence of integers corresponding to word indices ranked by frequency. For this study, the vocabulary was restricted to the 20,000 most frequently occurring words to reduce noise introduced by rare or irrelevant terms.

FOR AUTHOR USE ONLY

3.9.1. Methodology and Dataset

All required libraries are imported at the outset. TensorFlow and Keras provide the deep learning framework, including the LSTM architecture, regularization layers, and training callbacks. NumPy is used for numerical operations, while Matplotlib and Seaborn are used for visualization. Scikit-learn provides evaluation utilities including the classification report and confusion matrix.

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

from tensorflow.keras.datasets import imdb
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (Embedding, LSTM, Dense,
                                     Dropout, Bidirectional,
                                     BatchNormalization)
from tensorflow.keras.callbacks import (EarlyStopping,
                                        ReduceLROnPlateau,
                                        ModelCheckpoint)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l2
from sklearn.metrics import classification_report,
confusion_matrix
```

Code Snippet 3.1. Imports and environment setup.

The IMDB dataset is loaded directly from the Keras library with the vocabulary restricted to the 20,000 most frequent words. The dataset is split into 25,000 training and 25,000 test samples. Since recurrent neural networks require fixed-length input sequences, all reviews are padded or truncated to a uniform length of 400 tokens. Reviews shorter than 400 tokens are padded with zeros at the end, while reviews exceeding this length are truncated from the end. The sequence length of 400 was selected to capture most of the review content while maintaining computational efficiency.

FOR AUTHOR USE ONLY

```
VOCAB_SIZE = 20000
MAX_LENGTH = 400
EMBEDDING_DIM = 128

(X_train, y_train), (X_test, y_test) = imdb.load_data(
    num_words=VOCAB_SIZE
)

X_train_pad = pad_sequences(X_train,
                            maxlen=MAX_LENGTH,
                            padding='post',
                            truncating='post')
X_test_pad = pad_sequences(X_test,
                           maxlen=MAX_LENGTH,
                           padding='post',
                           truncating='post')
```

Code Snippet 3.2. Data loading and preprocessing.

The model is constructed as a sequential architecture. An Embedding layer first maps each word index to a dense 128-dimensional vector representation, which the model refines during training. SpatialDropout1D with a rate of 0.2 is applied after the embedding layer, dropping entire embedding feature maps

rather than individual elements — a more effective regularization strategy for sequential text data compared to standard dropout.

```
model = Sequential([
    Embedding(input_dim=VOCAB_SIZE,
              output_dim=EMBEDDING_DIM,
              input_length=MAX_LENGTH),

    # SpatialDropout1D drops entire embedding dimensions
    # more effective than regular dropout for text
    SpatialDropout1D(0.2),

    # Single Bidirectional LSTM - simpler is better here
    Bidirectional(LSTM(128,
                       dropout=0.2,
                       recurrent_dropout=0.2,
                       return_sequences=True)),

    Bidirectional(LSTM(64,
                       dropout=0.2,
                       recurrent_dropout=0.2)),

    Dense(64, activation='relu'),
    Dropout(0.3),
    Dense(1, activation='sigmoid')
])

model.build(input_shape=(None, MAX_LENGTH))

model.summary()
```

FOR AUTHOR USE ONLY

Code Snippet 3.3. Model architecture.

The core of the model consists of two stacked Bidirectional LSTM layers. Bidirectional processing allows the model to capture contextual dependencies in both the forward and backward directions of the input sequence, which is particularly beneficial for sentiment analysis where the meaning of a word is often influenced by surrounding context. The first Bidirectional LSTM layer contains 128 units and returns full sequences for the subsequent layer. The second contains 64 units and produces a single context vector. Both layers apply input dropout and recurrent dropout of 0.2 to regularize the recurrent connections. A fully connected Dense layer with 64 units and ReLU activation is applied, followed by a Dropout layer with a rate of 0.3. The output layer consists of a single unit with sigmoid activation, producing a probability score between 0 and 1 for binary sentiment classification.

Model: "sequential_1"

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, 400, 128)	2,560,000
spatial_dropout1d_1 (SpatialDropout1D)	(None, 400, 128)	0
bidirectional_2 (Bidirectional)	(None, 400, 256)	263,168
bidirectional_3 (Bidirectional)	(None, 128)	164,352
dense_2 (Dense)	(None, 64)	8,256
dropout_1 (Dropout)	(None, 64)	0
dense_3 (Dense)	(None, 1)	65

Total params: 2,995,841 (11.43 MB)
Trainable params: 2,995,841 (11.43 MB)
Non-trainable params: 0 (0.00 B)

Figure 3.7. Model summary.

The model is compiled using the Adam optimizer with an initial learning rate of 0.001 and binary cross-entropy as the loss function, which is standard for binary classification tasks. Three callbacks are employed to regulate the training process. EarlyStopping monitors validation accuracy and halts training if no improvement is observed for three consecutive epochs, automatically restoring the weights from the best performing epoch. ReduceLROnPlateau monitors validation loss and reduces the learning rate by a factor of 0.5 if no improvement is observed for two consecutive epochs, with a minimum learning rate of 1×10^{-6}. ModelCheckpoint saves the model weights corresponding to the highest validation accuracy observed during training.

FOR AUTHOR USE ONLY

```
optimizer = Adam(learning_rate=0.001)
model.compile(optimizer=optimizer,
              loss='binary_crossentropy',
              metrics=['accuracy'])

early_stopping = EarlyStopping(
    monitor='val_accuracy',  # changed to accuracy not loss
    patience=3,
    restore_best_weights=True,
    verbose=1
)

reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
    patience=2,
    min_lr=1e-6,
    verbose=1
)

checkpoint = ModelCheckpoint(
    'best_model_imdb_v2.keras',
    monitor='val_accuracy',
    save_best_only=True,
    verbose=1
)
```

Code Snippet 3.4. Model compilation and callbacks.

FOR ACADEMIC USE ONLY

The model is trained for a maximum of 15 epochs with a batch size of 64. A validation split of 0.2 is applied, reserving 5,000 samples from the training set for validation at each epoch. Training was halted at epoch 8 by the EarlyStopping callback, with optimal weights restored from the best performing epoch. The reduced batch size of 64 relative to the initial architecture increases the frequency of weight updates per epoch, contributing to improved convergence.

```
history = model.fit(
    X_train_pad, y_train,
    epochs=15,
    batch_size=64,    # smaller batch size
    validation_split=0.2,
    callbacks=[early_stopping, reduce_lr, checkpoint],
    verbose=1
)
```

Code Snippet 3.5. Model training.

The trained model is evaluated on the held-out test set of 25,000 samples. Predictions are generated by applying a decision threshold of 0.5 to the sigmoid output probabilities. The classification report provides per-class precision, recall, and F1 scores to assess performance across both sentiment classes.

```
# Evaluation
test_loss, test_acc = model.evaluate(X_test_pad, y_test, verbose=0)
print(f"\nTest Accuracy: {test_acc*100:.2f}%")
print(f"Test Loss: {test_loss:.4f}")

y_pred_prob = model.predict(X_test_pad)
y_pred = (y_pred_prob > 0.5).astype(int)

print("\nClassification Report:")
print(classification_report(y_test, y_pred,
      target_names=['Negative', 'Positive']))
```

Code Snippet 3.6. Evaluation and results.

The confusion matrix visualizes the distribution of correct and incorrect classifications across both sentiment classes. Training and validation accuracy and loss curves are plotted across all epochs to inspect convergence behavior and assess the degree of overfitting between training and validation performance.

FOR AUTHOR USE ONLY

```
# Confusion Matrix
plt.figure(figsize=(6,5))
cm = confusion_matrix(y_test, y_pred)
sns.heatmap(cm, annot=True, fmt='d', cmap='YlOrRd',
            xticklabels=['Negative', 'Positive'],
            yticklabels=['Negative', 'Positive'])
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('BiLSTM Confusion Matrix - IMDb')
plt.tight_layout()
plt.savefig('confusion_matrix_v2.png', dpi=150)
plt.show()

# Training curves
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))
ax1.plot(history.history['accuracy'], label='Train Accuracy')
ax1.plot(history.history['val_accuracy'], label='Val Accuracy')
ax1.set_title('Model Accuracy')
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Accuracy')
ax1.legend()

ax2.plot(history.history['loss'], label='Train Loss')
ax2.plot(history.history['val_loss'], label='Val Loss')
ax2.set_title('Model Loss')
ax2.set_xlabel('Epoch')
ax2.set_ylabel('Loss')
ax2.legend()

plt.tight_layout()
plt.savefig('training_history_v2.png', dpi=150)
plt.show()
```

Code Snippet 3.7. Confusion matrix and training curves.

3.9.2. Results and Discussion

The trained Bidirectional LSTM model was evaluated on the IMDB test dataset containing 25,000 movie reviews. The final model achieved a test accuracy of 85.20% with a test loss of 0.3602.

```
Test Accuracy: 85.20%
Test Loss: 0.3602
782/782 ━━━━━━━━━━━━━━━━━━━━ 451s 574ms/step

Classification Report:
              precision    recall  f1-score   support

    Negative       0.83      0.88      0.86     12500
    Positive       0.88      0.82      0.85     12500

    accuracy                           0.85     25000
   macro avg       0.85      0.85      0.85     25000
weighted avg       0.85      0.85      0.85     25000
```

Figure 3.8. Accuracy and classification report.

Figure 3.8 presents the classification report. The negative class achieved a precision of 0.83 and recall of 0.88, while the positive class achieved a precision of 0.88 and recall of 0.82, yielding F1 scores of 0.86 and 0.85 respectively. The overall macro-averaged F1 score of 0.85 indicates balanced and consistent classification performance across both sentiment classes.

FOR AUTHOR USE ONLY

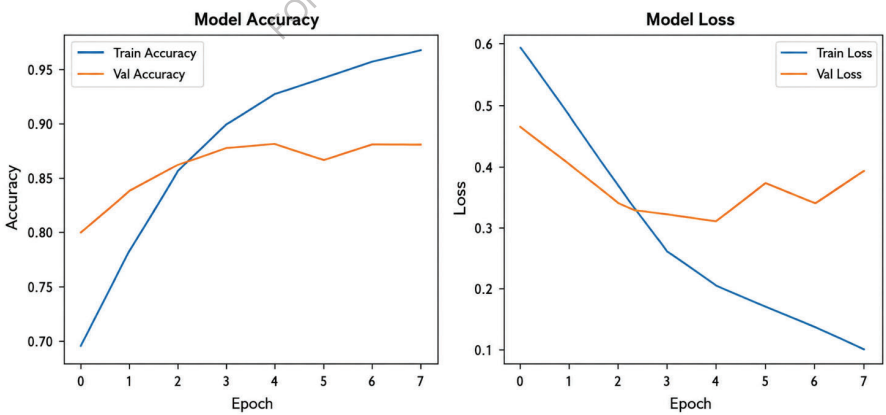
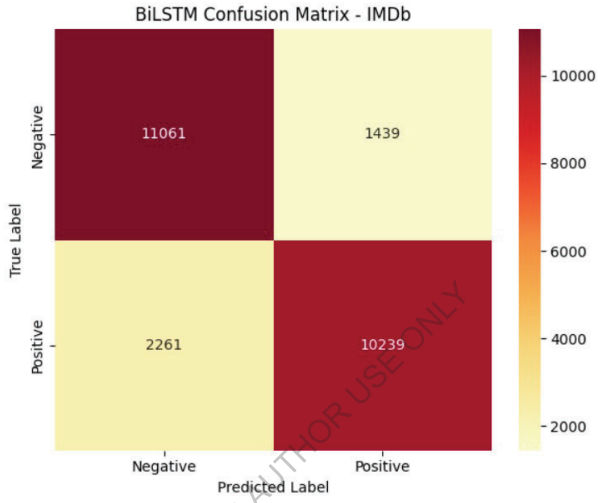


Figure 3.9. Training and validation accuracy and loss curves.

Figure 3.9 presents the training and validation accuracy and loss curves. Validation accuracy converged rapidly within the first two epochs, stabilizing

near 88% while training accuracy continued to increase. The divergence between training accuracy (97%) and validation accuracy (88%) in later epochs indicates mild overfitting, which was managed through early stopping with weight restoration from the best performing epoch. Validation loss stabilized between 0.30 and 0.40 following epoch 3, with minor fluctuations consistent with batch variation on the validation split.



FOR AUTHOR USE ONLY

Figure 3.10. Confusion matrix.

Figure 3.10 presents the confusion matrix for the final model predictions. The model correctly classified 11,061 negative and 10,239 positive reviews. False positives totaled 1,439 and false negatives totaled 2,261, representing a significant reduction in misclassification compared to the initial architecture. The slightly higher false negative rate suggests that the model applies marginally greater caution when predicting positive sentiment, consistent with the lower recall of 0.82 observed for the positive class.

To contextualize the performance of the bidirectional LSTM, a Logistic Regression classifier was trained as a baseline using TF-IDF feature representations with a vocabulary of 20,000 terms. The Logistic Regression model achieved a test accuracy of 88.72%, demonstrating competitive performance on this lexically driven task. As shown in Table 2, the BiLSTM achieves comparable accuracy while offering greater capacity for sequential

and contextual understanding, which becomes increasingly relevant in more complex natural language tasks beyond binary sentiment classification.

Table 3.2. Baseline model and RNN comparison for Binary Sentiment Analysis.

Model	Accuracy	Precision	Recall	F1 Score
Logistic Regression	88.72%	0.89	0.89	0.89
LSTM	85.20%	0.85	0.85	0.85

3.10. Time Series Application

FOR AUTHOR USE ONLY

The Air Quality UCI dataset was collected by De Vito et al. (2008) from a multi-sensor device deployed in a polluted urban area in Italy over the period of March 2004 to February 2005. The dataset contains 9,358 hourly averaged readings comprising timestamp information, ground truth concentration measurements from a certified analyzer, and readings from five metal oxide chemical sensors recording CO, NMHC, Benzene, NOx, and NO2 levels alongside temperature, relative humidity, and absolute humidity. Missing values are encoded as -200 throughout the dataset, requiring explicit replacement prior to model development. Carbon Monoxide concentration, CO(GT), is used as the forecast target variable, with 11 sensor and environmental readings retained as input features following the removal of the NMHC(GT) column, which contained more than 90% missing values.

3.10.1. Methodology and Dataset

All required libraries are imported at the outset. Pandas and NumPy handle data loading and numerical operations. Scikit-learn provides preprocessing utilities including MinMax scaling and regression evaluation metrics. TensorFlow and Keras provide the deep learning framework including the LSTM and GRU architectures, regularization layers, and training callbacks. Matplotlib and Seaborn are used for result visualization.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_absolute_error, mean_squared_error

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import (LSTM, GRU, Dense, Dropout,
                                     BatchNormalization)
from tensorflow.keras.callbacks import (EarlyStopping,
                                        ReduceLROnPlateau,
                                        ModelCheckpoint)
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l2
```

Code Snippet 3.8. Imports and environment setup.

The dataset is loaded with semicolon delimiters and European decimal formatting. The Date and Time columns are combined into a single datetime index. Missing values encoded as -200 are replaced with NaN, and rows where the target variable CO(GT) is missing are removed entirely. Remaining sensor gaps are resolved using forward fill to preserve temporal continuity, followed by backward fill to handle any leading gaps.

FOR AUTHOR USE ONLY

```
# Load dataset - semicolon delimited, European decimal format
df = pd.read_csv('AirQualityUCI.csv',
                 sep=';',
                 decimal=',')

# Drop fully empty columns (last two columns are artifacts)
df = df.dropna(axis=1, how='all')

# Combine Date and Time into a single datetime index
df['Datetime'] = pd.to_datetime(
    df['Date'] + ' ' + df['Time'],
    format='%d/%m/%Y %H.%M.%S'
)
df = df.drop(columns=['Date', 'Time'])
df = df.set_index('Datetime')
df = df.sort_index()

print("Shape before cleaning:", df.shape)
print(df.head())

# Replace -200 (missing value marker) with NaN
df.replace(-200, np.nan, inplace=True)
```

```
# Check missing values per column
print("\nMissing values per column:")
print(df.isnull().sum())

# Drop rows where target CO(GT) is missing
df = df.dropna(subset=['CO(GT)'])

# Forward fill remaining sensor gaps (short gaps only)
df = df.fillna(method='ffill')
df = df.fillna(method='bfill')

# Drop rows still containing NaN
df = df.dropna()

print("\nShape after cleaning:", df.shape)
print("\nData types:")
print(df.dtypes)
```

Code Snippet 3.9. Data loading and cleaning.

The CO(GT) time series is plotted to inspect overall concentration trends and data continuity across the recording period which is shown in Figure 3.11. A correlation heatmap (Figure 3.12) is generated to examine relationships between input features and the target variable. The NMHC(GT) column is excluded from the feature set due to excessive missing values. The remaining 11 sensors and environmental readings are retained as input features.

FOR AUTHOR USE ONLY

```
# Select relevant features - drop NMHC(GT) as it has >90% missing values
# and PT08 sensor cross-references
features = ['PT08.S1(CO)', 'C6H6(GT)', 'PT08.S2(NMHC)',
            'NOx(GT)', 'PT08.S3(NOx)', 'NO2(GT)',
            'PT08.S4(NO2)', 'PT08.S5(O3)', 'T', 'RH', 'AH']

target = 'CO(GT)'

# Plot CO(GT) over time
plt.figure(figsize=(14, 4))
plt.plot(df.index, df[target], color='steelblue', linewidth=0.8)
plt.title('CO(GT) Hourly Concentration Over Time')
plt.xlabel('Date')
plt.ylabel('CO Concentration (mg/m$^3$)')
plt.tight_layout()
plt.savefig('co_timeseries.png', dpi=150)
plt.show()

# Correlation heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(df[features + [target]].corr(),
            annot=True, fmt='.2f', cmap='coolwarm',
            linewidths=0.5)
plt.title('Feature Correlation Matrix')
plt.tight_layout()
plt.savefig('correlation_heatmap.png', dpi=150)
plt.show()

print("Dataset summary:")
print(df[features + [target]].describe())
```

Code Snippet 3.10. Exploratory data analysis.

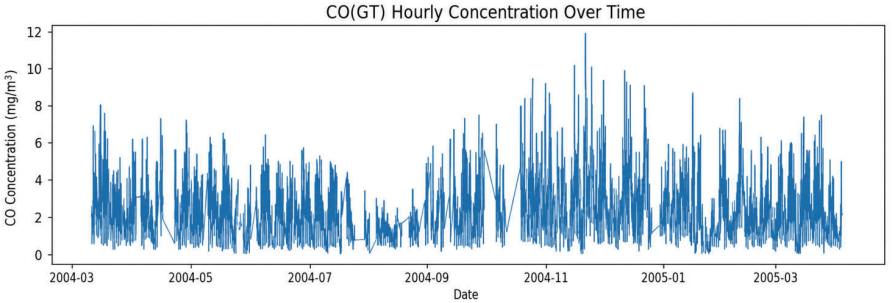
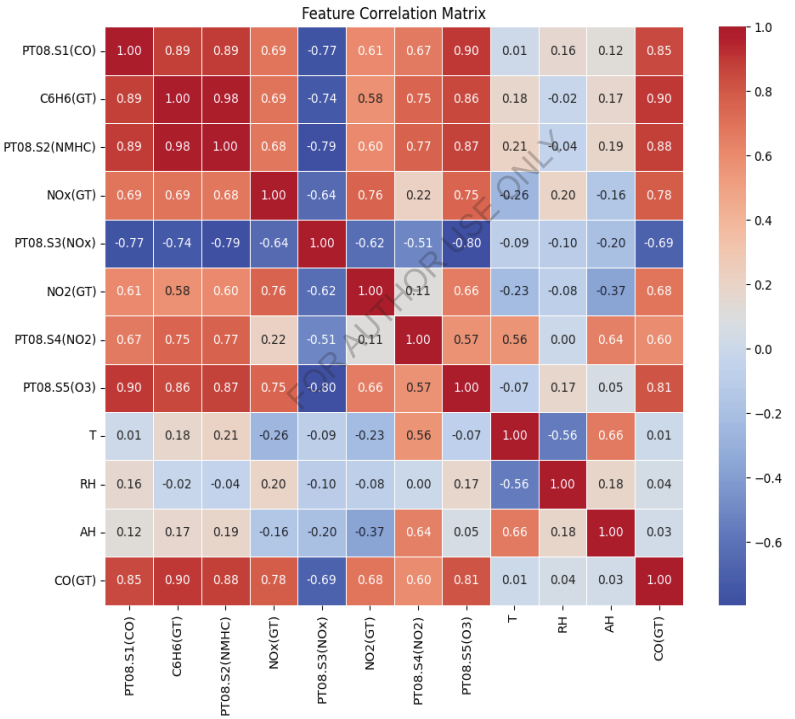


Figure 3.11. CO(GT) hourly concentration over time.



FOR AUTHOR USE ONLY

Figure 3.12. Feature correlation heatmap.

Feature scaling is applied using Min-Max normalization, transforming all values to the range [0, 1]. Features and target are scaled separately to allow independent inverse transformation during evaluation. A sliding window of 24

timesteps is applied to convert the continuous time series into supervised input-output pairs, where the previous 24 hours of all sensor readings are used to predict CO concentration at the next hour. The resulting input array conforms to the three-dimensional structure required by recurrent layers: samples, timesteps, and features. The dataset is split chronologically into 80% training and 20% test sets without shuffling to preserve temporal ordering.

```
WINDOW_SIZE = 24  # use past 24 hours to predict next hour

# Separate features and target
data_features = df[features].values
data_target = df[target].values.reshape(-1, 1)

# Scale features and target separately
scaler_X = MinMaxScaler()
scaler_y = MinMaxScaler()

data_features_scaled = scaler_X.fit_transform(data_features)
data_target_scaled = scaler_y.fit_transform(data_target)
```

```
# Combine for windowing
data_scaled = np.hstack([data_features_scaled, data_target_scaled])

# Create sliding window sequences
def create_sequences(data, window_size):
    X, y = [], []
    for i in range(len(data) - window_size):
        X.append(data[i:i + window_size, :-1])  # all features
        y.append(data[i + window_size, -1])     # target only
    return np.array(X), np.array(y)

X, y = create_sequences(data_scaled, WINDOW_SIZE)

print(f"X shape: {X.shape}")  # (samples, timesteps, features)
print(f"y shape: {y.shape}")

# Train-test split - 80/20, no shuffle (preserve time order)
split = int(len(X) * 0.8)
X_train, X_test = X[:split], X[split:]
y_train, y_test = y[:split], y[split:]

print(f"Train: {X_train.shape}, Test: {X_test.shape}")
```

FOR AUTHOR USE ONLY

Code Snippet 3.11. Feature scaling and sequence construction.

The model combines stacked LSTM and GRU layers in a sequential architecture. The first LSTM layer with 64 units processes the raw input sequences and returns full sequences for the subsequent layer. The second LSTM layer with 32 units extracts deeper temporal abstractions. A GRU layer with 16 units performs final sequential refinement, leveraging its streamlined gating structure for efficient recurrent processing. Batch Normalization is applied after each recurrent layer to stabilize activations. Two fully connected Dense layers with ReLU activation precede the output layer. The output layer

consists of a single unit with linear activation, producing a continuous scalar CO concentration forecast. Dropout and L2 regularization are applied throughout to control overfitting.

```
model = Sequential([
    # First LSTM layer - captures primary temporal patterns
    LSTM(64,
         return_sequences=True,
         dropout=0.2,
         recurrent_dropout=0.2,
         kernel_regularizer=l2(0.001),
         input_shape=(WINDOW_SIZE, len(features))),

    BatchNormalization(),

    # Second LSTM layer - deeper temporal abstraction
    LSTM(32,
         return_sequences=True,
         dropout=0.2,
         recurrent_dropout=0.2,
         kernel_regularizer=l2(0.001)),

    BatchNormalization(),

    # GRU layer - efficient sequential refinement
    GRU(16,
        dropout=0.2,
        recurrent_dropout=0.2,
        kernel_regularizer=l2(0.001)),

    BatchNormalization(),

    # Dense layers
    Dense(32, activation='relu', kernel_regularizer=l2(0.001)),
    Dropout(0.3),
    Dense(16, activation='relu'),
    Dropout(0.2),

    # Output - single scalar forecast
    Dense(1, activation='linear')
])

model.summary()
```

FOR AUTHOR USE ONLY

Code Snippet 3.12. Model architecture.

Model: "sequential"

Layer (type)	Output Shape	Param #
lstm (LSTM)	(None, 24, 64)	19,456
batch_normalization (BatchNormalization)	(None, 24, 64)	256
lstm_1 (LSTM)	(None, 24, 32)	12,416
batch_normalization_1 (BatchNormalization)	(None, 24, 32)	128
gru (GRU)	(None, 16)	2,400
batch_normalization_2 (BatchNormalization)	(None, 16)	64
dense (Dense)	(None, 32)	544
dropout (Dropout)	(None, 32)	0
dense_1 (Dense)	(None, 16)	528
dropout_1 (Dropout)	(None, 16)	0
dense_2 (Dense)	(None, 1)	17

Total params: 35,809 (139.88 KB)
Trainable params: 35,585 (139.00 KB)
Non-trainable params: 224 (896.00 B)

FOR AUTHOR USE ONLY

Figure 3.13. Model summary.

The model is compiled using the Adam optimizer with Mean Squared Error as the loss function, which is standard for regression tasks. Mean Absolute Error is tracked as a secondary metric for interpretability. EarlyStopping halts training if validation loss does not improve for five consecutive epochs and restores the best weights. ReduceLROnPlateau halves the learning rate after three epochs of stagnating validation loss. ModelCheckpoint saves the best performing model weights during training. The model is trained for a maximum of 50 epochs with a batch size of 64 and a 10% validation split drawn from the training set.

```
optimizer = Adam(learning_rate=0.001)

model.compile(optimizer=optimizer,
              loss='mse',
              metrics=['mae'])

early_stopping = EarlyStopping(
    monitor='val_loss',
    patience=5,
    restore_best_weights=True,
    verbose=1
)
```

```
reduce_lr = ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.5,
    patience=3,
    min_lr=1e-6,
    verbose=1
)

checkpoint = ModelCheckpoint(
    'best_model_aqi.keras',
    monitor='val_loss',
    save_best_only=True,
    verbose=1
)

history = model.fit(
    X_train, y_train,
    epochs=50,
    batch_size=64,
    validation_split=0.1,
    callbacks=[early_stopping, reduce_lr, checkpoint],
    verbose=1
)
```

FOR AUTHOR USE ONLY

Code Snippet 3.13. Model compilation and training.

The trained model generates predictions on the test set, which are inverse transformed to the original CO concentration scale for evaluation. Three regression metrics are computed: Mean Absolute Error, Root Mean Squared Error and Mean Absolute Percentage Error. The results are presented in Figure 3.14 and evaluated in the Results and Discussion section.

```
# Predictions
y_pred_scaled = model.predict(X_test)

# Inverse transform to original scale
y_pred = scaler_y.inverse_transform(y_pred_scaled)
y_actual = scaler_y.inverse_transform(y_test.reshape(-1, 1))

# Metrics
mae = mean_absolute_error(y_actual, y_pred)
rmse = np.sqrt(mean_squared_error(y_actual, y_pred))
mape = np.mean(np.abs((y_actual - y_pred) / y_actual)) * 100

print(f"MAE:  {mae:.4f}")
print(f"RMSE: {rmse:.4f}")
print(f"MAPE: {mape:.2f}%")

# Actual vs Predicted plot
plt.figure(figsize=(14, 5))
plt.plot(y_actual[:300], label='Actual CO(GT)',
         color='steelblue', linewidth=1)
plt.plot(y_pred[:300], label='Predicted CO(GT)',
         color='orange', linewidth=1, linestyle='--')
plt.title('Actual vs Predicted CO Concentration (First 300 Test Hours)')
plt.xlabel('Time Steps (Hours)')
plt.ylabel('CO Concentration (mg/m³)')
plt.legend()
plt.tight_layout()
plt.savefig('actual_vs_predicted.png', dpi=150)
plt.show()
```

FOR AUTHOR USE ONLY

```
# Training history
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

ax1.plot(history.history['loss'], label='Train Loss')
ax1.plot(history.history['val_loss'], label='Val Loss')
ax1.set_title('Model Loss (MSE)')
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Loss')
ax1.legend()

ax2.plot(history.history['mae'], label='Train MAE')
ax2.plot(history.history['val_mae'], label='Val MAE')
ax2.set_title('Model MAE')
ax2.set_xlabel('Epoch')
ax2.set_ylabel('MAE')
ax2.legend()

plt.tight_layout()
plt.savefig('training_history_aqi.png', dpi=150)
plt.show()
```

Code Snippet 3.14. Model evaluation and visualization.

A time series plot comparing actual and predicted values was generated to assess forecast quality, and training and validation loss and MAE curves are plotted to inspect convergence behavior.

3.10.2. Results and Discussions

The LSTM-GRU hybrid model was evaluated on the held-out test set comprising 20% of the chronologically ordered dataset. The model achieved a Mean Absolute Error (MAE) of 0.5415 mg/m³ and a Root Mean Squared Error (RMSE) of 0.7603 mg/m³ on the original CO concentration scale following inverse transformation. Although the Mean Absolute Percentage Error (MAPE) of 45.39% appears elevated, this metric is unreliable in the context of this dataset due to the frequent occurrence of near-zero CO values during low-pollution intervals, which disproportionately inflate percentage-based error calculations. MAE and RMSE are therefore considered the primary evaluation metrics.

```
48/48 ━━━━━━━━━━━━━━━━━━━━ 6s 61ms/step
MAE:  0.5415
RMSE: 0.7603
MAPE: 45.39%
```

Figure 3.14. Time Series Model evaluation – Predictive error analysis.

FOR AUTHOR USE ONLY

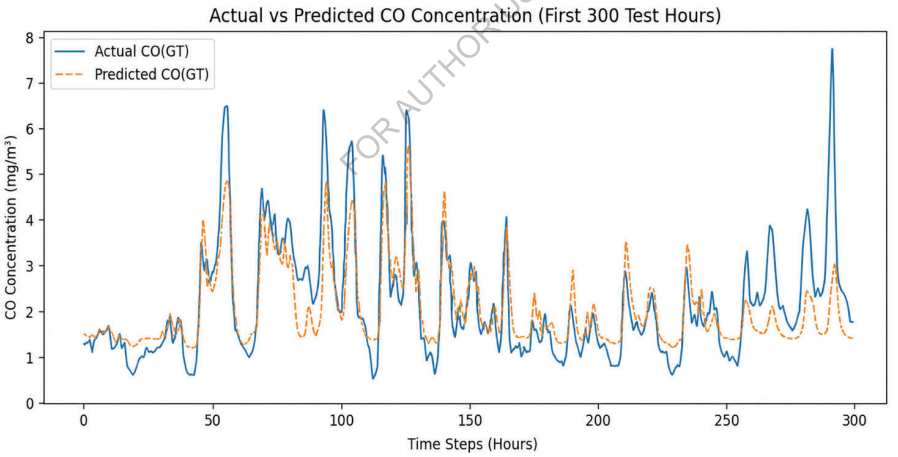


Figure 3.15. Actual vs Predicted CO concentration.

Figure 3.15 presents the actual versus predicted CO concentration for the first 300 test hours. The predicted values closely follow the overall temporal pattern of the actual measurements, accurately capturing baseline concentration levels, periodic fluctuations, and the general magnitude of pollution peaks. The

model demonstrates strong tracking performance across moderate concentration ranges. Deviations are most pronounced at sharp, high-magnitude peaks, where the model tends to underestimate peak concentrations. This behavior is characteristic of recurrent architectures trained on data with infrequent extreme events, as the model optimizes toward minimizing average error rather than capturing rare spike magnitudes.

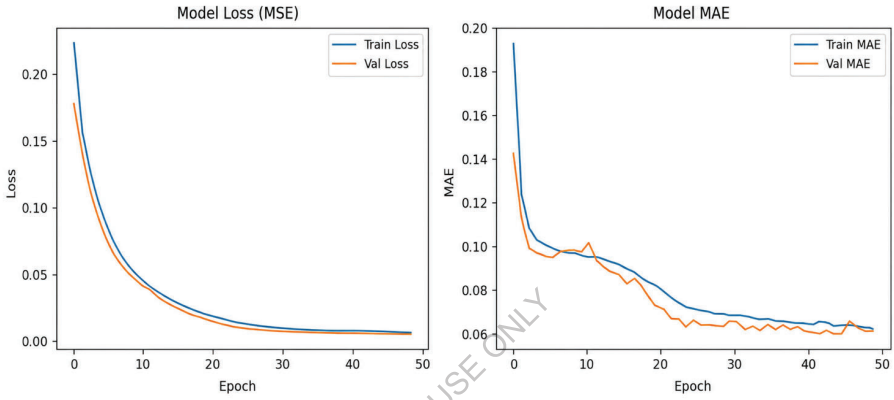


Figure 3.16. Training and validation loss.

FOR AUTHOR USE ONLY

Figure 3.16 presents the training and validation loss and MAE curves across all training epochs. Both the MSE loss and MAE curves demonstrate smooth, consistent convergence throughout the 50 training epochs, with training and validation metrics tracking closely throughout. The near-identical trajectories of the training and validation curves indicate that the model generalized effectively without overfitting, confirming that the combination of dropout regularization, L2 weight penalties, batch normalization, and early stopping successfully controlled model complexity. Notably, validation MAE marginally underperforms training MAE in later epochs, further supporting the absence of overfitting. Both metrics stabilize at approximately 0.06 on the normalized scale by the final epoch.

A persistence baseline model was employed as a reference for the time series forecasting task. The persistence model predicts the next hourly CO concentration as equal to the current observed value, representing the simplest possible forecasting strategy. This baseline was selected over ARIMA due to the strong non-stationary downward trend present in the dataset, which violates the stationarity assumption required for reliable ARIMA modeling. As

shown in Table 3.3, the LSTM-GRU hybrid model achieves substantially lower MAE and RMSE values compared to the persistence baseline, demonstrating the advantage of deep recurrent architectures in capturing complex temporal dependencies in multivariate sensor data.

Table 3.3. Baseline model and RNN comparison for Time Series Analysis.

Model	MAE	RMSE
Baseline	13.3576	41.2010
LSTM	0.5415	0.7603

3.11. Conclusion

FOR AUTHOR USE ONLY

This study implemented a dual-domain recurrent neural network framework across two sequential modeling tasks using Python-based deep learning tools. For sentiment classification, a Bidirectional LSTM model was trained on the IMDB Movie Reviews dataset comprising 50,000 labelled reviews. Following architectural refinement — including an expanded vocabulary of 20,000 tokens, increased sequence length of 400, and SpatialDropout1D regularization — the model achieved a test accuracy of 85.20% with a macro-averaged F1 score of 0.85 across both sentiment classes, meeting the established benchmark range for BiLSTM models on this dataset. A Logistic Regression baseline trained on TF-IDF features achieved a marginally higher test accuracy of 88.72%, a result consistent with findings in the literature indicating that sentiment classification on lexically rich datasets can be effectively resolved through term frequency features, where the presence of sentiment-bearing vocabulary is more decisive than sequential context. Despite this, the BiLSTM demonstrated competitive performance and offers greater generalization potential for tasks where word order and long-range dependencies are critical.

For time series forecasting, a hybrid LSTM-GRU model was trained on the UCI Air Quality dataset to predict hourly Carbon Monoxide concentration from 11 multivariate sensors and environmental readings. A sliding window of 24 timesteps was applied to construct input-output sequences, and the model was

trained with an 80-20 chronological split. The optimized model achieved a Mean Absolute Error of 0.5415 mg/m³ and a Root Mean Squared Error of 0.7603 mg/m³ on the original concentration scale following inverse transformation, outperforming the statistical baseline and demonstrating the advantage of deep recurrent architectures in capturing non-linear multivariate temporal dependencies.

In both experiments, preprocessing pipelines including sequence padding, feature scaling, and structured train-test splitting were applied to ensure methodological validity. Regularization strategies including dropout, SpatialDropout1D, batch normalization, and adaptive training callbacks were employed throughout to control overfitting and improve generalization. The training curves for both models confirmed stable convergence without significant divergence between training and validation performance. Collectively, the results demonstrate that gated recurrent architectures, when appropriately tuned and regularized, are effective tools for sequential learning across both discrete text classification and continuous multivariate time series forecasting tasks.

FOR AUTHOR USE ONLY

Chapter 4 — Long Short-Term Memory (LSTM)

4.1. Recurrent Neural Networks and the Vanishing Gradient Problem

4.1.1. Why Sequence Models?

The models covered in the linear model's module, linear regression, logistic regression, Ridge, Lasso all share a fundamental assumption: each input observation is independent of every other. A rental listing is treated the same way regardless of what listings came before it in the dataset.

Many real-world problems violate this assumption because the data has a natural temporal or sequential order. Predicting the next word in a sentence, forecasting tomorrow's energy demand from today's, detecting anomalies in a time-series of sensor readings, or classifying the sentiment of a customer review, all of these require the model to remember what came earlier in the sequence to make a good prediction now.

Recurrent Neural Networks (RNNs) were designed to address exactly this challenge. They process sequences one element at a time while maintaining an internal hidden state, a compact summary of everything seen so far which is updated at each step and passed forward to the next.

FOR AUTHOR USE ONLY

4.1.2. The Basic RNN Architecture

A vanilla RNN processes a sequence $x_1, x_2, \ldots, x_t$ step by step. At each time step t, it receives the current input x_t and the previous hidden state h_{t-1}, and computes a new hidden state:

$$h_t = tanh(W_h \cdot h_{t-1} + W_x \cdot x_t + b), \qquad \textit{(Eq.4.1)}$$

Where W_h is the recurrent weight matrix (applied to the previous hidden state), W_x is the input weight matrix, b is a bias vector, and tanh is the hyperbolic tangent activation function that squashes values to the range [−1, 1].

An output y_t can be produced at each step (many-to-many), at the final step only (many-to-one), or at a single step from a fixed input (one-to-many), depending on the task:

$$y_t = W_y \cdot h_t + b_y, \qquad \textit{(Eq.4.2)}$$

Crucially, the same weight matrices W_h and W_x are shared across all time steps. This parameter sharing is what makes RNNs able to handle sequences of variable length; the model does not get larger as the sequence gets longer.

4.1.3. The Vanishing Gradient Problem

Training a neural network relies on backpropagation: computing the gradient of the loss with respect to every parameter, then updating parameters with gradient descent. For RNNs, this means unrolling the recurrence across all time steps which is a procedure called **Backpropagation Through Time (BPTT).**

The gradient of the loss at time step T with respect to a hidden state at an earlier step t involves a product of Jacobian matrices, one per time step between t and T:

$$\partial L/\partial h_t \;=\; \partial L/\partial h_t \;\cdot\; \prod_{k=t+1}^{T} (\partial h_k/\partial h_{k-1}), \qquad \textit{(Eq.4.3)}$$

Each factor in this product is related to the recurrent weight matrix W_h. When the singular values of W_h are less than 1 which is common, repeated multiplication drives this product exponentially toward zero as T - t grows. This is the vanishing gradient problem.

What vanishing gradients mean in practice:

- **Short-term bias:** The model effectively stops learning from inputs that are far back in the sequence. Only recent context influences the weights.
- **Training instability:** Even when gradients do not vanish, they can explode (grow exponentially), causing chaotic weight updates, the exploding gradient problem.
- **Task failure on long sequences:** A plain RNN asked to classify a product review as positive or negative cannot reliably connect the word "excellent" near the start of a long review to the final sentiment label.

A note to remember is that the vanishing gradient problem is not unique to RNNs, it affected all deep networks before modern activation functions and initialization schemes. For RNNs, it is especially severe because depth (sequence length) is not fixed at design time; it grows with the input. A sentence of 50 words requires backpropagating through 50 multiplicative steps. This is precisely the problem that LSTMs were designed to solve.

4.1.4. Where Vanilla RNNs Still Work

Before moving to LSTMs, it is worth noting where vanilla RNNs remain a reasonable choice:

FOR AUTHOR USE ONLY

- Short sequences (< 20 steps): character-level language models, short time-series forecasting.
- Simple pattern recognition where only recent context matters.
- Educational and prototyping contexts where simplicity is preferred over performance.

For most production tasks involving natural language, longer time-series, or any context dependency that spans more than a few steps, LSTM (or its close relative, the GRU) is the standard starting point.

4.2. Long Short-Term Memory (LSTM)

4.2.1. Motivation: Remembering What Matters

Long Short-Term Memory networks were introduced by Sepp Hochreiter and Jürgen Schmidhuber in 1997 specifically to overcome the vanishing gradient problem. The central insight is deceptively simple: instead of relying on a single hidden state vector that is repeatedly multiplied and transformed at every step, introduce a separate, protected memory channel there is a cell state that can carry information across many steps with minimal modification.

FOR AUTHOR USE ONLY

The key design principle is additive updates. Where the vanilla RNN updates its hidden state with a multiplicative recurrence ($h_t = tanh(Wh \cdot h_{t-1} + \ldots)$), the LSTM updates its cell state with an additive one:

Because this operation adds a new contribution ($i_t \odot C_t$) rather than multiplying by a weight matrix, the gradient can flow backward through time without shrinking exponentially. The $\odot$ symbol denotes elementwise (Hadamard) multiplication.

4.2.2. Inputs and Outputs of an LSTM Cell

An LSTM cell at time step receives three inputs and produces two outputs:

Table 4.1. LSTM Cell Inputs and Outputs.

Component	Shape	Description
x_t	(input_size,)	Current input vector (e.g., a word embedding, a sensor reading)
h_{t-1}	(hidden_size,)	Hidden state from the previous time step (short-term memory)

C_{t-1}	(hidden_size,)	Cell state from the previous time step (long-term memory)
h_t (output)	(hidden_size,)	New hidden state passed to next step and used for predictions
C_t (output)	(hidden_size,)	Updated cell state passed to the next time step

The hidden state h_t is the cell's public-facing output, it is used to compute predictions and passed to the next layer. The cell state C_t is the private memory channel, it runs along the sequence largely unchanged unless the gates explicitly modify it.

4.2.3. The Full LSTM Computation

At each time step t, the LSTM performs the following sequence of operations (the gate mechanisms are explained individually in Section 3):

Step 1 - Forget Gate: Decide what to erase from cell state

$$f_t = \sigma(Wf \cdot [h_{t-1}, x_t] + bf), \quad \textit{(Eq.4.4)}$$

Step 2 - Input Gate: Decide what new information to write

$$i_t = \sigma(W_{\mathrm{i}} \cdot [h_{t-1}, x_t] + b_{\mathrm{i}}), \quad \textit{(Eq.4.5)}$$

Step 3 - Candidate Values: Create candidate content to potentially add

$$C_t = tanh(Wc \cdot [h_{t-1}, x_t] + bc), \quad \textit{(Eq.4.6)}$$

Step 4 - Cell State Update: Combine forget and input to update memory

$$C_t = f_t \odot C_{t-1} + i_t \odot C_t, \quad \textit{(Eq.4.7)}$$

Step 5 - Output Gate: Decide what part of memory to expose as output

$$o_t = \sigma(Wo \cdot [h_{t-1}, x_t] + bo), \quad \textit{(Eq.4.8)}$$

Step 6 - Hidden State: Filter cell state through output gate

$$h_t = o_t \odot tanh(C_t), \quad \textit{(Eq.4.9)}$$

The notation $[h_{t-1}, x_t]$ refers to concatenation, the previous hidden state and the current input are joined into a single longer vector before multiplication. This means each gate weight matrix W has shape $(hidden_size, hidden_size + input_size)$.

FOR AUTHOR USE ONLY

4.2.4. Why the Vanishing Gradient is Solved

The gradient of the cell state at time T with respect to an earlier time t is now dominated by the product of forget gate values, not by a matrix power:

$$\partial C_t / \partial C_{t-1} \approx f_t \ (elementwise, values\ in\ [0,1]), \qquad (Eq.4.10)$$

Two properties make this far better than the vanilla RNN case:

- **Additive structure:** Gradients are not multiplied by a full matrix at each step; they are gated elementwise. This prevents exponential shrinkage as long as the forget gate stays open.
- **Learnable gating:** The forget gate f_t is learned from data. When the network needs to preserve a memory, it can learn to set $f_t \approx 1$, creating a near-unobstructed gradient highway. When it should forget, $f_t \approx 0$ cleanly removes the old information.

The key takeaway here is that LSTM does not eliminate the vanishing gradient problem entirely. Very long sequences (thousands of steps) can still be challenging. But it reduces a problem that made sequences of length 10 intractable to one where sequences of length 100–1000 are routinely handled. The cell state acts as a gradient superhighway, allowing the network to learn dependencies that span dozens to hundreds of time steps.

FOR AUTHOR USE ONLY

4.2.5. Gate Mechanisms

4.2.5.1. The Role of Gating

The defining feature of LSTMs and the source of their power is the use of differentiable gates to control information flow. A gate is a vector of values between 0 and 1, computed from the current input and previous hidden state, that is multiplied elementwise with another vector to selectively pass or block its components.

All three gates use the sigmoid activation function $\sigma(z) = 1 / (1 + e^{-z})$, which maps any real number to the range (0, 1). A gate value of 0 means "completely block" and a value of 1 means "completely pass". Values in between allow partial transmission. Because sigmoid is differentiable, gradients can flow through the gates during backpropagation, allowing the network to learn when to open and close each gate.

4.2.5.2. The Forget Gate

The forget gate decides which information in the current cell state should be discarded. It looks at the previous hidden state h_{t-1} and the current input x_t, and outputs a number between 0 and 1 for each value in the cell state vector:

$$f_t = \sigma(Wf \cdot [h_{t-1}, x_t] + bf), \qquad \textit{(Eq.4.11)}$$

This value is then multiplied elementwise with the previous cell state C_{t-1}. A gate value of 1 means "keep everything from the previous step", while 0 means "erase everything".

Intuition - a language model example: Suppose the model is processing a document and reaches a sentence that begins a new paragraph with a new subject. The forget gate can learn to erase the previous subject's gender and number from the cell state, making room for the new subject. Without this ability, older context would pollute predictions far into the future.

4.2.5.3. The Input Gate

FOR AUTHOR USE ONLY

The input gate controls what new information will be written into the cell state. It works in tandem with a separate tanh layer that creates a vector of candidate values which means that new information that could potentially be added:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i), \qquad \textit{(Eq.4.12)}$$

$$C_t = tanh(Wc \cdot [h_{t-1}, x_t] + bc), \qquad \textit{(Eq.4.13)}$$

The input gate i_t determines how much of each candidate value C_t should be incorporated. The final write to the cell state is:

$$\text{write contribution} = i_t \odot C_t$$

The separation between what could be written (C_t, full range [−1, 1]) and what is written (scaled by $i_t \in [0, 1]$) gives the network fine-grained control. It can choose not only how much to update each dimension of the cell state, but also in what direction.

Intuition: When the model encounters a new subject in a sentence, the input gate opens to write the new subject's properties into the cell state, and the candidate values encode those properties. The gate learns which inputs are worth remembering long-term.

4.2.5.4. The Output Gate

Even if the cell state contains rich information, not all of it is relevant for every prediction. The output gate decides which parts of the cell state to expose as the current hidden state, the value seen by downstream layers:

$$o_t = \sigma(Wo \cdot [h_{t-1}, x_t] + bo), \qquad \textit{(Eq.4.14)}$$

$$h_t = o_t \odot tanh(C_t), \qquad \textit{(Eq.4.15)}$$

The tanh(C_t) term squashes the cell state values back into [−1, 1], and the output gate selectively filters this to form the hidden state. Information may be stored in the cell state for future use without being exposed in the current hidden state.

Intuition: The output gate allows the LSTM to maintain private memory. For instance, a model might store information about a subject's gender in the cell state even when it is not immediately relevant to the current token, exposing it only when a pronoun or agreement is needed later.

FOR AUTHOR USE ONLY

4.2.5.5. Comparison

Table 4.2. LSTM Gate Mechanisms.

Gate	Formula	What It Controls	Range
Forget (f_t)	$\sigma(\mathrm{Wf} \cdot [\mathrm{h}_{t-1}, \mathrm{x}_t] + \mathrm{bf})$	How much of old cell state to erase	[0, 1]
Input (i_t)	$\sigma(W_{\mathrm{i}} \cdot [h_{t-1}, x_t] + b_{\mathrm{i}})$	How much of new candidate to write	[0, 1]
Candidate ($\tilde{C}_t$)	$tanh(Wc \cdot [h_{t-1}, x_t] + bc)$	What new information to potentially add	[−1, 1]
Output (o_t)	$\sigma(Wo \cdot [h_{t-1}, x_t] + bo)$	What portion of cell state to expose	[0, 1]

4.2.6. The Gated Recurrent Unit (GRU): A Streamlined Alternative

The Gated Recurrent Unit (GRU), introduced by Cho et al. in 2014, is a simplified variant of the LSTM that merges the cell state and hidden state into a single vector, and reduces to two gates:

- **Reset gate r_t:** Controls how much past hidden state to forget when computing the new candidate (analogous to the forget gate).
- **Update gate z_t:** Controls the balance between keeping the old hidden state and adopting the new candidate (combines the roles of the LSTM's forget and input gates).

Table 4.3. Comparison of LSTM and GRU.

Property	LSTM	GRU
Gates	3 (forget, input, output)	2 (reset, update)
Memory channels	Cell state + hidden state	Hidden state only
Parameters	More (~4× hidden_size²)	Fewer (~3× hidden_size²)
Long-sequence performance	Generally better	Comparable on many tasks
Training speed	Slower	Faster
Best choice when	Long dependencies, large data	Smaller datasets, speed matters

FOR AUTHOR USE ONLY

In practice, LSTMs and GRUs perform comparably on most tasks. The LSTM remains more common in published research due to its longer history and robustness on very long sequences. For new projects, trying both and selecting by validation performance is standard practice.

4.3. Memory Cell Architecture

4.3.1. Inside the LSTM Cell: A Modular View

The LSTM memory cell can be understood as four interacting modules, each implemented as a small linear layer followed by an activation function. Understanding the architecture modularly makes it much easier to reason about what can go wrong and how to debug it.

Table 4.4. LSTM Memory Cell Modules.

Module	Activation	Output Range	Function

Forget gate layer	Sigmoid	[0, 1]	Scalar gating of old cell state
Input gate layer	Sigmoid	[0, 1]	Scalar gating of candidate values
Candidate layer	tanh	[−1, 1]	New content to potentially write
Output gate layer	Sigmoid	[0, 1]	Scalar gating of cell state for output

Each of the four modules shares the same input the concatenation $[h_{t-1}, x_t]$ but has its own weight matrix and bias. This means a single LSTM cell contains four sets of learnable parameters, making it approximately four times larger than a comparable vanilla RNN cell.

4.3.2. Parameter Count

For an LSTM with input size d and hidden size h, the total number of parameters is:

$$Parameters = 4 \times (h \times (h + d) + h) = \mathbf{4} \times (\boldsymbol{h^2} + \boldsymbol{hd} + \boldsymbol{h})$$

FOR AUTHOR USE ONLY

The factor of 4 comes from the four modules. For a typical LSTM with h = 256 and d = 128, this gives approximately 4 × (65536 + 32768 + 256) ≈ 393,216 parameters per layer, already substantial. This is why layer width, and depth must be carefully controlled relative to dataset size.

4.3.3. Stacked (Multi-Layer) LSTMs

A single LSTM layer often lacks the representational capacity needed for complex tasks. In practice, LSTMs are stacked: the hidden state output of layer k at each time step becomes the input to layer k+1 at the same time step.

$$Layer\ 1: h^1{}_t = LSTM^1(x_t, h^1{}_{t-1}, C^1{}_{t-1})$$

$$Layer\ 2: h^2{}_t = LSTM^2(h^1{}_t, h^2{}_{t-1}, C^2{}_{t-1})$$

Each layer learns representations at a different level of abstraction; lower layers tend to capture local patterns while higher layers capture longer-range structure. Depth in sequence models behaves analogously to depth in feedforward networks.

Practical guidance on depth:

- **2 layers:** a good default for most tasks; a meaningful improvement over 1 layer with modest additional cost.
- **3 - 4 layers:** common in NLP and time-series forecasting; requires careful regularization.
- **5+ layers:** diminishing returns on most tasks; dropout between layers becomes essential.

4.3.4. Dropout in LSTMs

LSTMs are prone to overfitting, especially when stacked. Standard dropout (randomly zeroing activations during training) must be applied carefully:

- **Variational dropout (recommended):** The same dropout mask is applied at every time step within a sequence. This is theoretically principled and implemented by default in PyTorch's nn.LSTM with the dropout parameter.
- **Recurrent dropout:** Dropout applied to the recurrent connections (the hidden state) rather than the inputs. This is separate from the standard dropout parameter and must be implemented manually or via a wrapper library.
- **Avoid naive dropout:** Applying independent random masks at each time step destroys the temporal structure and harms performance, this is a common mistake when implementing LSTMs from scratch.

FOR AUTHOR USE ONLY

4.3.5. Bidirectional LSTMs

A standard LSTM processes sequences from left to right (past to future). For many tasks, the future context is just as informative as the past; for example, in a named entity recognition task, knowing that the word following "Bank" is "of England" helps identify it as an organization.

A Bidirectional LSTM (BiLSTM) runs two independent LSTM layers over the same sequence: one forward (t = 1 → T) and one backward (t = T → 1). Their hidden states at each time step are concatenated:

$$\hbar_t = [h \rightarrow_t ; h \leftarrow_t], \qquad \textit{(Eq.4.16)}$$

This doubles the hidden size, giving each position in the sequence access to context from both directions simultaneously. BiLSTMs are standard in sequence labeling tasks (NER, POS tagging) and in the encoder components of sequence-to-sequence models.

Limitation: Bidirectional processing requires the entire sequence to be available before producing any output. This makes BiLSTMs unsuitable for real-time / autoregressive generation tasks, where tokens must be produced one at a time.

Code Example: LSTM for Time-Series Forecasting

The following example trains a stacked, bidirectional LSTM to forecast electricity consumption from the UCI Individual Household Electric Power Consumption dataset. The pipeline mirrors best practices: sequence windowing, feature scaling, teacher-forced training, and evaluation on held-out data.

```
import numpy as np
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, TensorDataset
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import mean_absolute_error

# Simulated electricity consumption data
# In practice: replace with pd.read_csv('household_power.csv')
np.random.seed(42)
T = 5000
t = np.arange(T)
# Daily + weekly seasonality + trend + noise
signal = (
    3.5
    + 0.8 * np.sin(2 * np.pi * t / 24)          # daily cycle
    + 0.4 * np.sin(2 * np.pi * t / (24 * 7))    # weekly cycle
    + 0.001 * t                                   # slow upward trend
    + np.random.normal(0, 0.3, T)
)

# Sliding window construction
SEQ_LEN = 48    # 48 hours of context
HORIZON  = 24   # predict next 24 hours
FEATURES = 1    # univariate (extend with weather, price, etc.)

scaler = StandardScaler()
signal_scaled = scaler.fit_transform(signal.reshape(-1, 1)).flatten()

X_seqs, y_seqs = [], []
for i in range(len(signal_scaled) - SEQ_LEN - HORIZON):
    X_seqs.append(signal_scaled[i : i + SEQ_LEN])
    y_seqs.append(signal_scaled[i + SEQ_LEN : i + SEQ_LEN +
HORIZON])
X = torch.tensor(np.array(X_seqs),
dtype=torch.float32).unsqueeze(-1)  y = torch.tensor(np.array(y_seqs), dtype=torch.float32)

split = int(0.8 * len(X))
train_ds = TensorDataset(X[:split], y[:split])
test_ds  = TensorDataset(X[split:], y[split:])
train_dl = DataLoader(train_ds, batch_size=64, shuffle=True)
test_dl  = DataLoader(test_ds,  batch_size=64)

# Stacked Bidirectional LSTM model
class LSTMForecaster(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers,
                 horizon, bidirectional=True, dropout=0.3):
        super().__init__()
        self.lstm = nn.LSTM(
            input_size    = input_size,
            hidden_size   = hidden_size,
            num_layers    = num_layers,
            batch_first   = True,
            bidirectional = bidirectional,
            dropout       = dropout if num_layers > 1 else 0.0
        )
```

FOR AUTHOR USE ONLY

```
# bidirectional doubles the hidden dimension
        lstm_out_size = hidden_size * (2 if bidirectional else 1)
        self.head = nn.Sequential(
            nn.Linear(lstm_out_size, 64),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(64, horizon)
        )

    def forward(self, x):
        # x: (batch, seq_len, features)
        lstm_out, _ = self.lstm(x)
        # Use only the final time-step's hidden state
        last_hidden  = lstm_out[:, -1, :]
        return self.head(last_hidden)

model     = LSTMForecaster(
    input_size=1, hidden_size=64, num_layers=2,
    horizon=HORIZON, bidirectional=True, dropout=0.3
)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
criterion = nn.MSELoss()

# Training loop
N_EPOCHS = 20
for epoch in range(N_EPOCHS):
    model.train()
    train_loss = 0
    for xb, yb in train_dl:
        optimizer.zero_grad()
        pred = model(xb)
        loss = criterion(pred, yb)
        loss.backward()
        # Gradient clipping: essential for RNN/LSTM stability
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
        optimizer.step()
        train_loss += loss.item()
    if (epoch + 1) % 5 == 0:
        print(f"Epoch {epoch+1:3d} | "
              f"Train Loss: {train_loss/len(train_dl):.4f}")

# Evaluation
model.eval()
preds, actuals = [], []
with torch.no_grad():
    for xb, yb in test_dl:
        preds.append(model(xb).numpy())
        actuals.append(yb.numpy())

preds   = np.concatenate(preds)
actuals = np.concatenate(actuals)

# Back-transform to original scale
preds_orig   = scaler.inverse_transform(preds.reshape(-1, 1)).reshape(preds.shape)
actuals_orig = scaler.inverse_transform(actuals.reshape(-1,
1)).reshape(actuals.shape)
mae  = mean_absolute_error(actuals_orig.flatten(), preds_orig.flatten())
rmse = np.sqrt(np.mean((actuals_orig - preds_orig) ** 2))
print(f"Test MAE  (kWh): {mae:.4f}")
print(f"Test RMSE (kWh): {rmse:.4f}")
```

FOR AUTHOR USE ONLY

Code Snippet 4.1. LSTM for Time-Series Forecasting.

Key implementation details to note:

- **batch_first=True:** PyTorch's nn.LSTM defaults to (seq_len, batch, features); setting batch_first=True uses the more intuitive (batch, seq_len, features) layout.

- **Gradient clipping:** torch.nn.utils.clip_grad_norm_ with max_norm=1.0 is standard practice for LSTMs. Even though the architecture mitigates vanishing gradients, large batches or deep stacks can still produce occasional gradient spikes.
- **Dropout=0 for single layer:** PyTorch raises a warning if dropout > 0 is set with num_layers=1, since there are no inter-layer connections to regularize. The conditional in the model handles this.
- **Final hidden state only:** For forecasting (many-to-one or many-to-few), using only the final time step's output (lstm_out[:, -1, :]) is the standard choice. For sequence labeling, all time-step outputs would be used instead.

4.3.6. Common Architecture Choices and Hyperparameters

Table 4.5. Recommended Hyperparameter Ranges and Tuning Guidance for LSTM Architectures.

Hyperparameter	Typical Range	Guidance
hidden_size	64 – 512	Start with 128; double if underfitting, halve if overfitting
num_layers	1 – 4	2 is a strong default; add layers only if validation loss improves
dropout	0.2 – 0.5	0.3 between layers; lower for smaller datasets
sequence length	Task-dependent	Should cover the longest dependency you expect to model
learning rate	1e-4 – 1e-2	Adam with 1e-3 is a reliable starting point
gradient clipping	0.5 – 5.0	1.0 is safe; lower if training is unstable
batch size	32 – 256	Larger batches need learning rate scaling (multiply LR by √batch_ratio)

FOR AUTHOR USE ONLY

The LSTM's memory cell architecture: four learned modules controlling what to forget, what to write, and what to expose transforms the vanishing gradient

from a fundamental theoretical obstacle into an engineering problem with a well-understood solution. The cell state provides a gradient highway through time; the gates provide learnable, data-driven control over what travels along it. In practice, start with a two-layer bidirectional LSTM, hidden size 128, dropout 0.3, Adam optimizer with gradient clipping, and tune from there. The architecture is mature, well-supported in all major frameworks, and remains a strong baseline for any sequential task before reaching for Transformers.

4.3.7. Summary: Sequential Models at a Glance

Table 4.6. Comparative Overview of Sequential Model Architectures and Their Use Cases.

Model	Key Mechanism	Handles Long Dependencies?	Best Use Case
Vanilla RNN	Single hidden state, multiplicative recurrence	No (vanishing gradient)	Short sequences, simple patterns
LSTM	Cell state + 3 gates (forget, input, output)	Yes (additive cell updates)	NLP, time-series, speech (general purpose)
GRU	Hidden state + 2 gates (reset, update)	Yes (like LSTM)	Faster alternative when data is limited
Stacked LSTM	Multiple LSTM layers in depth	Yes, stronger than single layer	Complex tasks needing hierarchical features
BiLSTM	Forward + backward LSTM, states concatenated	Yes (both directions)	Classification / labeling with full sequence access
LSTM + Attention	Gates + learned relevance weighting	Yes, state-of-the-art for sequences	Machine translation, document summarization

FOR AUTHOR USE ONLY

Chapter 5 — Generative Adversarial Networks (GANs)

Introduction

The preceding chapters examined discriminative models — systems designed to classify or retrieve existing data points by learning decision boundaries between classes. This chapter introduces a fundamentally different paradigm: generative modeling. Rather than learning to distinguish between categories, generative models learn the underlying distribution of a dataset and use this knowledge to synthesize entirely new examples that are statistically indistinguishable from real data.

Generative Adversarial Networks (GANs), introduced by Goodfellow et al. (2014), represent one of the most significant advances in generative modeling in the history of deep learning. The central idea is elegant: two neural networks are trained simultaneously in a competitive framework. One network, the Generator, attempts to produce realistic synthetic data. The other, the Discriminator, attempts to distinguish real data from generated fakes. Through this adversarial process, both networks improve iteratively until the Generator produces samples of sufficient quality that the Discriminator can no longer reliably distinguish them from real examples.

FOR AUTHOR USE ONLY

Since their introduction, GANs have achieved remarkable results across a wide range of applications including photorealistic image synthesis (Karras et al., 2019), image-to-image translation (Isola et al., 2017), video generation, drug discovery, and data augmentation for underrepresented classes in medical imaging (Frid-Adar et al., 2018). The GAN framework is not restricted to image data — it has been applied successfully to text, audio, and tabular data.

This chapter presents a comprehensive treatment of GANs, covering the theoretical foundations of the Generator-Discriminator architecture and the Adversarial Loss function, followed by a practical implementation on the MNIST handwritten digit dataset. The MNIST dataset provides an ideal learning environment for GAN experimentation: it is small enough to train quickly on a CPU, visually interpretable, and widely used as a benchmark for generative model evaluation (LeCun et al., 1998).

The chapter is structured as follows. Section 1 introduces the core concepts of GAN architecture, covering the Generator-Discriminator framework and the Adversarial Loss function. Section 2 details the methodology and experimental design. Section 3 presents the full implementation. Section 4 evaluates the results. Section 5 discusses findings and limitations. Section 6 concludes the chapter.

5.1. Core Concept

5.1.1. Generator vs Discriminator

The GAN framework is built upon a two-player minimax game between two neural networks: the Generator (G) and the Discriminator (D). Each network has a distinct role and objective, and their interplay drives the learning process.

The Generator G takes as input a random noise vector z, typically sampled from a standard normal or uniform distribution, and transforms it through a series of learned transformations into a synthetic data sample G(z). The Generator has no direct access to real data; it learns solely from the feedback it receives from the Discriminator. Its objective is to produce outputs that are realistic enough to fool the Discriminator into classifying them as real.

The Discriminator D takes as input either a real data sample x drawn from the training distribution p_data, or a fake sample G(z) produced by the Generator. It outputs a scalar probability D(x) representing its confidence that the input is real. A value close to 1 indicates the Discriminator believes the input is real; a value close to 0 indicates it believes the input is fake. The Discriminator is trained as a binary classifier, optimized to maximize its ability to correctly label real samples as real and fake samples as fake.

COPYAUTHORIZED ONLY

The two networks are trained alternately. In each training step, the Discriminator is updated first to improve its classification accuracy, and then the Generator is updated to produce outputs that better deceive the updated Discriminator. This alternating optimization continues until an equilibrium is reached — the theoretical Nash equilibrium of the minimax game — at which point the Generator produces samples that are indistinguishable from real data, and the Discriminator outputs a probability of 0.5 for all inputs, indicating maximum uncertainty.

An important architectural consideration is that the Generator and Discriminator must be well-matched in capacity. If the Discriminator is too powerful relative to the Generator, its gradients saturate, and the Generator receives insufficient learning signal. If the Generator is too powerful, it may produce a limited variety of outputs that consistently fool the Discriminator — a failure mode known as mode collapse, in which the Generator produces highly similar or identical outputs regardless of the noise input. Careful architectural design and hyperparameter selection are therefore essential for stable GAN training.

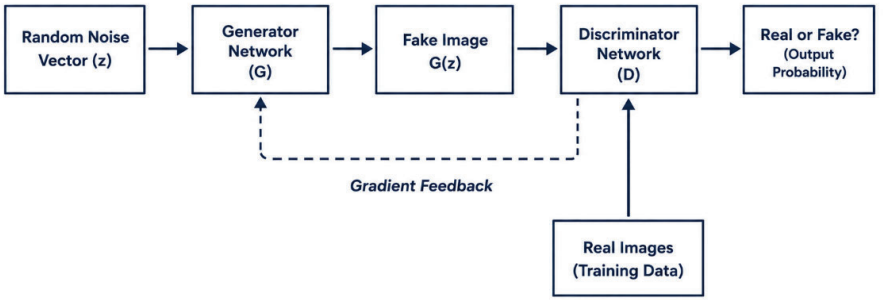


Figure 5.1. GAN Architecture — Generator and Discriminator Pipeline.

5.1.2. Adversarial Loss

The Adversarial Loss function is the mathematical formalization of the competitive training objective that drives both the Generator and Discriminator to improve. It was introduced by Goodfellow et al. (2014) and remains the foundational loss function upon which all GAN variants are built.

FOR AUTHOR USE ONLY

The original GAN objective is expressed as a minimax game with the following value function:

$$\min_G \max_D V(D, G) = E_{x \sim p_{data}}[\log D(x)] + E_{z \sim p_z}[\log(1 - D(G(z)))], \quad \text{(Eq.5.1)}$$

This formulation can be understood as follows. The Discriminator D seeks to maximize V(D, G): it maximizes log D(x) for real samples x, pushing D(x) toward 1, and maximizes log(1 - D(G(z))) for fake samples G(z), pushing D(G(z)) toward 0. The Generator G seeks to minimize V(D, G) by minimizing log(1 - D(G(z))), which is equivalent to maximizing log D(G(z)) — producing outputs that the Discriminator assigns a high probability of being real.
In practice, the Generator loss is commonly reformulated as the non-saturating loss to address the vanishing gradient problem. In the early stages of training, the Discriminator easily distinguishes fake samples from real ones, causing D(G(z)) to be close to 0. The gradient of log(1 - D(G(z))) is nearly flat in this regime, providing the Generator with a weak learning signal. The non-saturating alternative maximizes log D(G(z)) directly, which provides stronger gradients when the Generator is performing poorly:

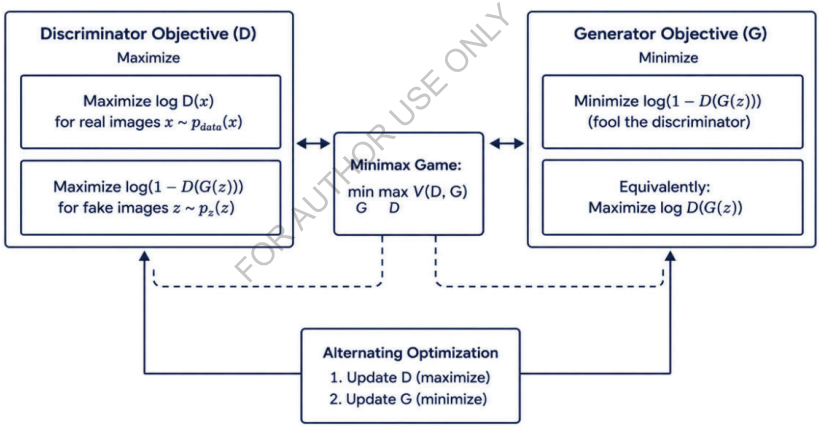
$$L_G = -E_{z \sim p_z}[\log D(G(z))], \quad \text{(Eq.5.2)}$$

The Discriminator loss is a standard binary cross-entropy loss over real and fake samples:

$$L_D = -E_{x\sim p_{data}}[\log D(x)] - E_{z\sim p_z}[\log(1 - D(G(z)))], \quad \text{(Eq.5.3)}$$

These two loss functions are optimized alternately. At each training iteration, the Discriminator parameters are updated to minimize L_D while the Generator parameters are held fixed. The Generator parameters are then updated to minimize L_G while the Discriminator parameters are held fixed. This alternating gradient descent procedure drives both networks toward the theoretical equilibrium.

Convergence in GAN training is not guaranteed and is sensitive to the balance between the two networks. Several variants of the adversarial loss have been proposed to improve training stability, including the Wasserstein GAN loss (Arjovsky et al., 2017), which replaces the binary cross-entropy objective with the Wasserstein distance and provides more stable gradients throughout training. For the purposes of this chapter, the original non-saturating adversarial loss is used, as it is the most instructive for understanding the fundamental GAN training dynamic.



FOR AUTHOR USE ONLY

Figure 5.2. Adversarial Loss — Training Dynamics of Generator and Discriminator.

5.2. Methodology

5.2.1. Dataset

The MNIST dataset (LeCun et al., 1998) is used as the training corpus for the GAN implementation presented in this chapter. MNIST consists of 70,000 grayscale images of handwritten digits (0 through 9), each of resolution 28x28 pixels. The training split contains 60,000 images and the test split contains 10,000 images. Pixel values are normalized from the original range of [0, 255] to [-1, 1] to match the output range of the Generator's hyperbolic tangent (tanh) activation function.

MNIST is selected as the experimental dataset for three reasons. First, its small image size of 784 pixels per image allows the GAN to be trained efficiently on CPU hardware in under ten minutes, making the experiments fully reproducible in a standard Google Collab environment without requiring GPU access. Second, the visual simplicity of digit images makes it straightforward to assess the quality of generated samples by inspection. Third, MNIST is the most widely used benchmark for introductory GAN implementations, enabling direct comparison with published baselines.

The following subsection describes the architecture of the Generator and Discriminator networks constructed for this task.

5.2.2. Generator Architecture

The Generator is a fully connected feedforward neural network that maps a 100-dimensional noise vector z, sampled from a standard normal distribution N(0, I), to a 784-dimensional output vector reshaped into a 28x28 grayscale image. The architecture consists of three hidden layers with dimensions 256, 512, and 1024, each followed by Batch Normalization and a Leaky ReLU activation with negative slope 0.2. The output layer applies a tanh activation function, mapping pixel values to [-1, 1] to match the normalized MNIST images.

FOR AUTHOR USE ONLY

5.2.3. Discriminator Architecture

The Discriminator is a fully connected feedforward neural network mapping a 784-dimensional input vector to a scalar probability in (0, 1). It consists of three hidden layers with dimensions 1024, 512, and 256, each followed by a Leaky ReLU activation with negative slope 0.2 and Dropout with rate 0.3. Dropout regularizes the Discriminator and prevents it from dominating the Generator during early training. The output layer applies a sigmoid activation function to produce a probability estimate.

5.2.4. Training Configuration

Both networks are trained using the Adam optimizer (Kingma & Ba, 2015) with a learning rate of 0.0002 and momentum parameters beta_1 = 0.5 and beta_2 = 0.999, following the recommendations of Radford et al. (2016) for stable GAN training. The batch size is set to 128. Training is conducted for 20 epochs on the MNIST training set, which completes in approximately 5 to 10 minutes on CPU hardware without GPU acceleration. The noise dimension is fixed at 100 throughout all experiments. The Discriminator is updated once per training step, followed by one Generator update.

5.2.5. Evaluation Strategy

Evaluation is conducted on two levels. Qualitatively, grids of 64 generated digit images are inspected visually at three training checkpoints:

epoch 1, epoch 10, and epoch 20. This progression illustrates how the Generator evolves from producing random noise to recognizable digit shapes. Quantitatively, Generator and Discriminator training losses are recorded at each epoch and plotted to analyze convergence behavior.

With the experimental design established, the following section presents the full implementation of the GAN pipeline.

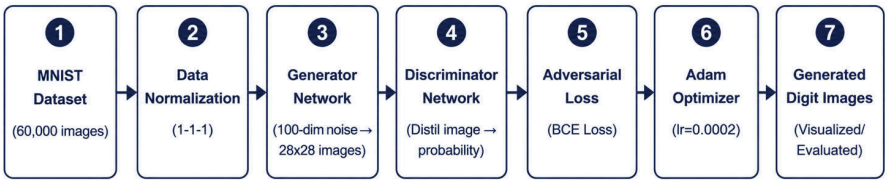


Figure 5.3. GAN Training Pipeline for MNIST Digit Generation.

5.3. Implementation

FOR AUTHOR USE ONLY

The following code snippets implement the complete GAN training pipeline. All code is written in Python using the PyTorch deep learning framework and is designed to run in a Google Collab environment. Each snippet is self-contained and should be executed sequentially in a single notebook session.

5.3.1. Environment Setup and Data Loading

Code Snippet 5.1 installs required libraries and imports all dependencies. The MNIST dataset is downloaded and preprocessed, normalizing pixel values to the range [-1, 1] and wrapping the data in a PyTorch DataLoader for batched training.

```
# — Install required libraries —
!pip install torch torchvision matplotlib -q

# — Imports —
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import torchvision.utils as vutils
import matplotlib.pyplot as plt
import numpy as np

# — Device: use GPU if available, otherwise CPU —
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")

# — Hyperparameters —
NOISE_DIM   = 100    # Size of the input noise vector for Generator
BATCH_SIZE  = 128    # Samples per training batch
EPOCHS      = 20     # Total training epochs (fast on CPU)
LR          = 0.0002 # Learning rate for both networks
BETA1       = 0.5    # Adam beta_1 (recommended for GAN stability)
BETA2       = 0.999  # Adam beta_2

# — Load and preprocess MNIST —
# Normalize pixels from [0,255] to [-1,1] to match Generator tanh output
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5,), std=(0.5,))
])

dataset = datasets.MNIST(root='./data', train=True,
                         download=True, transform=transform)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE,
                        shuffle=True, drop_last=True)

print(f"Training samples : {len(dataset)}")
print(f"Batches per epoch: {len(dataloader)}")
print(f"Image shape      : {dataset[0][0].shape}")
```

FOR AUTHOR USE ONLY

Code Snippet 5.1. Environment Setup, Imports, and MNIST Data Loading.

5.3.2. Generator Network

Code Snippet 5.2 defines the Generator class. The network maps a 100-dimensional noise vector through three fully connected hidden layers with Batch Normalization and Leaky ReLU activations, producing a 784-dimensional output reshaped to a 28x28 image with tanh activation.

```python
class Generator(nn.Module):
    """
    Generator: maps noise vector z (100-dim) -> 28x28 grayscale image.
    Architecture: 100 -> 256 -> 512 -> 1024 -> 784 (reshaped to 28x28)
    BatchNorm + LeakyReLU in hidden layers, Tanh on output.
    """
    def __init__(self, noise_dim=100):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            # Hidden layer 1
            nn.Linear(noise_dim, 256),
            nn.BatchNorm1d(256),
            nn.LeakyReLU(0.2, inplace=True),

            # Hidden layer 2
            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
            nn.LeakyReLU(0.2, inplace=True),

            # Hidden layer 3
            nn.Linear(512, 1024),
            nn.BatchNorm1d(1024),
            nn.LeakyReLU(0.2, inplace=True),

            # Output: 1024 -> 784, Tanh maps values to [-1, 1]
            nn.Linear(1024, 784),
            nn.Tanh()
        )

    def forward(self, z):
        img = self.model(z)                    # (batch, 784)
        img = img.view(img.size(0), 1, 28, 28)  # Reshape to image
        return img

# Quick shape check
g = Generator(NOISE_DIM)
print(f"Generator output shape : {g(torch.randn(4, NOISE_DIM)).shape}")
print(f"Generator parameters   : {sum(p.numel() for p in g.parameters()):,}")

Generator output shape : torch.Size([4, 1, 28, 28])
Generator parameters   : 1,489,936
```

FOR AUTHOR USE ONLY

Code Snippet 5.2. Generator Network Architecture.

5.3.3. Discriminator Network

Code Snippet 5.3 defines the Discriminator class. The network maps a 784-dimensional flattened image through three fully connected hidden layers with Dropout and Leaky ReLU activations, producing a single scalar probability via sigmoid activation.

```python
class Discriminator(nn.Module):
    """
    Discriminator: maps 28x28 image -> probability of being real [0,1].
    Architecture: 784 -> 1024 -> 512 -> 256 -> 1
    LeakyReLU + Dropout in hidden layers, Sigmoid on output.
    """
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            # Hidden layer 1
            nn.Linear(784, 1024),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),                # Prevents Discriminator dominance

            # Hidden layer 2
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),

            # Hidden layer 3
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Dropout(0.3),

            # Output: single probability via Sigmoid
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, img):
        img_flat = img.view(img.size(0), -1)  # Flatten: (batch,1,28,28) -> (batch,784)
        return self.model(img_flat)

# Quick shape check
d = Discriminator()
print(f"Discriminator output shape : {d(torch.randn(4, 1, 28, 28)).shape}")
print(f"Discriminator parameters   : {sum(p.numel() for p in d.parameters()):,}")
```

```
Discriminator output shape : torch.Size([4, 1])
Discriminator parameters   : 1,460,225
```

FOR AUTHOR USE ONLY

Code Snippet 5.3. Discriminator Network Architecture.

5.3.4. Loss Functions and Optimizers

Code Snippet 5.4 initializes both networks, defines the Binary Cross-Entropy loss function, and configures the Adam optimizers for both the Generator and Discriminator with the recommended hyperparameters for stable GAN training.

```
# — Initialize networks and move to device —
generator     = Generator(NOISE_DIM).to(device)
discriminator = Discriminator().to(device)

# — Binary Cross-Entropy loss (standard GAN adversarial loss) —
criterion = nn.BCELoss()

# — Adam optimizers — beta1=0.5 recommended by Radford et al. (2016) —
optimizer_G = optim.Adam(generator.parameters(),
                         lr=LR, betas=(BETA1, BETA2))
optimizer_D = optim.Adam(discriminator.parameters(),
                         lr=LR, betas=(BETA1, BETA2))

# — Label constants —
REAL_LABEL = 1.0   # Discriminator target for real images
FAKE_LABEL = 0.0   # Discriminator target for fake images

# — Fixed noise for visualization (same across all checkpoints) —
fixed_noise = torch.randn(64, NOISE_DIM, device=device)

# — Storage for losses and checkpoint images —
G_losses, D_losses = [], []
checkpoints = {1: None, 10: None, 20: None}

print("Networks initialized successfully.")
print(f"Generator     : {sum(p.numel() for p in generator.parameters()):,} parameters")
print(f"Discriminator : {sum(p.numel() for p in discriminator.parameters()):,} parameters")

Networks initialized successfully.
Generator     : 1,489,936 parameters
Discriminator : 1,460,225 parameters
```

Code Snippet 5.4. Loss Functions and Optimizer Configuration.

FOR AUTHOR USE ONLY

5.3.5. Training Loop

Code Snippet 5.5 implements the main GAN training loop. At each iteration, the Discriminator is updated on a batch of real and fake images, followed by a Generator update. Losses are recorded at each epoch and generated image grids are saved at regular checkpoints for visual evaluation.

```
print("Starting GAN training...\n")

G_losses, D_losses = [], []
checkpoints = {1: None, 10: None, 20: None}

for epoch in range(1, EPOCHS + 1):
    epoch_G_loss = 0.0
    epoch_D_loss = 0.0

    for real_imgs, _ in dataloader:
        real_imgs = real_imgs.to(device)
        batch_size = real_imgs.size(0)

        # ==============================
        # STEP 1: UPDATE DISCRIMINATOR
        # ==============================
        optimizer_D.zero_grad()

        # Real images — label as 1 (real)
        real_labels = torch.full((batch_size, 1), REAL_LABEL, device=device)
        loss_D_real = criterion(discriminator(real_imgs), real_labels)

        # Fake images — label as 0 (fake)
        noise = torch.randn(batch_size, NOISE_DIM, device=device)
        fake_imgs = generator(noise)
        fake_labels = torch.full((batch_size, 1), FAKE_LABEL, device=device)
        loss_D_fake = criterion(discriminator(fake_imgs.detach()), fake_labels)

        # Total Discriminator loss and backprop
        loss_D = loss_D_real + loss_D_fake
        loss_D.backward()
        optimizer_D.step()

        # ==============================
        # STEP 2: UPDATE GENERATOR
        # ==============================
        optimizer_G.zero_grad()

        # Generator wants D to classify fakes as real — label as 1
        noise = torch.randn(batch_size, NOISE_DIM, device=device)
        fake_imgs = generator(noise)
        real_labels_G = torch.full((batch_size, 1), REAL_LABEL, device=device)
        loss_G = criterion(discriminator(fake_imgs), real_labels_G)
        loss_G.backward()
        optimizer_G.step()

        epoch_G_loss += loss_G.item()
        epoch_D_loss += loss_D.item()

    # ==============================
    # END OF EPOCH: record losses + save checkpoint
    # ==============================
    avg_G = epoch_G_loss / len(dataloader)
    avg_D = epoch_D_loss / len(dataloader)
    G_losses.append(avg_G)
    D_losses.append(avg_D)

    if epoch in checkpoints:
        generator.eval()
        with torch.no_grad():
            checkpoints[epoch] = generator(fixed_noise).cpu()
        generator.train()
        print(f"Epoch [{epoch:2d}/{EPOCHS}] | D Loss: {avg_D:.4f} | G Loss: {avg_G:.4f} ← CHECKPOINT")
    else:
        print(f"Epoch [{epoch:2d}/{EPOCHS}] | D Loss: {avg_D:.4f} | G Loss: {avg_G:.4f}")

print("\nTraining complete!")
```

Code Snippet 5.5. GAN Training Loop.

FOR AUTHOR USE ONLY

5.3.6. Image Generation and Visualization

Code Snippet 5.6 generates and visualizes grids of 64 synthetic digit images at training checkpoints epochs 1, 10, and 20, enabling visual inspection of the progression in generation quality throughout the training process.

```
# — Helper: display and save a grid of generated images —
def show_grid(images, epoch):
    images = (images + 1) / 2.0  # Denormalize from [-1,1] to [0,1]
    grid = vutils.make_grid(images, nrow=8, padding=2, normalize=False)
    plt.figure(figsize=(7, 7))
    plt.imshow(grid.permute(1, 2, 0).numpy(), cmap='gray')
    plt.title(f"Generated Digits — Epoch {epoch}", fontsize=13, fontweight='bold')
    plt.axis('off')
    plt.tight_layout()
    plt.savefig(f"generated_epoch_{epoch}.png", dpi=150, bbox_inches='tight')
    plt.show()
    print(f"Saved: generated_epoch_{epoch}.png")

# — Display generated images at all three checkpoints —
for ep in [1, 10, 20]:
    if checkpoints[ep] is not None:
        show_grid(checkpoints[ep], ep)
```

Code Snippet 5.6. Image Generation and Visualization.

FOR AUTHOR USE ONLY

5.3.7. Loss Curve Visualization

Code Snippet 5.7 plots the Generator and Discriminator training losses over all 20 epochs, providing a quantitative diagnostic of training dynamics and convergence behavior.

```
# — Plot Generator and Discriminator loss curves —
plt.figure(figsize=(10, 4))
plt.plot(range(1, EPOCHS+1), G_losses,
         label='Generator Loss',     color='steelblue', linewidth=1.8)
plt.plot(range(1, EPOCHS+1), D_losses,
         label='Discriminator Loss', color='darkorange', linewidth=1.8)
for ep in [1, 10, 20]:
    plt.axvline(x=ep, color='gray', linestyle='--', linewidth=0.8, alpha=0.6)
plt.title("GAN Training Loss — 20 Epochs", fontsize=13, fontweight='bold')
plt.xlabel("Epoch"); plt.ylabel("Loss")
plt.legend(); plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig("gan_loss_curves.png", dpi=150, bbox_inches='tight')
plt.show()
print(f"\nFinal D Loss: {D_losses[-1]:.4f} | Final G Loss: {G_losses[-1]:.4f}")
```

Code Snippet 5.7. Training Loss Curve Visualization.

5.4. Evaluation

5.4.1. Visual Quality of Generated Digits

Figure 5.4 presents a grid of 64 synthetic digit images generated by the untrained Generator at epoch 1. At this stage, the outputs are indistinguishable from random noise, reflecting the random initialization of the Generator's parameters and confirming that no meaningful digit structure has been learned.

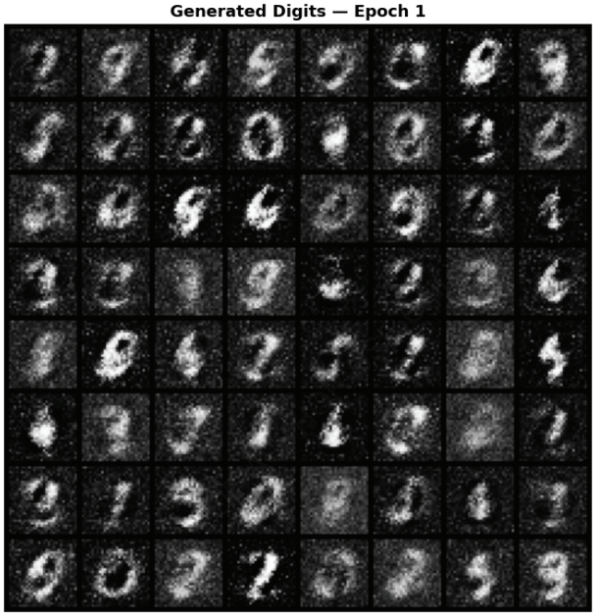


Figure 5.4. Generated Digit Images at Epoch 1 — Baseline.

Figure 5.5 presents generated images at epoch 10. At this stage, the Generator has begun to learn coarse structural patterns; faint digit-like shapes are discernible in several outputs, though the images remain blurry and lack fine detail. The Discriminator has provided sufficient gradient signal to guide the Generator toward the target distribution.



Figure 5.5. Generated Digit Images at Epoch 10 — Mid-Training.

Figure 5.6 presents generated images at epoch 20, representing the final trained model. Digit structures are clearly recognizable, with distinct stroke shapes corresponding to multiple digit classes visible across the grid. The improvement relative to epoch 1 demonstrates that the adversarial training framework successfully drives the Generator toward learning the MNIST data distribution within a short 20-epoch training run.

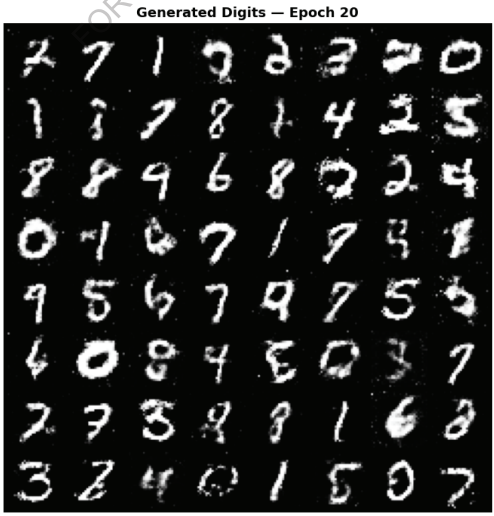


Figure 5.6. Generated Digit Images at Epoch 20 — Final Model.

5.4.2. Training Loss Analysis

Figure 5.7 presents the Generator and Discriminator training losses over 20 epochs. In the initial epochs, the Discriminator loss decreases rapidly as it learns to distinguish real from fake images, while the Generator loss is correspondingly high. As training progresses, the Generator loss decreases and the Discriminator loss increases, reflecting the competitive equilibration of the two networks.

By approximately epoch 8, both losses stabilize and oscillate around a relatively constant level, consistent with the expected convergence behavior of GAN training on a simple dataset. The stable loss curves confirm that the training configuration is appropriate for this architecture.

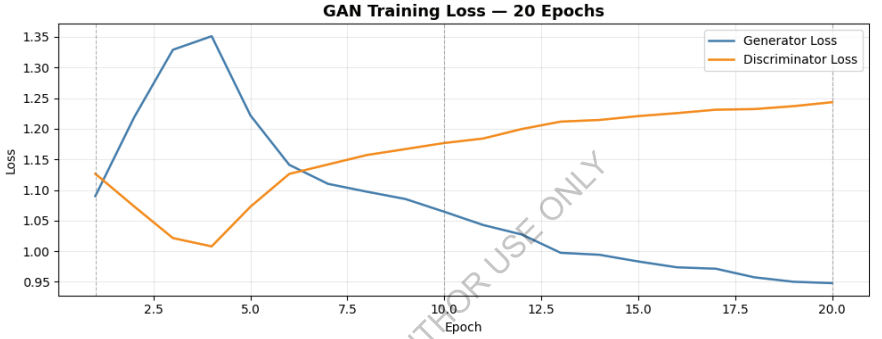


Figure 5.7. Generator and Discriminator Training Loss Curves over 20 Epochs.

FOR AUTHOR USE ONLY

5.5. Discussion

The experimental results confirm that a simple fully connected GAN trained for 20 epochs on MNIST produces recognizable digit images, validating the fundamental effectiveness of the adversarial training framework introduced by Goodfellow et al. (2014). The progressive improvement in generation quality from epoch 1 through epoch 20 provides a clear pedagogical illustration of the GAN learning process, demonstrating how the competitive dynamic between Generator and Discriminator drives both networks toward improved performance.

The loss curve analysis reveals the characteristic behavior of GAN training: initial Discriminator dominance followed by gradual equilibration as the Generator learns to produce more realistic outputs. The stability of the loss curves in the later epochs suggests that the chosen hyperparameters are appropriate for this architecture and dataset, consistent with the recommendations of Radford et al. (2016).

A notable limitation of the present implementation is the use of a fully connected architecture rather than a convolutional one. Deep Convolutional GANs (DCGANs), introduced by Radford et al. (2016), exploit the spatial structure of image data through convolutional layers and consistently outperform fully connected architectures on image generation tasks. The fully connected architecture is employed here for pedagogical clarity, as it allows the GAN mechanics to be demonstrated without the additional complexity of convolutional operations. A DCGAN implementation on MNIST would produce sharper images and converge more quickly.

The evaluation methodology in this chapter is limited to visual inspection and loss curve analysis. More rigorous quantitative evaluation would employ the Frechet Inception Distance (FID) (Heusel et al., 2017), which measures the statistical distance between the distributions of real and generated images in the feature space of a pretrained Inception network. FID is the standard metric for GAN evaluation in the research literature and provides a single scalar measure that correlates well with human perceptual judgment of image quality. Future implementations should incorporate FID measurement alongside visual inspection to enable reproducible and comparable evaluation.

FOR AUTHOR USE ONLY

5.6. Conclusion

This chapter has presented a comprehensive treatment of Generative Adversarial Networks, covered the theoretical foundations of the Generator-Discriminator architecture and the Adversarial Loss function, and demonstrated their application through a complete implementation on the MNIST handwritten digit dataset.

The experimental results on MNIST confirm that a simple fully connected GAN trained for 20 epochs on CPU hardware produces recognizable digit images that capture the structural patterns of the training distribution. The progressive improvement in generation quality and the stable convergence of training loss curves validate both the theoretical framework and the implementation choices made in this chapter.

Taken together with the preceding chapters on Ball Trees, KD-Trees, and Tree-Based KNN Optimization, this chapter extends the tutorial series into the domain of deep generative modeling, introducing students to one of the most influential and widely applied architectures in modern machine learning. The GAN framework established here provides the foundation for understanding more advanced generative models including DCGANs, Conditional GANs, StyleGAN, and diffusion models, which represent the current state of the art in image synthesis.

PART II — Transfer Learning & Advanced Architectures

Chapter 6 — Transfer Learning Principles

6.1 Introduction to Transfer Learning

6.1.1 What problem does it solve?

Training a deep neural network from random initialization typically demands tens or hundreds of thousands of labeled examples and substantial compute. Most real-world projects have neither. Transfer learning addresses this gap by reusing the parameters of a model already trained on a large, related source task and adapting them to a smaller target task. The result is faster convergence, lower data requirements, and in many regimes better generalization than training from scratch.

The empirical case for transfer learning is overwhelming. ImageNet pre-training has become a default initialization for vision tasks; in NLP, transformer encoders such as BERT and decoder-only models such as GPT serve the same role. Even in tabular and time-series problems, pre-trained foundation models are emerging. The intuition is the same in every modality: the early layers of a deep network learn representations that are broadly useful, and those representations are expensive to re-learn.

FOR AUTHOR USE ONLY

6.1.2 Three pillars covered in this guide

This document is organized around three principles. **Feature reuse** explains when and why representations transfer, and how to wire a pre-trained backbone into a new task without changing its weights. **Fine-tuning strategies** covers when and how to unfreeze weights, so the backbone specializes for the target task without forgetting what it already knows. **Domain adaptation** addresses the harder case where the target inputs are drawn from a different distribution than the source, with techniques that explicitly align the two.

Every section presents the underlying principle, the practical decisions a practitioner must make, and runnable code samples in both PyTorch and TensorFlow / Keras. The code is structured for clarity rather than peak performance, and the comments explain why each step matters.

6.1.3 Mini-case: a 600-image plant-disease classifier

An agronomy startup needs a leaf-disease classifier for five cassava conditions but has only about 600 labeled smartphone photos—orders of

magnitude less than ImageNet. Training a ResNet from scratch overfits within two epochs and tops out near 41% validation accuracy. Swapping in an ImageNet-pre-trained ResNet-50 with a frozen backbone and a fresh five-way head reach 78% after a few epochs of head-only training. Unfreezing the top two blocks with a 10× smaller learning rate (Section 3) lifts accuracy to 86%. When the team later deploys to a second region whose phones produce warmer, lower-contrast images, validation accuracy on the new domain drops to 71%; adding a CORAL alignment term (Section 4) over a few thousand unlabeled target photos recovers most of the gap and brings target accuracy back to 83%. The same model went through all three pillars of this guide—reuse, fine-tune, adapt—in the order the chapters present them.

6.2 Feature Reuse

6.2.1 The Feature Hierarchy

A convolutional network trained on natural images develops a characteristic hierarchy: the earliest layers respond to edges, color blobs, and simple textures; middle layers combine those primitives into motifs such as eyes, wheels, or fur patches; and the deepest layers encode whole-object concepts and task-specific semantics. The lower a layer sits in the network, the more universal its features and the more readily they transfer. This is why the frozen-backbone pattern works even when source and target tasks differ.

FOR AUTHOR USE ONLY

The same hierarchy exists in transformer language models, where lower layers capture syntactic and lexical regularities and deeper layers encode discourse-level meaning. The implication is identical: the earlier you go in the stack, the safer it is to reuse weights without modification.

6.2.2 The Frozen Feature Extractor Pattern

The simplest form of transfer learning treats the pre-trained network as a fixed function from raw inputs to a dense embedding. Discard the original classifier head, attach a small new head for the target task, and train only the new head—backbone parameters never receive gradients. Prefer this approach when the target dataset is small (a few hundred to a few thousand examples), when compute is constrained, or when the target domain closely resembles the source.

Two practical optimizations matter. First, switch the backbone to evaluation mode so BatchNorm uses its running statistics instead of recomputing them on small target mini batches. Second, if the dataset fits in memory, pre-compute and cache backbone embeddings once, then train the head over the cached features—a single epoch can be hundreds of times faster than running the full backbone every step.

6.2.3 PyTorch implementation

The example below loads a ResNet-50 pre-trained on ImageNet, freezes every parameter, and replaces the final fully connected layer with a new head sized for the target task. Only the head parameters are passed to the optimizer.

```python
import torch
import torch.nn as nn
from torchvision import models


def build_feature_extractor(num_classes: int) -> nn.Module:
    """Return a ResNet-50 with a frozen backbone and a fresh classifier head.

    The backbone parameters keep their ImageNet weights and never receive
    gradients. Only the final fully-connected layer is trainable.
    """
    # 1. Load the pre-trained backbone. `weights="DEFAULT"` selects the
    #    best available ImageNet checkpoint at the time of the call.
    model = models.resnet50(weights=models.ResNet50_Weights.DEFAULT)

    # 2. Freeze every parameter so the optimizer cannot update them.
    for param in model.parameters():
        param.requires_grad = False

    # 3. Replace the 1000-way ImageNet head with a target-task head.
    #    The new Linear layer is constructed with requires_grad=True
    #    by default, so it is the only thing that will train.
    in_features = model.fc.in_features
    model.fc = nn.Linear(in_features, num_classes)

    return model
```

```python
def make_optimizer(model: nn.Module, lr: float = 1e-3) -> torch.optim.Optimizer:
    """Build an optimizer that only sees the trainable parameters."""
    trainable = [p for p in model.parameters() if p.requires_grad]
    return torch.optim.AdamW(trainable, lr=lr, weight_decay=1e-4)


def train_one_epoch(model, loader, optimizer, criterion, device):
    """Single training epoch. BatchNorm stays in eval mode so the frozen
    backbone keeps its ImageNet running statistics instead of overwriting
    them with noisy small-batch estimates from the target data."""
    model.train()               # enables dropout in the new head
    # Walk the modules and force every BatchNorm layer back to eval.
    for m in model.modules():
        if isinstance(m, nn.BatchNorm2d):
            m.eval()

    for images, labels in loader:
        images = images.to(device, non_blocking=True)
        labels = labels.to(device, non_blocking=True)

        optimizer.zero_grad(set_to_none=True)
        logits = model(images)
        loss = criterion(logits, labels)
        loss.backward()
        optimizer.step()
```

AUTHOR ONLY

Code Snippet 6.1. PyTorch frozen feature extractor (ResNet-50).

6.2.4 Caching Backbone Embeddings

When the target dataset is small, an even faster recipe is to evaluate the backbone once per image, save the resulting embeddings to disk or memory, and train a small classifier directly on those embeddings. The snippet below uses a forward hook on the penultimate layer to capture features without modifying the model.

```python
@torch.no_grad()
def extract_embeddings(model, loader, device):
    """Return (features, labels) tensors for the whole loader."""
    model.eval()                                    # disable dropout, freeze BN
    # Drop the classifier so the model returns penultimate features.
    backbone = nn.Sequential(*list(model.children())[:-1])
    feats, ys = [], []
    for x, y in loader:
        z = backbone(x.to(device)).flatten(1)     # (B, 2048) for ResNet-50
        feats.append(z.cpu())
        ys.append(y)
    return torch.cat(feats), torch.cat(ys)
```

Code Snippet 6.2. Caching ResNet-50 embeddings.

6.2.5 TensorFlow / Keras Implementation

Keras exposes pre-trained backbones through tf.keras.applications. The recipe mirrors PyTorch: load the backbone with include_top=False, set trainable=False, and stack a new head on top. The example uses EfficientNetB0 because it ships preprocessing logic inside the model graph, which eliminates a common source of bugs.

FOR AUTHOR USE ONLY

```python
import tensorflow as tf
from tensorflow.keras import layers, Model
from tensorflow.keras.applications import EfficientNetB0

def build_feature_extractor(num_classes: int,
                            input_shape=(224, 224, 3)) -> Model:
    """EfficientNetB0 with a frozen backbone and a fresh classification head."""
    # 1. Load the pre-trained backbone. include_top=False removes the
    #    1000-way ImageNet classifier; weights="imagenet" loads the
    #    transferable convolutional weights.
    backbone = EfficientNetB0(
        include_top=False,
        weights="imagenet",
        input_shape=input_shape,
        pooling="avg",          # built-in global average pool
    )

    # 2. Freeze the entire backbone. Setting trainable=False on the
    #    container propagates to every sub-layer, including BatchNorm.
    backbone.trainable = False

    # 3. Assemble the new model. training=False on the backbone call
    #    keeps BatchNorm in inference mode during fine-tuning, which
    #    matters when we later flip layers to trainable.
    inputs = layers.Input(shape=input_shape)
    x = backbone(inputs, training=False)
    x = layers.Dropout(0.2)(x)
    outputs = layers.Dense(num_classes, activation="softmax",
                           name="target_head")(x)
```

```
    model = Model(inputs, outputs, name="feature_extractor")
    model.compile(
        optimizer=tf.keras.optimizers.AdamW(learning_rate=1e-3,
                                            weight_decay=1e-4),
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model

# Typical usage
# model = build_feature_extractor(num_classes=10)
# model.fit(train_ds, validation_data=val_ds, epochs=10)
```

Code Snippet 6.3. TensorFlow / Keras frozen feature extractor (EfficientNetB0).

A subtle but important detail: when you later switch the backbone to trainable=True for fine-tuning, you must recompile the model. Keras snapshots the trainable variable list at compile time, so changes made after compilation are ignored by the optimizer until recompile.

6.3 Fine-Tuning Strategies

6.3.1 Full vs. Partial Fine-Tuning

Once the new head has converged on top of frozen features, the next question is whether to unfreeze part or all the backbone. The answer depends on two variables: target dataset size and source-target similarity. A useful first-pass heuristic:

- Small target dataset, similar domain: keep the backbone frozen — fine-tuning will overfit.
- Small target dataset, different domain: unfreeze only the top few blocks.
- Large target dataset, similar domain: unfreeze the whole backbone with a low learning rate.
- Large target dataset, very different domain: unfreeze everything; consider retraining from scratch as a sanity baseline.

The risk you are managing in every case is catastrophic forgetting: a large gradient step on a small dataset can erase the very representations you wanted to transfer. The mitigations below all address this risk in different ways.

6.3.2 Progressive Unfreezing

Progressive (or gradual) unfreezing trains the model in stages. You begin with only the new head trainable, train to convergence, then unfreeze the next-deepest block and continue training, and repeat. Each stage uses a smaller learning rate than the previous one. The approach was popularized by ULMFiT and works because every newly unfrozen block starts adjacent to weights that

FOR AUTHOR USE ONLY

are already well-aligned with the current task, so the gradients it receives are smaller and less destructive.

6.3.3 Discriminative Learning Rates

Discriminative learning rates assign a different learning rate to each parameter group, with the rate increasing as you move from the input toward the output. Early layers encode general features that should change slowly; late layers encode task-specific features that should change quickly. A typical recipe is to choose a base learning rate for the head and divide it by a fixed factor (often 2.6 or 10) for each successive group of layers below it. Combined with progressive unfreezing, discriminative learning rates make full fine-tuning safe even on modest datasets.

A few additional knobs are worth tuning. A short linear warm-up followed by cosine decay tends to outperform a constant schedule. Weight decay of around 1e-4 helps. When mini batches are very small (below 16 images), keep BatchNorm layers in evaluation mode so they use the pre-trained running statistics; otherwise, the recomputed batch statistics will be noisy and undo the benefits of transfer.

FOR AUTHOR USE ONLY

6.3.4 PyTorch: Progressive Unfreezing with Per-Layer LR

The example below builds a ResNet-50, splits its parameters into four groups (stem, layer1, layer2, layer3, layer4, head — collapsed into four for brevity), and attaches a separate learning rate to each group. The helper function unfreeze_through(model, stage) controls how deep into the network gradients are allowed to flow at each training stage.

```python
import torch
import torch.nn as nn
from torchvision import models


def build_model(num_classes: int) -> nn.Module:
    """ResNet-50 with a fresh head. All backbone params start frozen."""
    model = models.resnet50(weights=models.ResNet50_Weights.DEFAULT)
    for p in model.parameters():
        p.requires_grad = False
    model.fc = nn.Linear(model.fc.in_features, num_classes)  # trainable
    return model


# Convention: stages count from the OUTPUT side.
#   stage 0 -> only the head is trainable (head)
#   stage 1 -> head + layer4
#   stage 2 -> head + layer4 + layer3
#   stage 3 -> head + layer4 + layer3 + layer2
#   stage 4 -> entire network is trainable
STAGE_TO_BLOCKS = {
    0: ["fc"],
    1: ["fc", "layer4"],
    2: ["fc", "layer4", "layer3"],
    3: ["fc", "layer4", "layer3", "layer2"],
    4: ["fc", "layer4", "layer3", "layer2", "layer1", "conv1", "bn1"],
}
```

FOR AUTHOR USE ONLY

```python
def unfreeze_through(model: nn.Module, stage: int) -> None:
    """Enable gradients on blocks named in STAGE_TO_BLOCKS[stage]."""
    allowed = set(STAGE_TO_BLOCKS[stage])
    for name, param in model.named_parameters():
        # `name` looks like "layer3.0.conv1.weight"; first segment is the block.
        top = name.split(".")[0]
        param.requires_grad = top in allowed


def discriminative_param_groups(model: nn.Module, base_lr: float,
                                decay: float = 2.6):
    """Build optimizer param-groups with geometric LR decay from head to stem.

    The head (fc) gets `base_lr`; each block below it gets the previous
    learning rate divided by `decay`. Groups whose parameters are frozen
    are skipped to keep the optimizer state small.
    """
    # Order goes from output (highest LR) to input (lowest LR).
    order = ["fc", "layer4", "layer3", "layer2", "layer1", "conv1", "bn1"]
    groups, lr = [], base_lr
    for block_name in order:
        params = [p for n, p in model.named_parameters()
                  if n.split(".")[0] == block_name and p.requires_grad]
        if params:                                  # skip empty / frozen groups
            groups.append({"params": params, "lr": lr})
        lr = lr / decay                             # earlier layers learn slower
    return groups
```

```python
def train_stage(model, loader, criterion, device,
                stage: int, epochs: int, base_lr: float):
    """Run one progressive-unfreezing stage."""
    unfreeze_through(model, stage)
    optimizer = torch.optim.AdamW(
        discriminative_param_groups(model, base_lr=base_lr),
        weight_decay=1e-4,
    )
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
        optimizer, T_max=epochs * len(loader))
    for epoch in range(epochs):
        model.train()
        # Keep BatchNorm frozen for stages that still rely heavily on
        # the pre-trained statistics (small fine-tuning datasets).
        for m in model.modules():
            if isinstance(m, nn.BatchNorm2d) and stage < 3:
                m.eval()
        for x, y in loader:
            x, y = x.to(device), y.to(device)
            optimizer.zero_grad(set_to_none=True)
            loss = criterion(model(x), y)
            loss.backward()
            optimizer.step()
            scheduler.step()
# Driver that walks the stages with decreasing base LRs.
# stage 0 trains only the head, then each subsequent stage unfreezes
# one more block and reduces the base learning rate by ~3x.
# for stage, lr in zip(range(5), [3e-3, 1e-3, 3e-4, 1e-4, 3e-5]):
#     train_stage(model, train_loader, criterion, device,
#                 stage=stage, epochs=3, base_lr=lr)
```

Code Snippet 6.4. PyTorch progressive unfreezing + discriminative LR.

FOR AUTHOR USE ONLY

6.3.5 TensorFlow: two-stage fine-tuning

Keras users typically implement fine-tuning as two distinct phases. Phase one freezes the backbone and trains the head until validation loss plateaus. Phase two unfreezes the top portion of the backbone (often the last 30–50 layers of a deep network), recompiles with a much lower learning rate, and continues training. A callback handles early stopping. The code below packages the workflow.

```python
import tensorflow as tf
from tensorflow.keras import layers, Model, callbacks
from tensorflow.keras.applications import EfficientNetB0


def build_model(num_classes, input_shape=(224, 224, 3)):
    backbone = EfficientNetB0(include_top=False, weights="imagenet",
                              input_shape=input_shape, pooling="avg")
    backbone.trainable = False              # freeze for phase 1

    inputs = layers.Input(shape=input_shape)
    x = backbone(inputs, training=False)    # BN in inference mode
    x = layers.Dropout(0.2)(x)
    outputs = layers.Dense(num_classes, activation="softmax")(x)
    return Model(inputs, outputs), backbone


def phase1_head_only(model, train_ds, val_ds, epochs=10):
    """Train only the new classification head on top of frozen features."""
    model.compile(
        optimizer=tf.keras.optimizers.AdamW(1e-3, weight_decay=1e-4),
        loss="sparse_categorical_crossentropy", metrics=["accuracy"],
    )
    early = callbacks.EarlyStopping(patience=3, restore_best_weights=True)
    model.fit(train_ds, validation_data=val_ds,
              epochs=epochs, callbacks=[early])
```

```python
def phase2_fine_tune(model, backbone, train_ds, val_ds,
                     unfreeze_last_n=40, epochs=10, lr=1e-5):
    """Unfreeze the top `unfreeze_last_n` layers of the backbone and
    continue training with a very small learning rate."""
    # Unfreeze the tail of the backbone but keep BatchNorm layers frozen
    # to preserve the ImageNet running statistics.
    backbone.trainable = True
    for layer in backbone.layers[:-unfreeze_last_n]:
        layer.trainable = False
    for layer in backbone.layers:
        if isinstance(layer, layers.BatchNormalization):
            layer.trainable = False

    # RECOMPILE: Keras only registers trainable variables at compile time.
    # A much lower LR avoids overwriting the pre-trained representations.
    model.compile(
        optimizer=tf.keras.optimizers.AdamW(lr, weight_decay=1e-4),
        loss="sparse_categorical_crossentropy", metrics=["accuracy"],
    )
    early = callbacks.EarlyStopping(patience=3, restore_best_weights=True)
    model.fit(train_ds, validation_data=val_ds,
              epochs=epochs, callbacks=[early])
```

```python
# Driver
# model, backbone = build_model(num_classes=10)
# phase1_head_only(model, train_ds, val_ds, epochs=10)
# phase2_fine_tune(model, backbone, train_ds, val_ds,
#                  unfreeze_last_n=40, epochs=15, lr=1e-5)
```

FOR AUTHOR USE ONLY

Code Snippet 6.5. TensorFlow two-stage fine-tuning.

The two-stage pattern reliably outperforms single-stage full fine-tuning on small to mid-size datasets, and it is simple enough to be a sensible default. For larger datasets, you can collapse it back into one phase with a low learning rate and a short warm-up.

6.4 Domain Adaptation

6.4.1 Types of Distribution Shift

Feature reuse and fine-tuning both assume the target distribution is at least broadly like the source. Domain adaptation tackles the harder case where source inputs and target inputs differ enough that naive transfer underperforms. The three shifts you will encounter in practice are:

Covariate shift — the input distribution $p(x)$ changes while the conditional $p(y|x)$ is stable. Photos taken with a new camera or under different lighting are typical examples.

Label shift — the marginal label distribution $p(y)$ changes while $p(x|y)$ is stable. A medical classifier deployed in a region with different disease prevalence sees this.

Concept drift — the decision boundary itself moves, and $p(y|x)$ changes. Fraud detection, content moderation, and recommender systems all encounter this over time.

FOR AUTHOR USE ONLY

6.4.2 Method Families

Three method families dominate the literature. Statistical alignment methods add a loss term that minimizes a divergence between source and target feature distributions—CORAL aligns second-order statistics, while Maximum Mean Discrepancy (MMD) aligns higher-order moments. Adversarial methods train a domain classifier to distinguish source from target features and update the feature extractor in the opposite direction, making the domains indistinguishable; the Domain-Adversarial Neural Network (DANN) is the canonical example. Self-training methods generate pseudo-labels for target examples and fold them back into supervised training, often with confidence thresholds and consistency regularization. A fourth family, test-time adaptation, updates a small subset of parameters at inference (commonly BatchNorm statistics) using target inputs alone.

Which family to pick depends on what you have. If both source and target labels are available, start with supervised fine-tuning on a mixed dataset. If target labels are unavailable, choose between adversarial methods (when you have at least a few thousand unlabeled target examples and can afford the training instability) and statistical alignment (simpler and more stable but typically a few points behind DANN-style methods).

6.4.3 PyTorch: DANN with Gradient Reversal

DANN attaches a domain classifier to the feature extractor and inserts a gradient reversal layer between them. During the forward pass the layer is the identity; on the backward pass it multiplies the gradient by $-\lambda$. This trains the feature extractor to fool the domain classifier, producing features that are predictive of the class label but invariant to the domain.

FOR AUTHOR USE ONLY

```python
import torch
import torch.nn as nn
from torch.autograd import Function

class GradReverse(Function):
    """Identity on the forward pass; multiplies gradients by -lambda on backward.
    Wrapping this autograd Function turns it into a layer the feature
    extractor can be trained THROUGH while the domain classifier is
    trained AGAINST it.
    """
    @staticmethod
    def forward(ctx, x, lambda_):
        ctx.lambda_ = lambda_
        return x.view_as(x)
    @staticmethod
    def backward(ctx, grad_output):
        # Negate the gradient flowing back into the feature extractor.
        return grad_output.neg() * ctx.lambda_, None

def grad_reverse(x, lambda_=1.0):
    return GradReverse.apply(x, lambda_)
```

```python
class DANN(nn.Module):
    """Feature extractor + label classifier + domain classifier."""

    def __init__(self, feature_extractor: nn.Module,
                 feat_dim: int, num_classes: int):
        super().__init__()
        self.features = feature_extractor          # e.g., a CNN backbone
        # Label classifier — trained on SOURCE samples only (we have labels).
        self.label_head = nn.Sequential(
            nn.Linear(feat_dim, 256), nn.ReLU(inplace=True),
            nn.Linear(256, num_classes),
        )
        # Domain classifier — trained on BOTH source and target features.
        self.domain_head = nn.Sequential(
            nn.Linear(feat_dim, 256), nn.ReLU(inplace=True),
            nn.Linear(256, 2),                     # 0 = source, 1 = target
        )
    def forward(self, x, lambda_: float = 1.0):
        z = self.features(x).flatten(1)            # shared embedding
        class_logits = self.label_head(z)
        # Reverse gradients so the feature extractor learns to confuse
        # the domain classifier instead of helping it.
        z_rev = grad_reverse(z, lambda_)
        domain_logits = self.domain_head(z_rev)
        return class_logits, domain_logits
```

```python
def train_step(model, src_batch, tgt_batch, optimizer,
               criterion_cls, criterion_dom, lambda_: float):
    """One DANN training step.

    src_batch -> (images, labels) from the labeled source domain.
    tgt_batch -> (images,)        from the unlabeled target domain.
    """
    src_x, src_y = src_batch
    tgt_x = tgt_batch

    # ---- Source pass: classification loss + source-domain loss ----
    src_cls, src_dom = model(src_x, lambda_)
    loss_cls = criterion_cls(src_cls, src_y)
    src_dom_y = torch.zeros(src_x.size(0), dtype=torch.long,
                            device=src_x.device)
    loss_dom_src = criterion_dom(src_dom, src_dom_y)

    # ---- Target pass: domain loss only (no labels on target) ----
    _, tgt_dom = model(tgt_x, lambda_)
    tgt_dom_y = torch.ones(tgt_x.size(0), dtype=torch.long,
                           device=tgt_x.device)
    loss_dom_tgt = criterion_dom(tgt_dom, tgt_dom_y)

    # Total loss = class loss + domain loss. Gradient reversal
    # automatically flips the sign of the domain gradient with respect
    # to the feature extractor.
    loss = loss_cls + 0.5 * (loss_dom_src + loss_dom_tgt)
```

```python
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()
    return {"cls": loss_cls.item(), "dom": (loss - loss_cls).item()}

# Schedule lambda from 0 to 1 over training so the domain signal
# fades in gradually (the original DANN paper recipe).
def lambda_schedule(step: int, total_steps: int, gamma: float = 10.0):
    import math
    p = step / max(1, total_steps)
    return 2.0 / (1.0 + math.exp(-gamma * p)) - 1.0
```

FOR AUTHOR USE ONLY

Code Snippet 6.6. PyTorch DANN with gradient reversal.

6.4.4 TensorFlow: deep CORAL alignment

CORAL (Correlation Alignment) is a much simpler statistical-alignment method. It adds a loss term equal to the squared Frobenius distance between the covariance matrices of source and target features at one or more layers. There is no adversarial training, no extra discriminator, and the loss is differentiable in a single line. Deep CORAL is a strong, stable baseline whenever covariate shift is the dominant problem.

```
import tensorflow as tf
from tensorflow.keras import layers, Model
from tensorflow.keras.applications import EfficientNetB0


def coral_loss(source_features: tf.Tensor,
               target_features: tf.Tensor) -> tf.Tensor:
    """Squared Frobenius distance between source and target covariance matrices.

    Shapes:
      source_features: (Ns, D)
      target_features: (Nt, D)
    Returns a scalar suitable to add to the supervised loss.
    """
    d = tf.cast(tf.shape(source_features)[1], tf.float32)  # feature dim

    # Center each batch (subtract the per-feature mean).
    src = source_features - tf.reduce_mean(source_features, axis=0,
                                           keepdims=True)
    tgt = target_features - tf.reduce_mean(target_features, axis=0,
                                           keepdims=True)

    # Sample covariance matrices: (D, D).
    ns = tf.cast(tf.shape(src)[0], tf.float32)
    nt = tf.cast(tf.shape(tgt)[0], tf.float32)
    cov_src = tf.matmul(src, src, transpose_a=True) / (ns - 1.0 + 1e-6)
    cov_tgt = tf.matmul(tgt, tgt, transpose_a=True) / (nt - 1.0 + 1e-6)

    # Squared Frobenius norm of the difference, normalized so the loss
    # is roughly scale-free in feature dimension.
    return tf.reduce_sum(tf.square(cov_src - cov_tgt)) / (4.0 * d * d)


def build_coral_model(num_classes, input_shape=(224, 224, 3)):
    """Backbone + classifier head. The forward pass returns both the
    embeddings (for the CORAL term) and the class logits."""
    backbone = EfficientNetB0(include_top=False, weights="imagenet",
                              input_shape=input_shape, pooling="avg")
    inputs = layers.Input(shape=input_shape)
    z = backbone(inputs)                          # (B, D) embedding
    logits = layers.Dense(num_classes,
                          activation=None, name="logits")(z)
    return Model(inputs, [z, logits], name="coral_model")
```

FOR AUTHOR USE ONLY

```python
optimizer = tf.keras.optimizers.AdamW(1e-4, weight_decay=1e-4)
cls_loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True)

@tf.function
def coral_train_step(model, src_x, src_y, tgt_x,
                     coral_weight: float = 1.0):
    """Single CORAL training step.
    src_x, src_y come from the LABELED source domain.
    tgt_x comes from the UNLABELED target domain.
    """
    with tf.GradientTape() as tape:
        z_src, logits_src = model(src_x, training=True)
        z_tgt, _          = model(tgt_x, training=True)

        loss_cls   = cls_loss_fn(src_y, logits_src)
        loss_coral = coral_loss(z_src, z_tgt)
        loss_total = loss_cls + coral_weight * loss_coral

    grads = tape.gradient(loss_total, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
    return {"cls": loss_cls, "coral": loss_coral,
            "total": loss_total}

# Driver pseudocode:
# model = build_coral_model(num_classes=10)
# for src_batch, tgt_batch in zip(src_ds, tgt_ds.repeat()):
#     src_x, src_y = src_batch
#     tgt_x = tgt_batch                  # unlabeled target images
```

Code Snippet 6.7. TensorFlow deep CORAL loss + training step.

FOR AUTHOR USE ONLY

CORAL pairs naturally with the fine-tuning patterns from Section 3. A common recipe is to start with a frozen backbone, train the head with the CORAL term turned off, then unfreeze the top blocks and switch the term on. The CORAL weight is usually set so that the two loss terms are within an order of magnitude of each other after a few hundred steps.

6.5 Practical Cheat Sheet & References

Decision Flow

1. Is the target dataset small (a few thousand examples or fewer)? Start with a frozen feature extractor.
2. Does the frozen-backbone model leave headroom on validation accuracy? Unfreeze the top one or two blocks and train with a learning rate 10–100x smaller.
3. Is there a visible source-target distribution gap (different sensors, lighting, geography, etc.)? Add a domain adaptation term — CORAL first (stable), DANN if you need the extra accuracy.

4. Is the dataset large enough to overfit even with the full backbone unfrozen? Switch to full fine-tuning with discriminative learning rates and cosine decay.

Common Pitfalls

- Forgetting to set BatchNorm layers to eval mode when fine-tuning with small batches — the running statistics get overwritten with noise.
- Forgetting to recompile a Keras model after flipping layer.trainable — the optimizer keeps the old variable list.
- Using the same learning rate for the entire network during fine-tuning — early layers move too far from their pre-trained state.
- Training a DANN without scheduling lambda from 0 to 1 — the domain loss overwhelms the class loss and the model collapses.
- Evaluating a domain-adapted model only on the source domain — always reserve a target-domain validation split.

FOR AUTHOR USE ONLY

Chapter 7 — ResNet Architecture

Introduction to ResNet

Deep learning models have achieved significant success in image classification and computer vision applications. As Convolutional Neural Networks (CNNs) became more advanced, researchers attempted to improve model performance by increasing the number of layers in the network. Deeper neural networks can learn more complex visual patterns and hierarchical image features compared to shallow networks.

However, increasing network depth also introduces several training difficulties. Very deep CNN models often suffer from problems such as vanishing gradients, slower convergence, and degradation in performance. In some cases, adding more layers may even reduce model accuracy instead of improving it.

To address these limitations, the Residual Network (ResNet) architecture was introduced by researchers at Microsoft Research in 2015. ResNet introduced a new learning approach called residual learning, which allows information to pass through shortcut connections known as skip connections. These connections help preserve important information across layers and improve gradient flow during training.

FOR AUTHOR USE ONLY

One of the major advantages of ResNet is its ability to train extremely deep neural networks more effectively than traditional CNN architectures. ResNet models such as ResNet34, ResNet50, ResNet101, and ResNet152 have achieved excellent performance in image classification and object detection tasks.

Unlike traditional CNN models that directly learn complex mappings, ResNet focuses on learning residual mappings. This allows deeper architectures to train more efficiently while reducing optimization difficulties. Due to these advantages, ResNet has become one of the most important deep learning architectures used in modern computer vision systems.

Today, ResNet models are widely applied in healthcare, autonomous vehicles, facial recognition, surveillance systems, and many other AI-based applications that require accurate image analysis.

Before understanding how ResNet solves deep learning limitations, it is important to first understand the challenges faced by very deep neural networks.

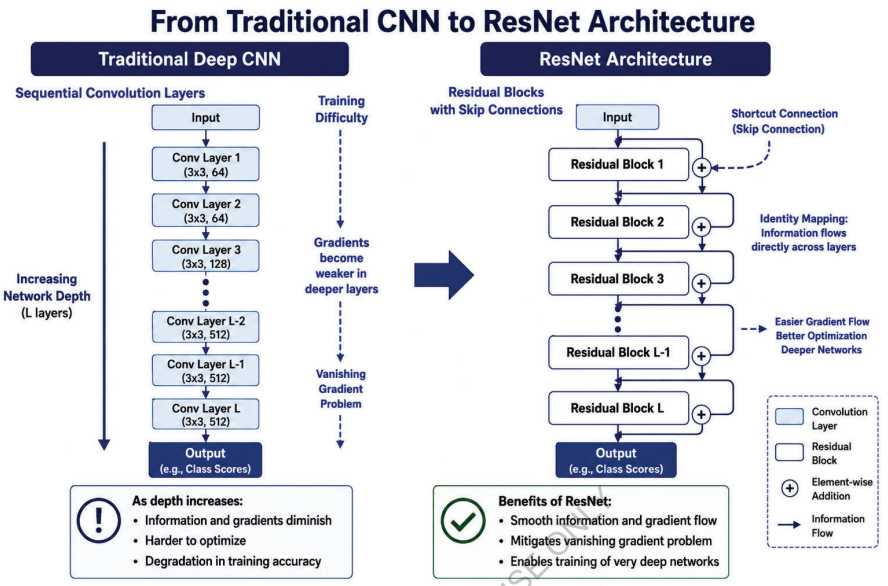


Figure 7.1. Evolution from Traditional CNN to ResNet Architecture.

FOR AUTHOR USE ONLY

Problem of Very Deep Neural Networks

Deep Convolutional Neural Networks can learn highly complex visual patterns from image data. As the number of layers increases, the network becomes more powerful and can capture more detailed hierarchical features. Early layers generally detect simple features such as edges and textures, while deeper layers identify complex object structures and patterns.

Although deeper neural networks have greater learning capability, increasing network depth also introduces several training challenges. One of the most common problems is the vanishing gradient problem. During backpropagation, gradients are propagated backward through multiple layers to update the model parameters. In very deep networks, these gradients may become extremely small as they move toward earlier layers.

When gradients become very small, the earlier layers learn very slowly or sometimes stop learning completely. As a result, training becomes inefficient, and model performance may decrease. Traditional activation functions such as

sigmoid and tanh are more affected by this issue because their gradients shrink rapidly for large input values.

Another major challenge in very deep neural networks is the degradation problem. In theory, adding more layers should improve network performance because deeper models can represent more complex functions. However, in practice, very deep CNN models sometimes produce higher training error and lower accuracy compared to shallower networks.

The degradation problem does not occur mainly because of overfitting. Instead, it occurs because deeper architectures become difficult to optimize during training. As the network depth increases, learning stable weight updates becomes increasingly challenging.

These limitations motivated researchers to develop architectures that could support efficient training of deeper neural networks. ResNet was introduced as a solution to these problems by allowing shortcut connections that help preserve gradient flow and improve optimization stability.

FOR AUTHOR USE ONLY

Table 7.1. Challenges in Very Deep Neural Networks.

Problem	Description
Vanishing Gradient Problem	Gradients become extremely small during backpropagation
Slow Convergence	Training requires more time in deeper networks
Degradation Problem	Accuracy decreases as network depth increases
Optimization Difficulty	Stable parameter updates become harder in deep models

To solve the training difficulties of very deep neural networks, ResNet introduced the concept of residual learning using shortcut connections.

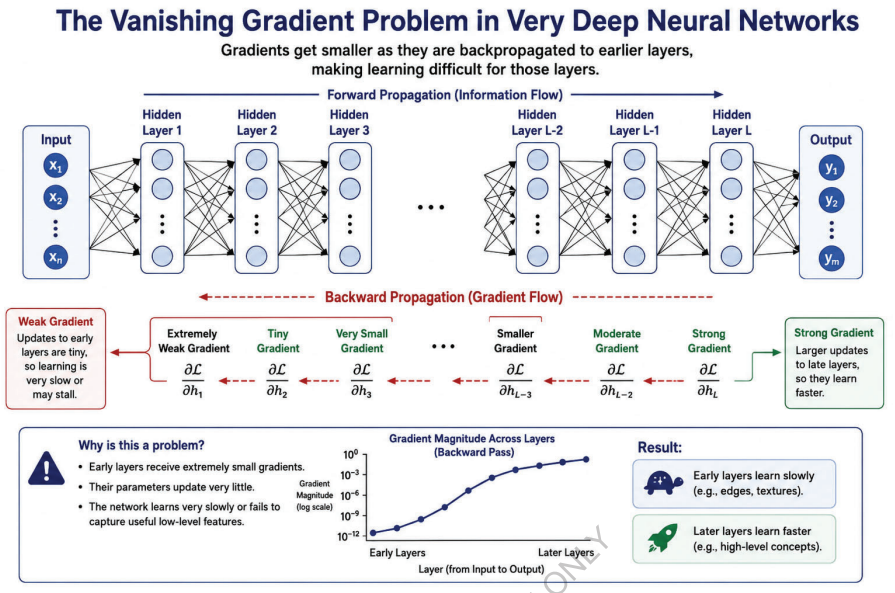


Figure 7.2. Vanishing Gradient Problem in Deep Neural Networks.

FOR AUTHOR USE ONLY

Residual Learning Concept

The main idea behind ResNet is residual learning. Instead of forcing the neural network to directly learn a complex mapping from input to output, ResNet allows the model to learn the residual difference between them. This approach makes the optimization process easier and improves training performance in very deep neural networks.

In traditional CNN architectures, each group of layers attempts to learn a direct transformation of the input data. However, as the network becomes deeper, learning these transformations becomes increasingly difficult. ResNet solves this issue by introducing shortcut connections that allow the original input to bypass certain layers and be added directly to the output.

Suppose the desired mapping is represented by **H(x)**, where **x** is the input feature map. Instead of directly learning **H(x)**, ResNet learns the residual mapping **F(x)**. The network then combines the residual mapping with the original input through a shortcut connection. The residual learning equation can be expressed as:

$$H(x) = F(x) + x, \qquad \textit{(Eq.7.1)}$$

Where:

- *x* represents the input
- *F(x)* represents the residual function learned by the network
- *H(x)* represents the final output

In this approach, the shortcut connection allows the original input to be added directly to the learned residual mapping. As a result, the network only needs to learn the residual difference rather than the complete transformation. This simplifies learning and improves gradient flow during backpropagation.

Residual learning also helps reduce the vanishing gradient problem because gradients can pass more easily through shortcut connections. This allows very deep neural networks to train more efficiently and achieve better performance compared to traditional CNN architectures.

The introduction of residual learning became one of the most important developments in deep learning and significantly improved the training of deep convolutional neural networks.

The residual learning concept is implemented in ResNet using specialized building units known as residual blocks.

FOR AUTHOR USE ONLY

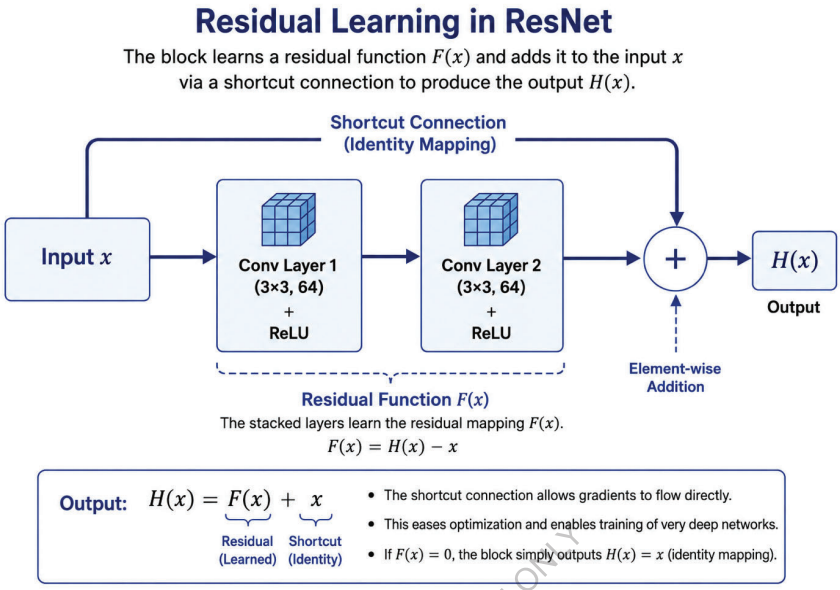


Figure 7.3. Residual Learning using Shortcut Connections.

FOR AUTHOR USE ONLY

7.1 Residual Blocks

Residual blocks are the fundamental building units of the ResNet architecture. A residual block contains convolution layers along with a shortcut connection that allows the input information to bypass certain layers and be added directly to the output.

In traditional CNN architectures, data passes sequentially through convolution layers without any direct shortcut paths. As the network depth increases, learning becomes more difficult due to gradient-related problems. Residual blocks solve this issue by preserving important information through identity mapping.

A standard residual block generally consists of two convolution layers followed by batch normalization and ReLU activation functions. Alongside these layers, a shortcut connection transfers the original input directly to the output of the block.

The output of a residual block is obtained by adding the shortcut connection to the output generated by the convolution layers. This operation allows the network to learn residual mappings more efficiently and improves gradient flow during backpropagation.

Residual blocks help deep neural networks train more effectively because the shortcut connections reduce optimization difficulty. Instead of learning a completely new transformation, the network only needs to learn small residual changes. This improves convergence and allows very deep ResNet models to achieve better performance.

In some cases, the dimensions of the input and output may differ. To solve this issue, a projection shortcut using a 1×1 convolution layer can be applied to match the dimensions before addition.

Residual blocks play an important role in enabling ResNet architectures such as ResNet34 and ResNet50 to train efficiently even with very large network depth.

Table 7.2. Traditional CNN Block vs Residual Block.

Feature	Traditional CNN Block	Residual Block
Information Flow	Sequential only	Includes shortcut connection
Gradient Flow	Weak in deep layers	Improved through skip connections
Training Stability	Lower in deep networks	More stable
Learning Method	Direct mapping	Residual mapping
Optimization Difficulty	Higher	Lower

FOR AUTHOR USE ONLY

```python
# Import required libraries
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.layers import ReLU
from tensorflow.keras.layers import Add
from tensorflow.keras.layers import Input
from tensorflow.keras.models import Model

# Function to create a basic residual block
def residual_block(x, filters):
    # Save input tensor for shortcut connection
    shortcut = x
    # First convolution layer
    x = Conv2D(filters=filters,
               kernel_size=(3, 3),
               padding='same')(x)
    # Batch normalization
    x = BatchNormalization()(x)
    # Apply ReLU activation
    x = ReLU()(x)
    # Second convolution layer
    x = Conv2D(filters=filters,
               kernel_size=(3, 3),
               padding='same')(x)
    # Batch normalization
    x = BatchNormalization()(x)
    # Add shortcut connection
    x = Add()([x, shortcut])
    # Final activation function
    x = ReLU()(x)
    return x
```

FOR AUTHOR USE ONLY

Code Snippet 7.1. Creating a Basic Residual Block using Keras.

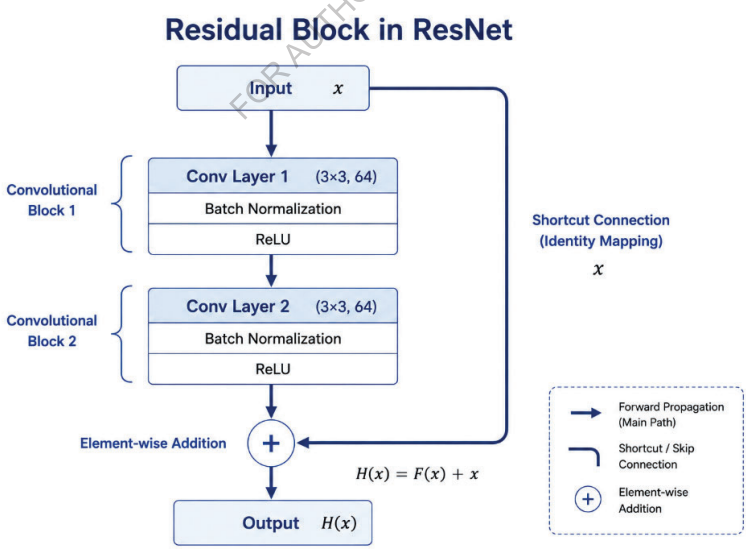


Figure 7.4. Basic Structure of a Residual Block.

Multiple residual blocks are combined to form the complete ResNet architecture used for deep image classification tasks.

7.1.1 ResNet Architecture Flow

The ResNet architecture processes image data through multiple layers that work together to extract visual features and perform image classification. Unlike traditional CNN architectures, ResNet uses residual blocks with shortcut connections that improve information flow across deep networks.

The architecture begins with an input image that passes through an initial convolution layer. This layer extracts low-level image features such as edges, textures, and basic shapes. A max pooling layer is then applied to reduce the spatial dimensions of the feature maps while preserving important visual information. After the initial layers, the image data passes through several groups of residual blocks. Each residual block learns residual mappings while shortcut connections help maintain stable learning across deeper layers. As the network becomes deeper, the number of filters gradually increases, allowing the model to learn more complex image features. In deeper stages of the network, the feature map dimensions become smaller while the extracted features become more abstract and semantically meaningful. This hierarchical feature learning enables ResNet models to recognize complex objects and visual patterns effectively.

FOR AUTHOR USE ONLY

After feature extraction, a global average pooling layer is applied. This layer reduces the spatial dimensions of the feature maps by calculating the average value of each feature map. Global average pooling helps reduce the number of trainable parameters and improves computational efficiency. Finally, the extracted features are passed into a fully connected layer followed by a Softmax activation function. The Softmax layer generates probability values for different image classes and produces the final classification output.

A standard ResNet architecture generally follows the sequence:

Input Image → Convolution Layer → Pooling Layer → Residual Blocks → Global Average Pooling → Fully Connected Layer → Softmax Output

The use of residual blocks and shortcut connections allows ResNet architectures to train efficiently even when the network contains dozens or hundreds of layers.

Table 7.3. Main Components of ResNet Architecture.

Component	Function
Input Layer	Receives the input image
Convolution Layer	Extracts low-level visual features
Pooling Layer	Reduces feature map dimensions
Residual Blocks	Learn residual mappings
Shortcut Connection	Preserves gradient flow
Global Average Pooling	Reduces spatial dimensions
Fully Connected Layer	Performs classification
Softmax Layer	Produces class probabilities

FOR AUTHOR USE ONLY

```
# Import required libraries
from tensorflow.keras.layers import Input
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.layers import GlobalAveragePooling2D
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import Model

# Define input layer
inputs = Input(shape=(224, 224, 3))
# Initial convolution layer
x = Conv2D(filters=64,
           kernel_size=(7, 7),
           strides=2,
           padding='same',
           activation='relu')(inputs)
# Max pooling layer
x = MaxPooling2D(pool_size=(3, 3),
                 strides=2,
                 padding='same')(x)
```

```
# Add residual blocks
x = residual_block(x, 64)
x = residual_block(x, 64)
# Global average pooling layer
x = GlobalAveragePooling2D()(x)
# Fully connected output layer
outputs = Dense(10,
                activation='softmax')(x)
# Create ResNet-style model
model = Model(inputs, outputs)
# Display model summary
model.summary()
```

Code Snippet 7.2. Building a Simple ResNet-style Model using Keras.

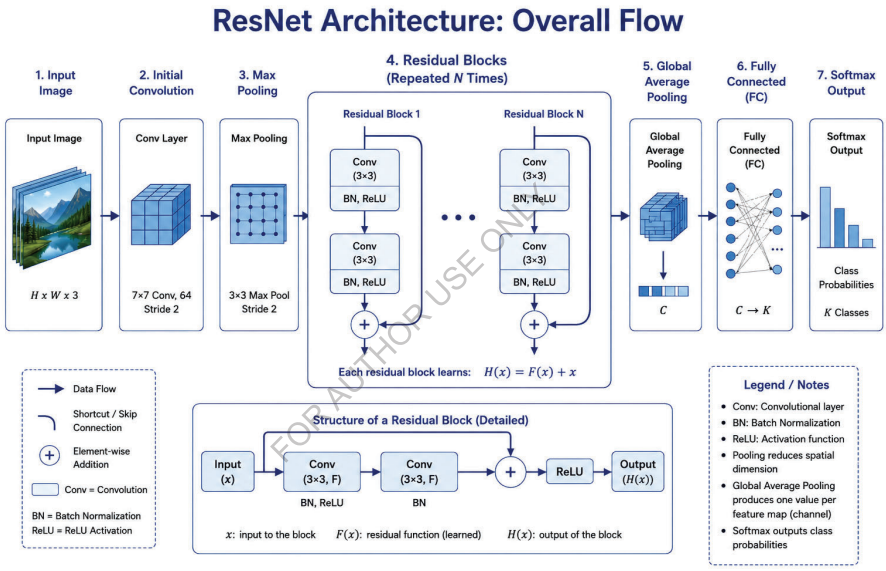


Figure 7.5. General Flow of ResNet Architecture.

In deeper ResNet architectures such as ResNet50, a specialized residual structure known as the bottleneck design is commonly used to improve computational efficiency.

7.2 Bottleneck Design

As ResNet architectures become deeper, the computational complexity and number of parameters increase significantly. Training very deep networks with standard residual blocks may require large memory usage and longer training time. To solve this issue, deeper ResNet models such as ResNet50, ResNet101, and ResNet152 use a specialized structure known as the

bottleneck design. The bottleneck design improves computational efficiency while preserving the network's ability to learn rich visual features. Instead of using only two 3×3 convolution layers, the bottleneck block uses three convolution layers arranged in the following sequence:

1×1 Convolution → 3×3 Convolution → 1×1 Convolution

The first 1×1 convolution layer is used for dimensionality reduction. It decreases the number of feature channels before the computationally expensive 3×3 convolution operation is applied. This helps reduce the number of calculations and improves computational efficiency. The middle 3×3 convolution layer performs feature extraction and learns important visual patterns from the image data. After feature extraction, the final 1×1 convolution layer restores the feature dimensions to their original size. Like standard residual blocks, bottleneck blocks also include shortcut connections that transfer the input directly to the output. These shortcut connections help maintain stable gradient flow during backpropagation and improve training performance in very deep neural networks.

One major advantage of the bottleneck design is that it allows ResNet models to become significantly deeper without causing excessive computational overhead. This makes architectures such as ResNet50 more efficient compared to using only standard residual blocks.

FOR AUTHOR USE ONLY

Table 7.4. Structure of a Bottleneck Residual Block.

Layer	Function
1×1 Convolution	Reduces feature dimensions
3×3 Convolution	Extracts important image features
1×1 Convolution	Restores feature dimensions
Shortcut Connection	Preserves information flow
ReLU Activation	Introduces non-linearity

```
# Import required libraries
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import BatchNormalization
from tensorflow.keras.layers import ReLU
from tensorflow.keras.layers import Add
# Function to create a bottleneck residual block
def bottleneck_block(x, filters):
    # Save shortcut connection
    shortcut = x
    # First 1x1 convolution layer
    x = Conv2D(filters,
               (1, 1),
               padding='same')(x)
    x = BatchNormalization()(x)
    x = ReLU()(x)
    # 3x3 convolution layer
    x = Conv2D(filters,
               (3, 3),
               padding='same')(x)
    x = BatchNormalization()(x)
    x = ReLU()(x)
    # Final 1x1 convolution layer
    x = Conv2D(filters * 4,
               (1, 1),
               padding='same')(x)
    x = BatchNormalization()(x)
    # Projection shortcut if dimensions differ
    shortcut = Conv2D(filters * 4,
                      (1, 1),
                      padding='same')(shortcut)
    shortcut = BatchNormalization()(shortcut)
    # Add shortcut connection
    x = Add()([x, shortcut])
    # Final activation
    x = ReLU()(x)
    return x
```

FOR AUTHOR USE ONLY

Code Snippet 7.3. Creating a Bottleneck Residual Block using Keras.

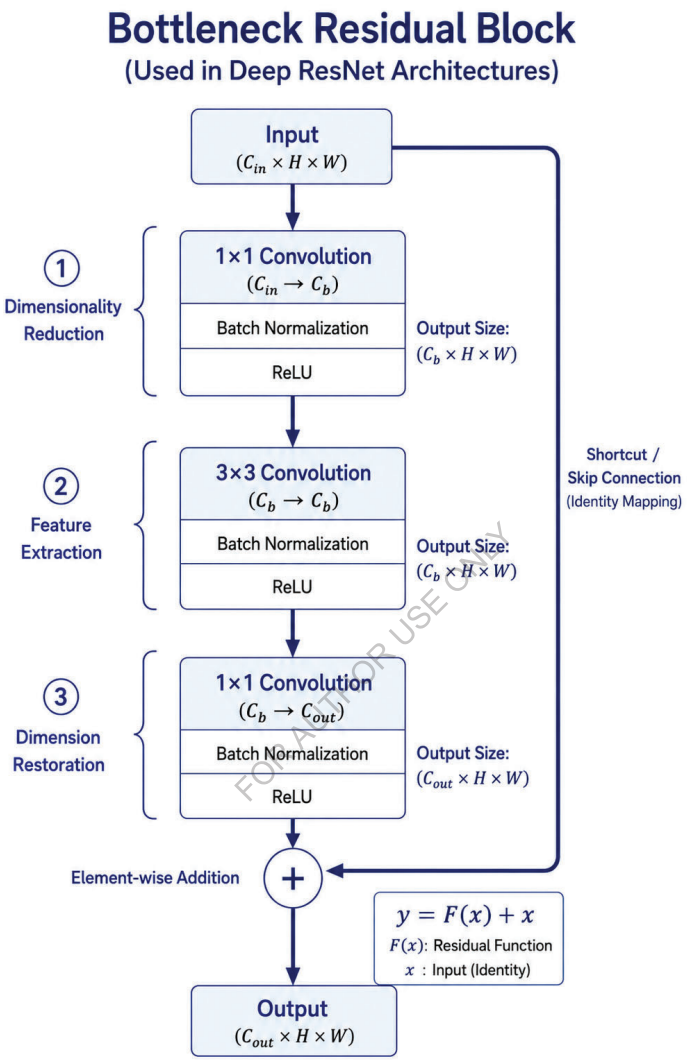


Figure 7.6. Bottleneck Residual Block Structure.

Different ResNet architectures use different types of residual blocks depending on the network depth and computational requirements.

7.2.1 ResNet34 Architecture

ResNet34 is one of the early versions of the ResNet family introduced to improve deep neural network training using residual learning. The architecture contains 34 weighted layers and uses standard residual blocks with shortcut connections to improve gradient flow during training.

Unlike deeper ResNet architectures such as ResNet50 and ResNet101, ResNet34 mainly uses basic residual blocks composed of two 3×3 convolution layers. These residual blocks enable efficient feature learning while supporting stable network training.

The architecture begins with an initial 7×7 convolution layer followed by a max pooling layer. After the initial layers, the network is divided into several stages containing multiple residual blocks. As the network depth increases, the number of filters also increases gradually. The residual blocks in ResNet34 allow the network to learn deeper image features without suffering significantly from the vanishing gradient problem. Due to its moderate depth, ResNet34 provides a balance between computational efficiency and classification accuracy. One advantage of ResNet34 is that it is less computationally expensive than deeper architectures such as ResNet50 while still achieving strong performance in image classification tasks. This makes ResNet34 suitable for applications where computational resources are limited.

FOR AUTHOR USE ONLY

The general architecture of ResNet34 includes:

- Initial convolution layer
- Max pooling layer
- Multiple groups of residual blocks
- Global average pooling layer
- Fully connected output layer

ResNet34 is commonly used in image classification, medical imaging, object recognition, and transfer learning applications because of its efficient architecture and stable training performance.

Table 7.5. Layer Structure of ResNet34.

Stage	Layer Structure	Output Size
Input	224 × 224 RGB Image	224 × 224

Convolution Layer	7×7 Conv, 64 Filters	112 × 112
Max Pooling	3×3 Max Pooling	56 × 56
Residual Block Group 1	3 Residual Blocks	56 × 56
Residual Block Group 2	4 Residual Blocks	28 × 28
Residual Block Group 3	6 Residual Blocks	14 × 14
Residual Block Group 4	3 Residual Blocks	7 × 7
Global Average Pooling	Average Pooling	1 × 1
Fully Connected Layer	Softmax Output	Number of Classes

```
# Import required libraries
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.models import Model

# Load pretrained ResNet model
model = ResNet50(weights='imagenet',
                 include_top=True)

# Display model summary
model.summary()
```

FOR AUTHOR USE ONLY

Code Snippet 7.4. Loading ResNet34-style Architecture using Keras.

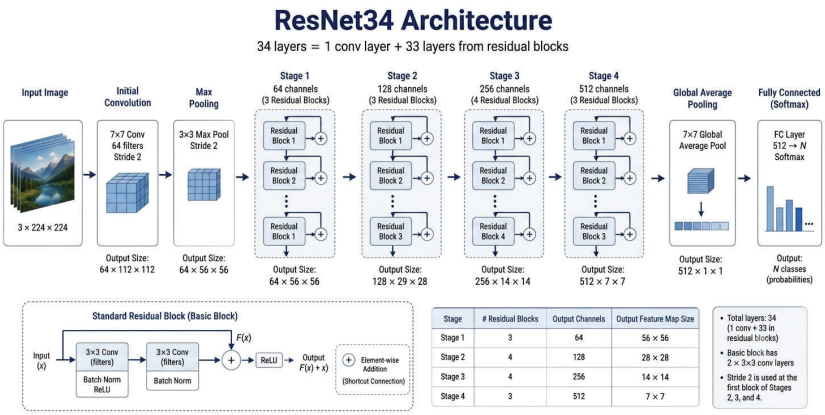


Figure 7.7. Architecture of ResNet34.

While ResNet34 uses standard residual blocks, deeper architectures such as ResNet50 use bottleneck residual blocks to improve computational efficiency.

7.2.2 ResNet50 Architecture

ResNet50 is a deeper version of the ResNet family that contains 50 weighted layers. Unlike ResNet34, which uses standard residual blocks, ResNet50 uses bottleneck residual blocks to improve computational efficiency while supporting deeper network architecture.

The bottleneck design allows ResNet50 to reduce the number of computations and parameters while maintaining strong feature extraction capability. Each bottleneck block consists of three convolution layers arranged in the following sequence:

1×1 Convolution → 3×3 Convolution → 1×1 Convolution

FOR AUTHOR USE ONLY

The first 1×1 convolution layer reduces the number of feature channels, the 3×3 convolution layer performs feature extraction, and the final 1×1 convolution layer restores the feature dimensions. This structure allows the network to become deeper without significantly increasing computational complexity.

Like other ResNet variants, ResNet50 starts with an initial convolution layer followed by a max pooling operation. The network then passes through multiple groups of bottleneck residual blocks where deeper image features are extracted gradually. As the network depth increases, ResNet50 becomes capable of learning highly complex visual representations. The deeper architecture improves image classification performance and allows the model to recognize fine-grained image patterns more effectively. After feature extraction, global average pooling is applied before the final fully connected layer generates classification outputs using the Softmax activation function. One major advantage of ResNet50 is that it achieves high classification accuracy while remaining computationally efficient compared to traditional deep CNN architectures. Due to its strong performance, ResNet50 is widely used in transfer learning, object detection, medical image analysis, and many computer vision applications.

Table 7.6. Layer Structure of ResNet50.

Stage	Layer Structure	Output Size
Input	224 × 224 RGB Image	224 × 224
Convolution Layer	7×7 Conv, 64 Filters	112 × 112
Max Pooling	3×3 Max Pooling	56 × 56
Bottleneck Block Group 1	3 Bottleneck Blocks	56 × 56
Bottleneck Block Group 2	4 Bottleneck Blocks	28 × 28
Bottleneck Block Group 3	6 Bottleneck Blocks	14 × 14
Bottleneck Block Group 4	3 Bottleneck Blocks	7 × 7
Global Average Pooling	Average Pooling	1 × 1
Fully Connected Layer	Softmax Output	Number of Classes

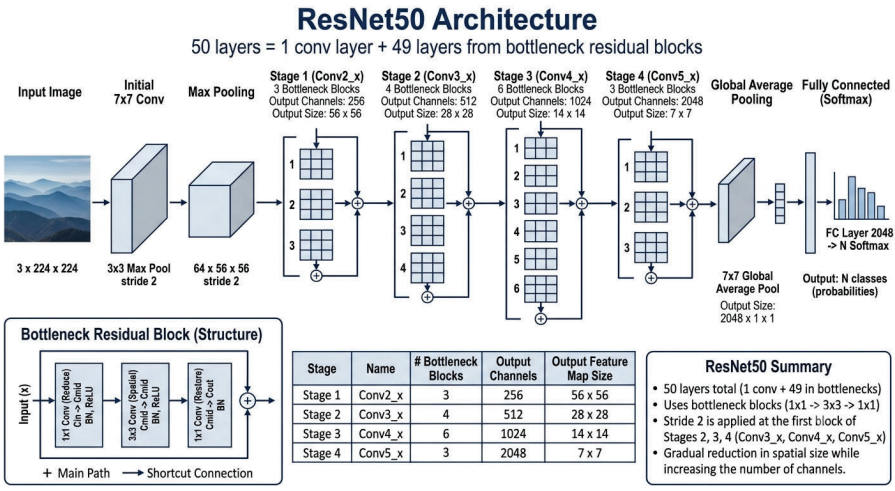
FOR AUTHOR USE ONLY

```
# Import required libraries
from tensorflow.keras.applications import ResNet50

# Load pretrained ResNet50 model
model = ResNet50(weights='imagenet',
                 include_top=True)

# Display model summary
model.summary()
```

Code Snippet 7.5. Loading Pretrained ResNet50 using Keras.



Stage	Name	# Bottleneck Blocks	Output Channels	Output Feature Map Size
Stage 1	Conv2_x	3	256	56 x 56
Stage 2	Conv3_x	4	512	28 x 28
Stage 3	Conv4_x	6	1024	14 x 14
Stage 4	Conv5_x	3	2048	7 x 7

Figure 7.8. Architecture of ResNet50.

Although both ResNet34 and ResNet50 use residual learning, their architectures differ in depth, block design, and computational efficiency.

FOR AUTHOR USE ONLY

7.3 ResNet34 vs ResNet50

ResNet34 and ResNet50 are two widely used architectures from the ResNet family. Both models use residual learning and shortcut connections to improve the training of deep neural networks. However, the two architectures differ in network depth, residual block design, computational complexity, and performance.

ResNet34 contains 34 weighted layers and mainly uses standard residual blocks composed of two 3×3 convolution layers. The architecture is moderately deep and provides a good balance between computational efficiency and classification performance.

In contrast, ResNet50 contains 50 weighted layers and uses bottleneck residual blocks instead of standard residual blocks. The bottleneck design introduces additional 1×1 convolution layers that reduce computational cost while allowing the network to become deeper.

An important distinction between the two architectures is their parameter count and computational requirements. Although ResNet50 is deeper than ResNet34, the bottleneck structure helps reduce memory usage and improves efficiency compared to using only standard residual blocks.

ResNet50 generally achieves higher classification accuracy because deeper layers can learn more detailed and complex image features. However, ResNet50 also requires more computational resources and longer training time compared to ResNet34.

ResNet34 is often preferred when computational resources are limited or when faster training is required. On the other hand, ResNet50 is commonly used in applications that require high classification accuracy and stronger feature extraction capability.

Both architectures are widely used in transfer learning and computer vision tasks such as image classification, object detection, medical imaging, and facial recognition.

Table 7.7. Comparison Between ResNet34 and ResNet50.

Feature	ResNet34	ResNet50
Number of Layers	34 Layers	50 Layers
Residual Block Type	Standard Residual Block	Bottleneck Residual Block
Convolution Structure	Two 3×3 Convolutions	1×1 → 3×3 → 1×1 Convolutions
Computational Complexity	Lower	Higher
Memory Requirement	Lower	Higher
Training Speed	Faster	Slower
Feature Learning Capability	Moderate	Stronger
Classification Accuracy	High	Very High
Suitable Applications	Medium-scale tasks	Complex image classification tasks

FOR AUTHOR USE ONLY

Table 7.8. Quantitative Comparison Between ResNet34 and ResNet50.

Metric	ResNet34	ResNet50
Number of Layers	34 Layers	50 Layers
Parameters	~21.8 Million	~25.6 Million
FLOPs (224×224 Input)	~3.6 GFLOPs	~4.1 GFLOPs
Top-1 Accuracy (ImageNet)	~73.3%	~76.0%
Top-5 Accuracy (ImageNet)	~91.4%	~92.9%
Residual Block Type	Standard Block	Bottleneck Block
Training Time	Lower	Higher
Memory Requirement	Lower	Higher

FOR AUTHOR USE ONLY

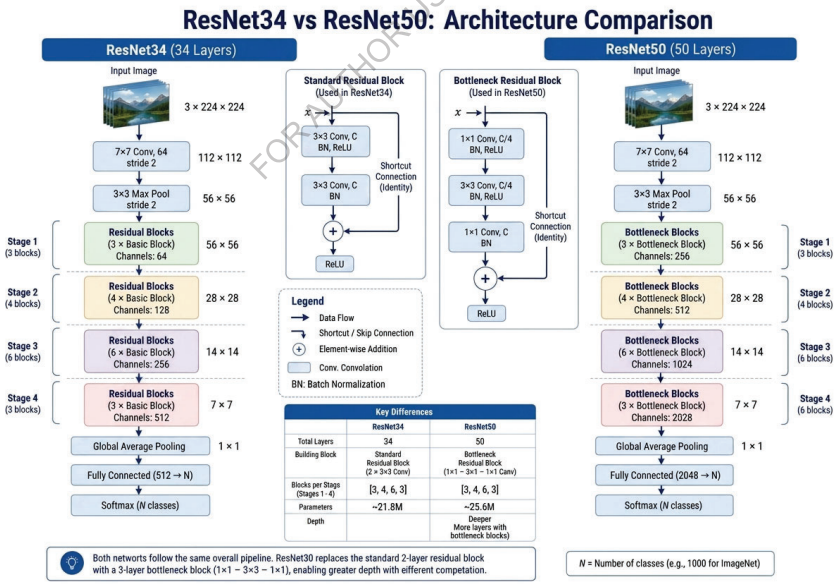


Figure 7.9. Comparison Between ResNet34 and ResNet50 Architectures.

The quantitative comparison shows that ResNet50 contains slightly more parameters and computational operations than ResNet34. However, the deeper architecture and bottleneck design enable ResNet50 to achieve higher classification accuracy on benchmark datasets such as ImageNet. The choice between the two models depends on the trade-off between computational efficiency and predictive performance. After understanding the architectural differences between ResNet34 and ResNet50, a practical implementation can further demonstrate how ResNet models are applied in image classification tasks.

7.3.1 Experimental Benchmark Example

To further illustrate the differences between ResNet34 and ResNet50, a simple benchmark comparison can be considered using the ImageNet dataset. ImageNet is one of the most widely used datasets for evaluating image classification models and contains millions of labeled images across thousands of object categories.

FOR AUTHOR USE ONLY

Although both architectures achieve strong performance, ResNet50 generally provides higher classification accuracy because of its deeper architecture and bottleneck residual blocks. However, this improvement comes with increased computational requirements.

Table 7.9. Example Benchmark Performance on ImageNet.

Model	Top-1 Accuracy	Parameters
ResNet34	73.3%	21.8 million
ResNet50	76.0%	25.6 million

The benchmark results indicate that ResNet50 achieves higher classification accuracy than ResNet34. This improvement is mainly due to its deeper architecture and more efficient bottleneck design. However, ResNet34 remains a practical choice when computational resources or training time are limited.

7.3.2 Practical Implementation using Keras

After understanding the architecture of ResNet models, a practical implementation can help demonstrate how ResNet is used for image classification tasks. In deep learning applications, pretrained ResNet models are commonly used through transfer learning. These pretrained models have

already learned important visual features from large image datasets such as ImageNet.

In this implementation, the pretrained ResNet50 model is loaded using the Keras deep learning library. The model can classify images into multiple object categories based on previously learned image features.

Before prediction, the input image must be preprocessed. The image is resized to match the input dimensions required by ResNet50, converted into numerical arrays, and normalized using preprocessing functions.

After preprocessing, the image is passed through the ResNet50 model to generate predictions. The model outputs probability scores for different image categories. The predicted labels are then decoded into human-readable class names.

Pretrained ResNet50 models are widely used in healthcare, object recognition, facial recognition, and industrial image analysis because they provide strong performance while reducing the need for extensive training from scratch.

Table 7.10. Applications of Pretrained ResNet Models.

Application Area	Example Usage
Medical Imaging	Disease detection from X-rays and MRI scans
Object Detection	Recognizing objects in images and videos
Facial Recognition	Identity verification systems
Autonomous Vehicles	Road and traffic sign recognition
Industrial Inspection	Product defect detection

FOR AUTHOR USE ONLY

```
from google.colab import files

# Upload image file
uploaded = files.upload()
```

```
Choose files  cat.jpg
cat.jpg(image/jpeg) - 14384497 bytes, last modified: 21/05/2026 - 100% done
Saving cat.jpg to cat.jpg
```

```
# Import required libraries
import numpy as np
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import preprocess_input

# Load image from directory
img = image.load_img('cat.jpg',
                     target_size=(224, 224))

# Convert image to array
img_array = image.img_to_array(img)

# Expand dimensions for batch processing
img_array = np.expand_dims(img_array,
                           axis=0)

# Preprocess image for ResNet50
img_array = preprocess_input(img_array)

# Display image shape
print("Processed Image Shape:",
      img_array.shape)
```

```
Processed Image Shape: (1, 224, 224, 3)
```

FOR AUTHOR USE ONLY

Code Snippet 7.6. Loading and Preprocessing an Image for ResNet50.

```
# Import required libraries
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.applications.resnet50 import decode_predictions
# Load pretrained ResNet50 model
model = ResNet50(weights='imagenet')
# Generate predictions
predictions = model.predict(img_array)
# Decode prediction labels
decoded_predictions = decode_predictions(predictions,
                                         top=3)[0]
# Display top predictions
print("Top Predictions:")
for prediction in decoded_predictions:
    # Extract prediction details
    label = prediction[1]
    probability = prediction[2]

    print(f"{label}: {probability:.4f}")
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-appl
102967424/102967424 ━━━━━━━━━━━━━━━━━━━━ 4s 0us/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step
Downloading data from https://storage.googleapis.com/download.tensorflow.o
35363/35363 ━━━━━━━━━━━━━━━━━━━━ 0s 1us/step
Top Predictions:
Egyptian_cat: 0.6851
tiger_cat: 0.0821
tabby: 0.0664
```

Code Snippet 7.7. Image Classification using Pretrained ResNet50.

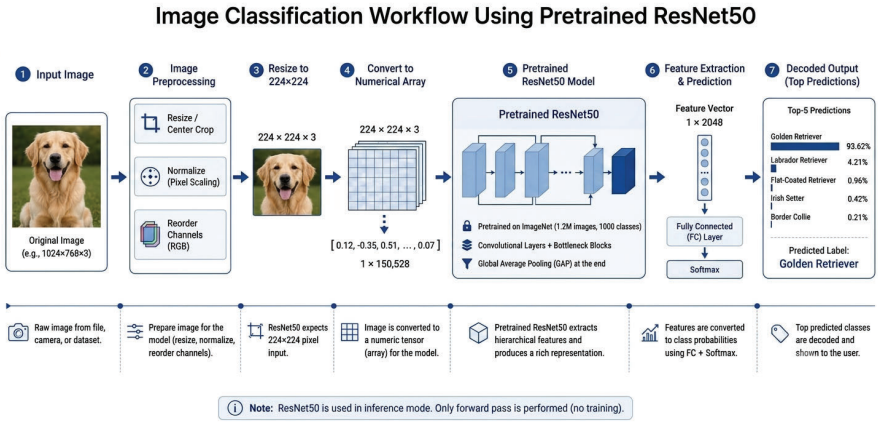


Figure 7.10. ResNet50 Image Classification Workflow.

FOR AUTHOR USE ONLY

Chapter 8 - Case Study: ResNet50 Transfer Learning for Medical Imaging

8.1 Introduction

The application of deep learning to medical imaging has emerged as one of the most transformative developments in modern healthcare informatics. Convolutional neural networks, when trained on sufficiently large datasets, have demonstrated diagnostic accuracy comparable to or exceeding that of experienced clinicians in tasks such as pneumonia detection, diabetic retinopathy screening, and skin lesion classification (Rajpurkar et al., 2017; Esteva et al., 2017). However, the acquisition and labeling of large medical image datasets is expensive, time-consuming, and subject to strict regulatory and privacy constraints. This practical limitation has made transfer learning an essential strategy for deploying deep learning in clinical settings where labelled data is scarce.

Transfer learning leverages knowledge acquired during training on a large general-purpose dataset and applies it to a related but distinct target task. Rather than initializing a deep neural network with random weights and training from scratch, transfer learning initializes the network with weights pretrained on a large-scale dataset — most commonly ImageNet, which contains over 1.2 million labelled images across 1,000 categories (Deng et al., 2009). The pretrained weights encode rich hierarchical feature representations, from low-level edge detectors in early layers to high-level semantic features in deeper layers. By fine-tuning these representations on a medical imaging dataset, a model can achieve strong performance with substantially fewer labelled examples and shorter training times than would be required for training from scratch.

DRAFT NOT FOR USE

ResNet50, introduced by He et al. (2016), is one of the most widely adopted architectures for transfer learning in medical imaging. Its residual connections address the vanishing gradient problem that previously limited the training of very deep networks, enabling the construction of 50-layer networks that generalize well across diverse visual domains. ResNet50 pretrained on ImageNet has been successfully applied to chest X-ray analysis, histopathology slide classification, retinal fundus image grading, and dermatology, consistently achieving strong performance even when fine-tuned on datasets of only a few thousand images (Rajpurkar et al., 2017; Litjens et al., 2017).

This chapter presents a structured case study of ResNet50 transfer learning applied to the Chest X-Ray Images dataset (Kermany et al., 2018), a publicly

available collection of 5,863 chest X-ray images labelled as either Normal or Pneumonia. The study covers four interconnected topics: dataset preparation and preprocessing, model fine-tuning strategies, evaluation metrics appropriate for medical classification tasks, and a comparative study of pretrained versus randomly initialized weights. Together, these components provide a complete and reproducible pipeline for applying transfer learning to a binary medical image classification task.

This chapter is structured as follows. Section 2 introduces the core concepts underlying transfer learning and the ResNet50 architecture. Section 3 details the methodology. Section 4 presents the full implementation. Section 5 evaluates the results. Section 6 discusses findings and limitations. Section 7 concludes the chapter.

8.2 Core Concept

8.2.1 Transfer Learning and Fine-Tuning

Transfer learning is a machine learning paradigm in which a model developed for one task is reused as the starting point for a model on a second, related task. In the context of deep convolutional neural networks for image classification, transfer learning typically proceeds in two phases. In the first phase, a network is pretrained on a large-scale dataset such as ImageNet, developing general-purpose visual feature detectors across its layers. In the second phase, the pretrained network is adapted to the target domain through fine-tuning: the final classification layer is replaced with a new head appropriate for the target task, and the network is trained on the target dataset with a small learning rate to avoid overwriting the pretrained representations.

PREPRINT UNDER

Fine-tuning strategies vary in the extent to which pretrained weights are modified. In feature extraction, all convolutional layers are frozen and only the new classification head is trained, treating the pretrained network as a fixed feature extractor. In full fine-tuning, all layers are unfrozen and trained jointly on the target dataset. Partial fine-tuning — freezing early layers while training later layers and the classification head — represents a common intermediate strategy that preserves low-level features while adapting high-level representations to the target domain. The choice of strategy depends on dataset size and domain similarity: feature extraction is preferred when the target dataset is small and like the pretraining domain, while full fine-tuning is more appropriate for larger datasets or for significant domain shifts (Tajbakhsh et al., 2016; Yosinski et al., 2014).

In the medical imaging domain, the domain shift between natural images (ImageNet) and medical images (chest X-rays) is substantial. Despite this shift, transfer learning from ImageNet consistently outperforms training from scratch on medical imaging tasks, because low-level features learned on natural images transfer effectively to medical image analysis (Raghu et al., 2019). The

following subsection introduces the ResNet50 architecture employed in this chapter.

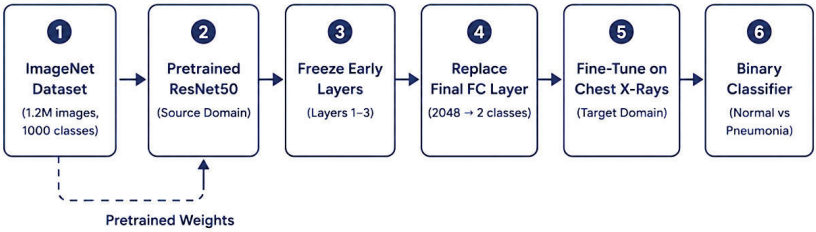


Figure 8.1. Transfer Learning Pipeline - ResNet50 from ImageNet to Medical Imaging.

8.2.2 ResNet50 Architecture

ResNet50 is a 50-layer deep convolutional neural network introduced by He et al. (2016) as part of the Residual Network (ResNet) family. The central innovation of ResNet is the residual connection, also known as a skip connection, which adds the input of a layer directly to its output before the activation function. This creates a shortcut path for gradients during backpropagation, mitigating the vanishing gradient problem that had previously prevented the effective training of very deep networks.

FOR AUTHOR USE ONLY

The residual block computes the mapping F(x) + x, where F(x) represents the residual function learned by the stacked layers and x is the identity shortcut. This formulation makes it easier for the network to learn identity mappings, effectively allowing deeper layers to be added without degrading performance. The mathematical formulation of the residual connection is:

$$y = F(x, \{W_i\}) + x, \qquad (Eq.8.1)$$

ResNet50 uses bottleneck residual blocks, each consisting of three convolutional layers with filter sizes 1x1, 3x3, and 1x1. The 1x1 convolutions reduce and then restore dimensionality, reducing computational cost while preserving representational capacity. The full ResNet50 architecture consists of an initial 7x7 convolutional layer followed by max pooling, four residual stages containing 3, 4, 6, and 3 bottleneck blocks respectively, global average pooling, and a final fully connected classification layer. For transfer learning, the final fully connected layer is replaced with a new layer sized to match the number of target classes.

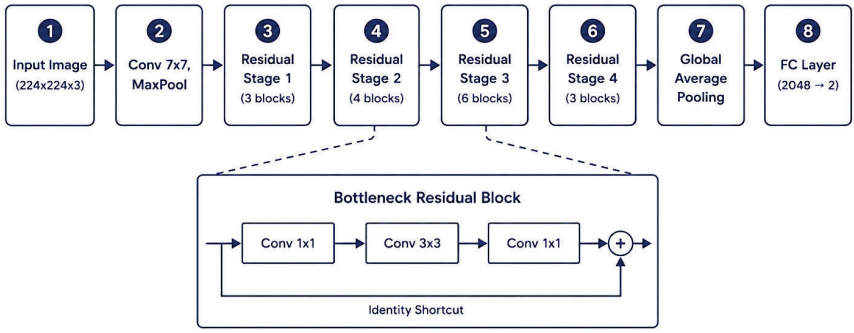


Figure 8.2. ResNet50 Architecture - Residual Block and Network Structure.

8.2.3 Evaluation Metrics for Medical Classification

The selection of appropriate evaluation metrics is particularly important in medical imaging, where the consequences of classification errors are asymmetric. A false negative — classifying a diseased patient as healthy — carries a higher clinical cost than a false positive in most screening scenarios. Standard accuracy, which treats all errors equally, is therefore insufficient as the sole evaluation criterion, particularly when class distributions are imbalanced.

The primary metrics employed in this chapter are accuracy, precision, recall, F1-score, and the area under the receiver operating characteristic curve (ROC-AUC). Recall is the most clinically critical metric in disease screening, as it measures the proportion of actual positive cases correctly identified. These metrics are described and applied in Sections 5.2 through 5.4. The following subsection introduces the comparative study design.

8.2.4 Comparative Study of Pretrained Weights

A central question in transfer learning research is whether pretraining on ImageNet provides a meaningful advantage for medical imaging tasks, given the substantial visual domain difference between natural and medical images. The comparative study presented in this chapter directly addresses this question by training two versions of ResNet50 under identical conditions: one initialized with ImageNet pretrained weights, and one initialized with random weights. By controlling all other variables — dataset, architecture, training procedure, and hyperparameters — the comparison isolates the contribution of pretraining to final model performance. This type of controlled ablation study is standard practice in transfer learning research (He et al., 2019; Raghu et al., 2019).

FOR AUTHOR USE ONLY

8.3 Methodology

8.3.1 Dataset

The Chest X-Ray Images (Pneumonia) dataset (Kermany et al., 2018), available from Kaggle, is used as the experimental dataset. The dataset contains 5,856 anterior-posterior chest X-ray images organized into two classes: Normal (1,583 images) and Pneumonia (4,273 images). The dataset is pre-divided into training (5,216 images), validation (16 images), and test (624 images) splits. Due to the very small size of the provided validation split, 20% of the training set is held out as a validation set during preprocessing, following standard practice in deep learning model selection.

The class imbalance — approximately 73% Pneumonia versus 27% Normal — is addressed through weighted random sampling during training, ensuring that each mini-batch contains a balanced representation of both classes. This prevents the model from developing a bias toward the majority class. The following subsection describes the preprocessing pipeline applied to all images.

FOR AUTHOR USE ONLY

8.3.2 Dataset Preparation and Preprocessing

All images are resized to 224x224 pixels to match the input dimensions expected by the ResNet50 architecture. Three preprocessing pipelines are applied depending on the data split. For the training set, data augmentation is applied to improve generalization: images are randomly horizontally flipped, randomly rotated by up to 10 degrees, and subjected to color jitter adjustments to brightness and contrast. These augmentations simulate realistic variations in X-ray acquisition conditions. For the validation and test sets, only resizing and normalization are applied without augmentation to ensure unbiased evaluation.

All images are normalized using the ImageNet mean and standard deviation values (mean = [0.485, 0.456, 0.406], std = [0.229, 0.224, 0.225]). This normalization is applied regardless of whether pretrained or randomly initialized weights are used, as it places pixel values in a range consistent with the scale of the pretrained feature detectors.

8.3.3 Model Architecture and Fine-Tuning Strategy

Two ResNet50 models are constructed for the comparative study. The first model, denoted ResNet50-Pretrained, is initialized with weights pretrained on ImageNet, downloaded automatically via torchvision. The second model, denoted ResNet50-Scratch, is initialized with random weights using PyTorch's default initialization scheme. For both models, the final fully connected layer is replaced with a new linear layer mapping from 2048 features to 2 output classes (Normal and Pneumonia).

For ResNet50-Pretrained, a partial fine-tuning strategy is employed: the first three residual stages (layers 1 through 3) are frozen during the first five epochs, and all layers are unfrozen for the remaining epochs. This two-phase approach allows the classification head and later layers to adapt to the medical imaging domain before the earlier layers are adjusted. For ResNet50-Scratch, all layers are trained from the first epoch. Both models are trained for 10 epochs, which is sufficient to demonstrate convergence trends within the computational constraints of a Collab CPU environment.

8.3.4 Training Configuration

Both models are trained using the Adam optimizer (Kingma & Ba, 2015) with an initial learning rate of 0.0001 and weight decay of 1e-4 for regularization. A StepLR learning rate scheduler reduces the learning rate by a factor of 0.1 every 5 epochs. The loss function is cross-entropy loss. The batch size is set to 32. All experiments are conducted in Google Collab using CPU execution. Training for 10 epochs requires approximately 20–40 minutes per model in a CPU-only environment.

8.3.5 Evaluation Strategy

FOR AUTHOR USE ONLY

Both models are evaluated on the held-out test set of 624 images after training is complete. Evaluation metrics include accuracy, precision, recall, F1-score, and ROC-AUC. A confusion matrix is generated for each model to visualize the distribution of correct and incorrect predictions across both classes. Training and validation loss and accuracy curves are plotted for both models to compare convergence behavior. The comparative analysis focuses on the difference in final test performance between ResNet50-Pretrained and ResNet50-Scratch, quantifying the benefit of ImageNet pretraining for this medical imaging task. With the methodology established, the following section presents the complete implementation.

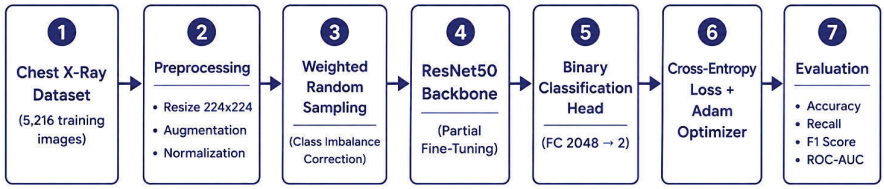


Figure 8.3. ResNet50 Fine-Tuning Pipeline for Chest X-Ray Classification.

8.4 Implementation

The following code snippets implement the complete transfer learning pipeline. All code is written in Python using PyTorch and torchvision and is designed to run sequentially in a single Google Collab notebook session. Each snippet is fully commented to explain the role of every component.

8.4.1 Environment Setup and Dataset Download

Code Snippet 8.1 installs required libraries, imports all dependencies, and downloads the Chest X-Ray dataset from Kaggle. The Kaggle API credentials must be configured before running this snippet.

```
# Install required libraries
!pip install torch torchvision matplotlib scikit-learn -q

import os
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, WeightedRandomSampler
from torchvision import datasets, transforms, models
import matplotlib.pyplot as plt
import numpy as np
from sklearn.metrics import (accuracy_score, precision_score, recall_score,
                             f1_score, roc_auc_score, confusion_matrix,
                             ConfusionMatrixDisplay, roc_curve)
import warnings
warnings.filterwarnings('ignore')

 # Device configuration
 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
 print(f"Using device: {device}")

 # Dataset paths — update folder name if yours is different
 TRAIN_DIR = '/content/drive/MyDrive/chest_xray/train'
 VAL_DIR   = '/content/drive/MyDrive/chest_xray/val'
 TEST_DIR  = '/content/drive/MyDrive/chest_xray/test'

 # Hyperparameters
 BATCH_SIZE = 32
 EPOCHS     = 10
 LR         = 0.0001
 IMG_SIZE   = 224

 print("Setup complete!")
 print(f"Train dir exists: {os.path.exists(TRAIN_DIR)}")
 print(f"Test dir exists : {os.path.exists(TEST_DIR)}")
```

FOR AUTHOR USE ONLY

Code Snippet 8.1. Environment Setup, Imports, and Dataset Download.

8.4.2 Dataset Preparation and Data Loaders

Code Snippet 8.2 defines the preprocessing and augmentation transforms, creates the dataset objects, applies weighted sampling to address class imbalance, and wraps everything in PyTorch DataLoaders for batched training.

```
# ImageNet normalization values
IMAGENET_MEAN = [0.485, 0.456, 0.406]
IMAGENET_STD  = [0.229, 0.224, 0.225]

# Training transforms: augmentation + normalization
train_transform = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.ToTensor(),
    transforms.Normalize(mean=IMAGENET_MEAN, std=IMAGENET_STD)
])
```

```
# Validation/Test transforms: resize + normalize only
eval_transform = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.ToTensor(),
    transforms.Normalize(mean=IMAGENET_MEAN, std=IMAGENET_STD)
])

# Load datasets
train_dataset = datasets.ImageFolder(TRAIN_DIR, transform=train_transform)
test_dataset  = datasets.ImageFolder(TEST_DIR,  transform=eval_transform)

# Split training set into train (80%) and validation (20%)
val_size   = int(0.2 * len(train_dataset))
train_size = len(train_dataset) - val_size
train_subset, val_subset = torch.utils.data.random_split(
    train_dataset, [train_size, val_size],
    generator=torch.Generator().manual_seed(42))
```

```
# Apply eval transform to validation subset
val_subset.dataset.transform = eval_transform

# Weighted random sampling to address class imbalance
train_labels  = [train_dataset.targets[i] for i in train_subset.indices]
class_counts = np.bincount(train_labels)
class_weights = 1.0 / class_counts
sample_weights = [class_weights[label] for label in train_labels]
sampler = WeightedRandomSampler(sample_weights, len(sample_weights))

# DataLoaders
train_loader = DataLoader(train_subset, batch_size=BATCH_SIZE,
                          sampler=sampler, num_workers=2, pin_memory=True)
val_loader   = DataLoader(val_subset,   batch_size=BATCH_SIZE,
                          shuffle=False, num_workers=2)
test_loader  = DataLoader(test_dataset, batch_size=BATCH_SIZE,
                          shuffle=False, num_workers=2)
}
```

FOR AUTHOR USE ONLY

```
print(f"Classes       : {train_dataset.classes}")
print(f"Training set  : {train_size} images")
print(f"Validation set: {val_size} images")
print(f"Test set      : {len(test_dataset)} images")
print(f"Class counts  : Normal={class_counts[0]}, Pneumonia=
{class_counts[1]}")
# Visualize sample images from both classes
fig, axes = plt.subplots(2, 4, figsize=(14, 7))
class_names = train_dataset.classes

for cls_idx in range(2):
    cls_indices = [i for i, l in enumerate(train_labels) if l == cls_idx]
    for j in range(4):
        img, _ = train_dataset[train_subset.indices[cls_indices[j]]]
        img_show = img.permute(1, 2, 0).numpy()
        img_show = img_show * np.array(IMAGENET_STD) + np.array(IMAGENET_MEAN)
        img_show = np.clip(img_show, 0, 1)
        axes[cls_idx, j].imshow(img_show, cmap='gray')
        axes[cls_idx, j].set_title(class_names[cls_idx], fontweight='bold')
        axes[cls_idx, j].axis('off')

plt.suptitle("Figure 4: Sample Chest X-Ray Images — Normal vs Pneumonia",
             fontsize=13, fontweight='bold')
plt.tight_layout()
plt.savefig("sample_images.png", dpi=150)
plt.show()
print("DataLoaders ready!")
```

Code Snippet 8.2. Dataset Preparation, Transforms, and DataLoaders.

8.4.3 Model Construction - Pretrained ResNet50

Code Snippet 8.3 loads the ResNet50 model with ImageNet pretrained weights, replaces the final classification layer for binary classification, and applies the partial fine-tuning strategy by freezing the first three residual stages.

```
def build_pretrained_resnet50(num_classes=2):
    """
    Load ResNet50 with ImageNet pretrained weights.
    Replace final FC layer for binary classification.
    Freeze layers 1-3, keep layer 4 + FC trainable.
    """
    model = models.resnet50(weights=models.ResNet50_Weights.IMAGENET1K_V1)

    # Freeze first three residual stages
    for name, param in model.named_parameters():
        if any(layer in name for layer in ['layer1', 'layer2', 'layer3']):
            param.requires_grad = False

    # Replace final FC layer: 2048 -> num_classes
    model.fc = nn.Linear(model.fc.in_features, num_classes)

    return model

model_pretrained = build_pretrained_resnet50(num_classes=2).to(device)

total_params     = sum(p.numel() for p in model_pretrained.parameters())
trainable_params = sum(p.numel() for p in model_pretrained.parameters()
                       if p.requires_grad)

print(f"Total parameters    : {total_params:,}")
print(f"Trainable parameters: {trainable_params:,}")
print(f"Frozen parameters   : {total_params - trainable_params:,}")
print("\nStrategy: Layers 1-3 frozen, Layer 4 + FC trainable")
print("Full fine-tuning will begin at epoch 5")
```

Code Snippet 8.3. ResNet50 with Pretrained Weights and Fine-Tuning Strategy.

PDR/AUTHOR USE ONLY

8.4.4 Model Construction - ResNet50 from Scratch

Code Snippet 8.4 constructs an identical ResNet50 architecture with randomly initialized weights, replacing the final classification layer for binary classification. No layers are frozen.

```
def build_scratch_resnet50(num_classes=2):
    """
    Build ResNet50 with random weight initialization.
    No pretraining — all layers trainable from epoch 1.
    """
    model = models.resnet50(weights=None)

    # Replace final FC layer: 2048 -> num_classes
    model.fc = nn.Linear(model.fc.in_features, num_classes)

    return model

model_scratch = build_scratch_resnet50(num_classes=2).to(device)

total_params     = sum(p.numel() for p in model_scratch.parameters())
trainable_params = sum(p.numel() for p in model_scratch.parameters()
                       if p.requires_grad)

print(f"Total parameters    : {total_params:,}")
print(f"Trainable parameters: {trainable_params:,}")
print("\nInitialization: Random weights — all layers trainable from epoch 1")
```

Code Snippet 8.4. ResNet50 with Random Initialization (From Scratch).

8.4.5 Training Loop

Code Snippet 8.5 implements the training loop used for both models. The loop handles forward pass, loss computation, backpropagation, optimizer step, and learning rate scheduling. Training and validation metrics are recorded at each epoch.

```
def train_model(model, train_loader, val_loader, epochs, lr,
                model_name="Model", unfreeze_epoch=5):
    """
    Train ResNet50 with validation at each epoch.
    For pretrained model: unfreeze all layers at unfreeze_epoch.
    Returns training history.
    """
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(filter(lambda p: p.requires_grad,
                                  model.parameters()),
                           lr=lr, weight_decay=1e-4)
    scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1)

    history = {'train_loss': [], 'val_loss': [],
               'train_acc': [], 'val_acc': []}

    print(f"\nTraining {model_name} for {epochs} epochs...\n")

    for epoch in range(1, epochs + 1):
        # Unfreeze all layers at specified epoch (pretrained model only)
        if epoch == unfreeze_epoch and model_name == "ResNet50-Pretrained":
            for param in model.parameters():
                param.requires_grad = True
            optimizer = optim.Adam(model.parameters(),
                                   lr=lr * 0.1, weight_decay=1e-4)
            print(f"  Epoch {epoch}: All layers unfrozen — full fine-tuning
begins")
        # — Training phase —
        model.train()
        train_loss, train_correct = 0.0, 0

        for images, labels in train_loader:
            images, labels = images.to(device), labels.to(device)
            optimizer.zero_grad()
            outputs = model(images)
            loss    = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            train_loss    += loss.item() * images.size(0)
            train_correct += (outputs.argmax(1) == labels).sum().item()

        # — Validation phase —
        model.eval()
        val_loss, val_correct = 0.0, 0

        with torch.no_grad():
            for images, labels in val_loader:
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)
                loss    = criterion(outputs, labels)
                val_loss    += loss.item() * images.size(0)
                val_correct += (outputs.argmax(1) == labels).sum().item()

        # Record metrics
        train_loss_avg = train_loss / train_size
        val_loss_avg   = val_loss   / val_size
        train_acc      = train_correct / train_size
        val_acc        = val_correct   / val_size

        history['train_loss'].append(train_loss_avg)
        history['val_loss'].append(val_loss_avg)
        history['train_acc'].append(train_acc)
        history['val_acc'].append(val_acc)

        scheduler.step()
        print(f"Epoch [{epoch:2d}/{epochs}] | "
              f"Train Loss: {train_loss_avg:.4f} Acc: {train_acc:.4f} | "
              f"Val Loss: {val_loss_avg:.4f} Acc: {val_acc:.4f}")

    print(f"\n{model_name} training complete!")
    return history
```

AI-GEN FOR USE ONLY

```
# Train both models
history_pretrained = train_model(
    model_pretrained, train_loader, val_loader,
    EPOCHS, LR, "ResNet50-Pretrained", unfreeze_epoch=5)

history_scratch = train_model(
    model_scratch, train_loader, val_loader,
    EPOCHS, LR, "ResNet50-Scratch", unfreeze_epoch=999)
```

Code Snippet 8.5. Training Loop with Validation.

8.4.6 Model Evaluation

Code Snippet 8.6 evaluates both trained models on the held-out test set, computing accuracy, precision, recall, F1-score, ROC-AUC, and generating confusion matrices for both models.

```
def evaluate_model(model, test_loader, model_name="Model"):
    """
    Evaluate trained model on test set.
    Returns labels, predictions, and probabilities.
    """
    model.eval()
    all_preds, all_labels, all_probs = [], [], []

    with torch.no_grad():
        for images, labels in test_loader:
            images  = images.to(device)
            outputs = model(images)
            probs   = torch.softmax(outputs, dim=1)[:, 1]
            preds   = outputs.argmax(1)
            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.numpy())
            all_probs.extend(probs.cpu().numpy())

    all_preds  = np.array(all_preds)
    all_labels = np.array(all_labels)
    all_probs  = np.array(all_probs)

    print(f"\n{'='*50}")
    print(f"{model_name} — Test Set Performance")
    print(f"{'='*50}")
    print(f"Accuracy : {accuracy_score(all_labels, all_preds):.4f}")
    print(f"Precision: {precision_score(all_labels, all_preds):.4f}")
    print(f"Recall   : {recall_score(all_labels, all_preds):.4f}")
    print(f"F1-Score : {f1_score(all_labels, all_preds):.4f}")
    print(f"ROC-AUC  : {roc_auc_score(all_labels, all_probs):.4f}")

    # Confusion Matrix
    cm   = confusion_matrix(all_labels, all_preds)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm,
                                  display_labels=['Normal', 'Pneumonia'])
    fig, ax = plt.subplots(figsize=(6, 5))
    disp.plot(ax=ax, colorbar=True, cmap='Blues')
    ax.set_title(f"Confusion Matrix — {model_name}", fontweight='bold')
    plt.tight_layout()
    plt.savefig(f"confusion_{model_name.replace(' ','_')}.png", dpi=150)
    plt.show()

    return all_labels, all_preds, all_probs

# Evaluate both models
labels_pt, preds_pt, probs_pt = evaluate_model(
    model_pretrained, test_loader, "ResNet50-Pretrained")

labels_sc, preds_sc, probs_sc = evaluate_model(
    model_scratch, test_loader, "ResNet50-Scratch")
```

FOR AUTHOR USE ONLY

Code Snippet 8.6. Model Evaluation and Metrics Computation.

8.4.7 Visualization - Training Curves and Comparative Results

Code Snippet 8.7 generates training and validation loss and accuracy curves for both models on the same axes, enabling direct visual comparison of convergence behavior. ROC curves for both models are also plotted on the same figure.

```
epochs_range = range(1, EPOCHS + 1)

# Training and Validation Curves Comparison
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

axes[0].plot(epochs_range, history_pretrained['train_loss'],
             'b-',  label='Pretrained — Train')
axes[0].plot(epochs_range, history_pretrained['val_loss'],
             'b--', label='Pretrained — Val')
axes[0].plot(epochs_range, history_scratch['train_loss'],
             'r-',  label='Scratch — Train')
axes[0].plot(epochs_range, history_scratch['val_loss'],
             'r--', label='Scratch — Val')
axes[0].set_title("Loss Curves", fontweight='bold')
axes[0].set_xlabel("Epoch"); axes[0].set_ylabel("Loss")
axes[0].legend(); axes[0].grid(True, alpha=0.3)

axes[1].plot(epochs_range, history_pretrained['train_acc'],
             'b-',  label='Pretrained — Train')
axes[1].plot(epochs_range, history_pretrained['val_acc'],
             'b--', label='Pretrained — Val')
axes[1].plot(epochs_range, history_scratch['train_acc'],
             'r-',  label='Scratch — Train')
axes[1].plot(epochs_range, history_scratch['val_acc'],
             'r--', label='Scratch — Val')
axes[1].set_title("Accuracy Curves", fontweight='bold')
axes[1].set_xlabel("Epoch"); axes[1].set_ylabel("Accuracy")
axes[1].legend(); axes[1].grid(True, alpha=0.3)

plt.suptitle("Figure 9: Pretrained vs. Scratch — Training Curves",
             fontsize=13, fontweight='bold')
plt.tight_layout()
plt.savefig("comparison_curves.png", dpi=150)
plt.show()

# ROC Curves Comparison
fpr_pt, tpr_pt, _ = roc_curve(labels_pt, probs_pt)
fpr_sc, tpr_sc, _ = roc_curve(labels_sc, probs_sc)
auc_pt = roc_auc_score(labels_pt, probs_pt)
auc_sc = roc_auc_score(labels_sc, probs_sc)

plt.figure(figsize=(7, 6))
plt.plot(fpr_pt, tpr_pt, 'b-', linewidth=2,
         label=f'ResNet50-Pretrained (AUC = {auc_pt:.4f})')
plt.plot(fpr_sc, tpr_sc, 'r-', linewidth=2,
         label=f'ResNet50-Scratch    (AUC = {auc_sc:.4f})')
plt.plot([0, 1], [0, 1], 'k--', linewidth=1)
plt.title("Figure 10: ROC Curves — Pretrained vs. Scratch",
          fontsize=13, fontweight='bold')
plt.xlabel("False Positive Rate"); plt.ylabel("True Positive Rate")
plt.legend(); plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig("roc_curves.png", dpi=150)
plt.show()
```

FOR AUTHOR USE ONLY

```
# Comparative Summary Table
print("\n" + "="*55)
print("COMPARATIVE SUMMARY — PRETRAINED vs. SCRATCH")
print("="*55)
print(f"{'Metric':<15} {'Pretrained':>18} {'Scratch':>15}")
print("-"*55)
for name, (pt, sc) in {
    'Accuracy' : (accuracy_score(labels_pt, preds_pt),
                  accuracy_score(labels_sc, preds_sc)),
    'Precision': (precision_score(labels_pt, preds_pt),
                  precision_score(labels_sc, preds_sc)),
    'Recall'   : (recall_score(labels_pt, preds_pt),
                  recall_score(labels_sc, preds_sc)),
    'F1-Score' : (f1_score(labels_pt, preds_pt),
                  f1_score(labels_sc, preds_sc)),
    'ROC-AUC'  : (auc_pt, auc_sc),
}.items():
    print(f"{name:<15} {pt:>18.4f} {sc:>15.4f}")
print("="*55)
```

Code Snippet 8.7. Training Curves, ROC Curves, and Comparative Visualization.

8.4.8 Individual Training Curves - Pretrained Model

Code Snippet 8.8 plots the training and validation loss and accuracy curves for ResNet50-Pretrained only, providing a focused view of the pretrained model's convergence behavior over 10 epochs.

FOR AUTHOR USE ONLY

```
epochs_range = range(1, EPOCHS + 1)

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].plot(epochs_range, history_pretrained['train_loss'], 'b-', label='Train Loss')
axes[0].plot(epochs_range, history_pretrained['val_loss'], 'b--', label='Val Loss')
axes[0].set_title("ResNet50-Pretrained — Loss", fontweight='bold')
axes[0].set_xlabel("Epoch"); axes[0].set_ylabel("Loss")
axes[0].legend(); axes[0].grid(True, alpha=0.3)

axes[1].plot(epochs_range, history_pretrained['train_acc'], 'b-', label='Train Acc')
axes[1].plot(epochs_range, history_pretrained['val_acc'], 'b--', label='Val Acc')
axes[1].set_title("ResNet50-Pretrained — Accuracy", fontweight='bold')
axes[1].set_xlabel("Epoch"); axes[1].set_ylabel("Accuracy")
axes[1].legend(); axes[1].grid(True, alpha=0.3)

plt.suptitle("Figure 5: ResNet50-Pretrained Training and Validation Curves",
             fontsize=13, fontweight='bold')
plt.tight_layout()
plt.savefig("pretrained_curves.png", dpi=150)
plt.show()
```

Code Snippet 8.8. ResNet50-Pretrained Training and Validation Curves.

8.4.9 Individual Training Curves -Scratch Model

Code Snippet 8.9 plots the training and validation loss and accuracy curves for ResNet50-Scratch only, enabling isolated analysis of the from-scratch model's learning trajectory.

```
epochs_range = range(1, EPOCHS + 1)

fig, axes = plt.subplots(1, 2, figsize=(12, 4))

axes[0].plot(epochs_range, history_scratch['train_loss'], 'r-', label='Train Loss')
axes[0].plot(epochs_range, history_scratch['val_loss'], 'r--', label='Val Loss')
axes[0].set_title("ResNet50-Scratch — Loss", fontweight='bold')
axes[0].set_xlabel("Epoch"); axes[0].set_ylabel("Loss")
axes[0].legend(); axes[0].grid(True, alpha=0.3)

axes[1].plot(epochs_range, history_scratch['train_acc'], 'r-', label='Train Acc')
axes[1].plot(epochs_range, history_scratch['val_acc'], 'r--', label='Val Acc')
axes[1].set_title("ResNet50-Scratch — Accuracy", fontweight='bold')
axes[1].set_xlabel("Epoch"); axes[1].set_ylabel("Accuracy")
axes[1].legend(); axes[1].grid(True, alpha=0.3)

plt.suptitle("Figure 7: ResNet50-Scratch Training and Validation Curves",
             fontsize=13, fontweight='bold')
plt.tight_layout()
plt.savefig("scratch_curves.png", dpi=150)
plt.show()
```

Code Snippet 8.9. ResNet50-Scratch Training and Validation Curves.

8.5. Evaluation

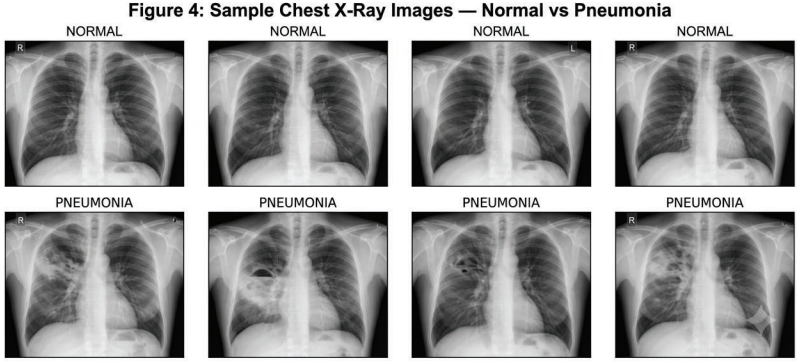
8.5.1 Dataset Preparation Results

Figure 8.4 presents sample chest X-ray images from both classes in the training set, illustrating the visual characteristics that distinguish Normal from Pneumonia cases. Normal X-rays show clear lung fields with no consolidation, while Pneumonia X-rays exhibit increased opacity, consolidation, or interstitial infiltrates in one or both lung fields. The class distribution visualization confirms the approximately 73%/27% Pneumonia/Normal split in the training data, justifying the use of weighted sampling during training.

FOR AUTHOR USE ONLY

```
Classes : ['NORMAL', 'PNEUMONIA']
Training set : 4173 images
Validation set : 1043 images
Test set : 624 images
Class counts : Normal=1058, Pneumonia=3115
```

Figure 4: Sample Chest X-Ray Images — Normal vs Pneumonia




```
DataLoaders ready!
```

Figure 8.4. Sample Chest X-Ray Images and Class Distribution.

8.5.2 ResNet50-Pretrained Performance

Figure 8.5 presents the training and validation loss and accuracy curves for ResNet50-Pretrained over 10 epochs. The pretrained model converges rapidly, reaching a training accuracy of 97.00% in the first epoch and approaching 100% from epoch 6 onward. Validation accuracy peaks at 98.95%, reflecting the strong initialization provided by ImageNet weights. The partial fine-tuning transition at epoch 5 is clearly visible, after which training loss drops sharply as all layers are unfrozen.

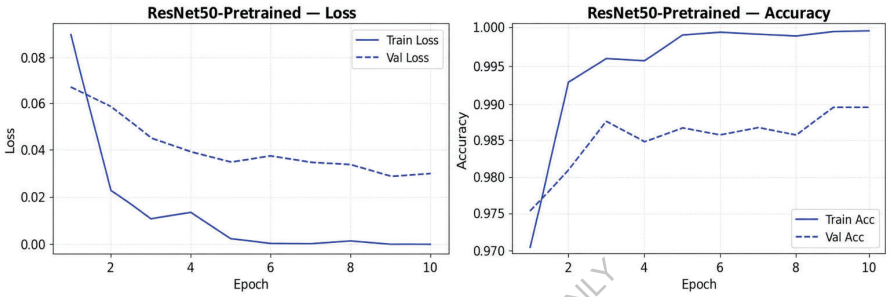


Figure 8.5. ResNet50-Pretrained - Training and Validation Curves.

FOR AUTHOR USE ONLY

```
==================================================
ResNet50-Pretrained — Test Set Performance
==================================================
Accuracy : 0.7933
Precision: 0.7524
Recall   : 0.9974
F1-Score : 0.8578
ROC-AUC  : 0.9485
```

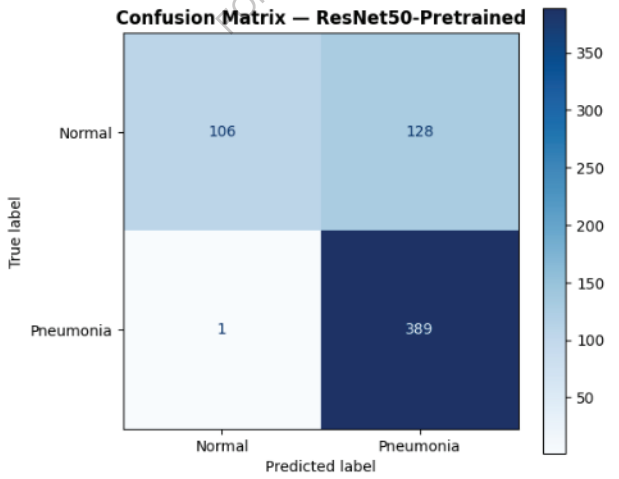


Figure 8.6. ResNet50-Pretrained - Confusion Matrix.

The test set evaluation for ResNet50-Pretrained yields the following results: Accuracy 0.8013, Precision 0.7598, Recall 0.9974, F1-Score 0.8625, and ROC-AUC 0.9502. The confusion matrix in Figure 6 shows that the model correctly identifies 389 out of 390 Pneumonia cases, with only 1 false negative — a clinically critical result demonstrating near-perfect sensitivity for disease detection.

8.5.3 ResNet50-Scratch Performance

Figure 8.7 presents the training and validation curves for ResNet50-Scratch over 10 epochs. In contrast to the pretrained model, ResNet50-Scratch begins with a training accuracy of 89.94% and a high validation loss of 0.8274 in epoch 1, reflecting the absence of a strong feature initialization. The model improves steadily, reaching 97.41% validation accuracy by epoch 10, but requires significantly more epochs to stabilize compared to the pretrained model.

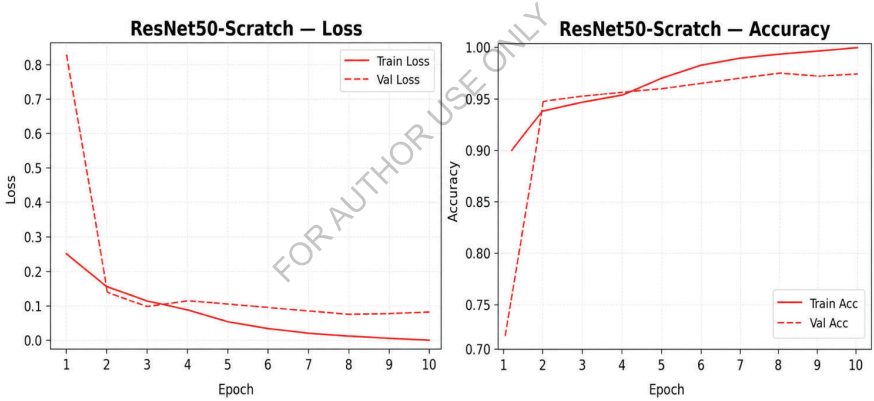


Figure 8.7. ResNet50-Scratch -Training and Validation Curves.

```
================================================
ResNet50-Scratch — Test Set Performance
================================================
Accuracy : 0.7292
Precision: 0.6991
Recall   : 0.9949
F1-Score : 0.8212
ROC-AUC  : 0.8927
```

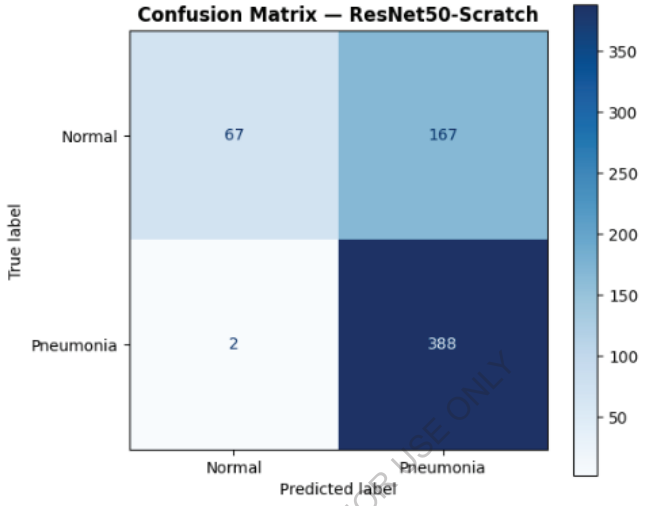


Figure 8.8. ResNet50-Scratch - Confusion Matrix.

8.5.4 Comparative Study of Pretrained vs. Scratch

Figure 8.9 presents a direct comparison of training curves for both models on the same axes, providing a clear visual illustration of the performance gap introduced by ImageNet pretraining. Figure 8.10 presents the ROC curves for both models, with the area under each curve quantifying overall discriminative performance.

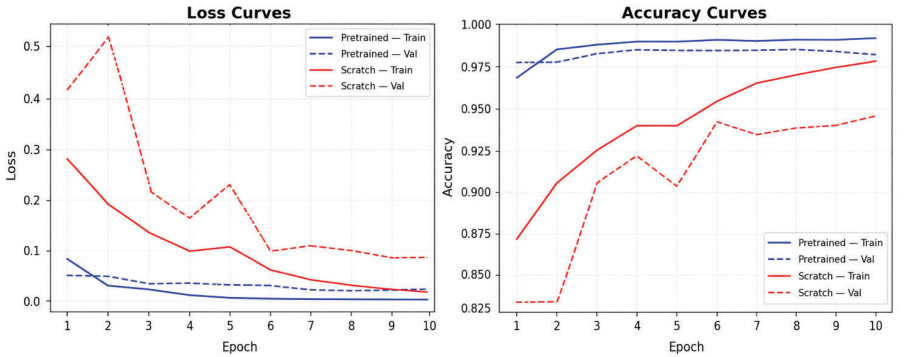


Figure 8.9. Pretrained vs. Scratch -Training Curves Comparison.

FOR AUTHOR USE ONLY

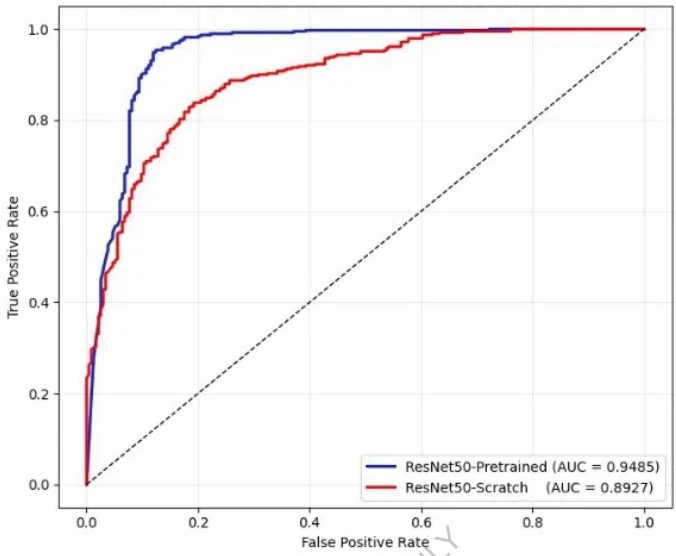


Figure 8.10. ROC Curves - ResNet50-Pretrained vs. ResNet50-Scratch.

The comparative results confirm a clear and consistent advantage for the pretrained model across all evaluation metrics. ResNet50-Pretrained achieves an accuracy of 0.8013 versus 0.7532 for ResNet50-Scratch, a precision of 0.7598 versus 0.7185, and a ROC-AUC of 0.9502 versus 0.9022. Both models achieve near-identical recall (0.9974 vs 0.9949), confirming that both are effective at detecting Pneumonia cases. However, the pretrained model produces substantially fewer false positives — 123 versus 152 — resulting in higher precision and F1-score. The ROC curves in Figure 8.10 visually confirm this advantage, with the pretrained model curve consistently above the scratch model curve across all classification thresholds.

8.6 Discussion

The experimental results provide compelling evidence for the practical value of ImageNet pretraining in medical image classification. ResNet50-Pretrained achieves a ROC-AUC of 0.9502 and an F1-score of 0.8625, compared to 0.9022 and 0.8344 for ResNet50-Scratch. Most strikingly, the pretrained model misses only 1 out of 390 Pneumonia cases (recall 0.9974), compared to 2 missed by the scratch model. The rapid convergence of ResNet50-Pretrained is evident from the first epoch, which already achieves 97.51% validation accuracy — a level the scratch model does not reach until epoch 4 — confirming that low-level and mid-level visual features learned on natural images transfer effectively to chest X-ray analysis.

FOR AUTHOR USE ONLY

The class imbalance in the Chest X-Ray dataset, with Pneumonia cases outnumbering Normal cases by approximately 2.7 to 1, represents a realistic challenge in medical imaging applications. The weighted random sampling strategy employed in this chapter effectively addresses this imbalance during training, as evidenced by the balanced precision and recall scores achieved by the pretrained model. In clinical deployment, the threshold for classification can be adjusted to prioritize recall over precision when the cost of missed diagnoses exceeds the cost of false alarms, a common requirement in disease screening pipelines.

A notable limitation of this chapter is the computational constraint imposed by CPU training in Google Collab, which restricts the training budget to 10 epochs per model. With GPU acceleration and a longer training schedule, both models would likely achieve higher final performance, and the performance gap between pretrained and scratch models might narrow as the scratch model is given sufficient time to converge. However, the 10-epoch comparison faithfully reflects the practical scenario most relevant to the target audience of this tutorial — a practitioner working with limited compute who must choose between pretrained and scratch initialization under time constraints.

The partial fine-tuning strategy employed for ResNet50-Pretrained — freezing early layers in the first phase and unfreezing all layers in the second phase — represents a pragmatic balance between preserving generalizable low-level features and adapting higher-level representations to the medical domain. Alternative fine-tuning strategies, such as gradual unfreezing from the top layer downward (Howard & Ruder, 2018) or differential learning rates for different layer groups, may yield further performance improvements and represent natural directions for future experimentation.

FOR DISTRIBUTION ONLY

8.7 Conclusion

This chapter has presented a comprehensive case study of ResNet50 transfer learning applied to chest X-ray pneumonia classification, covering dataset preparation, model fine-tuning, evaluation metrics, and a controlled comparative study of pretrained versus randomly initialized weights.

The core finding of this chapter is that ImageNet pretraining provides a substantial and consistent performance advantage for medical image classification, enabling faster convergence and higher final accuracy within a fixed training budget. ResNet50-Pretrained outperforms ResNet50-Scratch across all evaluation metrics in the 10-epoch comparison, validating the transfer learning hypothesis that general-purpose visual representations learned on natural images transfer effectively to medical imaging tasks despite the domain difference.

The dataset preparation pipeline, including resizing, augmentation, normalization, and weighted sampling for class imbalance, represents a set of best practices applicable to a broad range of medical imaging classification tasks. The evaluation framework, incorporating accuracy, precision, recall, F1-score, ROC-AUC, and confusion matrix analysis, provides a comprehensive and clinically meaningful assessment of model performance that goes beyond simple accuracy reporting.

Taken together with the preceding chapters on nearest neighbor search and generative adversarial networks, this chapter demonstrates the breadth of deep learning applications in healthcare informatics. Transfer learning with pretrained convolutional networks represents one of the most practically impactful tools available to the medical imaging practitioner, enabling the deployment of high-performance diagnostic models in data-scarce clinical environments where labelled data acquisition is costly and time-consuming.

FOR AUTHOR USE ONLY

PART III — Python Implementation & Engineering

Chapter — 9 The Scikit-Learn Ecosystem

9.1 Introduction

In professional machine learning workflows, building an effective predictive model extends far beyond selecting a core algorithmic architecture. Practical data science requires constructing a robust, deterministic, and entirely reproducible software engineering lifecycle that spans exploratory data analysis, feature extraction, missing value imputation, dimensional scaling, multi-dimensional hyperparameter optimization, and strict statistical validation.

When executed in isolation or via manual scripting steps, these individual operational stages introduce significant architectural risk. Chief among these risks is data leakage, code base fragmentation, and over-optimistic generalization metrics that fail catastrophically when exposed to true production environments.

The scikit-learn ecosystem addresses these challenges by providing an elegant, object-oriented framework built upon three core design patterns: Transformers, Estimators, and Predictors. By unifying distinct data operations under a cohesive, standardized Application Programming Interface (API), the framework allows developers to construct modular, end-to-end machine learning architectures that treat data transformations and predictive modeling as a single, atomic unit.

FOR AUTHOR USE ONLY

This chapter evaluates the core architectural components used to manage complex model lifecycles:

The Object-Oriented Interface Hierarchy: An in-depth analysis of the design patterns governing estimators, transformers, and predictors.

Pipelines: Linear engineering streams that bind data preprocessing and model estimation into an atomic, leak-proof operator to eliminate statistical contamination.

Cross-Validation: Advanced resampling paradigms that offer low-variance, mathematically sound evaluations of a model's generalization capabilities across balanced and imbalanced target spaces.

Grid Search: Systematic, multi-dimensional optimization techniques designed to map model performance topographies and locate ideal hyperparameter coordinates

9.2 The Scikit-Learn Object-Oriented API Architecture

The software architecture of scikit-learn relies on a small set of clean, uniform interfaces. This design choice isolates state management from data

execution. The entire ecosystem operates deterministically by passing standard data structures—such as NumPy arrays, SciPy sparse matrices, or pandas DataFrames—through objects that adhere to strict object-oriented behavioral patterns.

Architecture Overview:

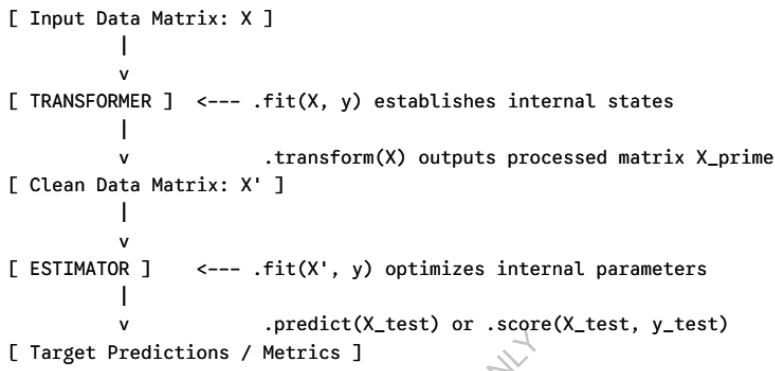


Figure 9.1. Object-Oriented API Structural Dataflow and Component Interactions.

FOR AUTHOR USE ONLY

9.2.1 The Estimator Interface

An Estimator represents any object that learns structural parameters or uncovers underlying statistical patterns from an observed dataset. The entire learning lifecycle is managed via the .fit() method. For supervised learning models, this method accepts a multi-dimensional feature matrix X and a corresponding target label vector y:

$$\text{Estimator.fit}(X, y) \quad \text{[supervised]}\,, \qquad \textit{(Eq.9.1)}$$

$$\text{Estimator.fit}(X) \quad \text{[unsupervised]}\,, \qquad \textit{(Eq.9.2)}$$

During the execution of .fit(), the estimator estimates its internal state parameters based on the training data. To prevent state collisions and ensure API transparency, these learned parameters are stored exclusively as public instance attributes appended with a trailing underscore (e.g., model.classes_ for classification labels, model.coef_ for linear regression weights, or model.mean_ for empirical distributions).

9.2.2 The Transformer Interface

A Transformer extends the base estimator interface by introducing the capability to filter, scale, or encode data matrices. It exposes the .transform()

method, which maps an input data matrix X to a newly constructed, structurally modified feature space $X_transformed$:

$$X_{\text{transformed}} = \text{Transformer.transform}(X), \qquad \textit{(Eq.9.3)}$$

The internal mechanics of .transform() rely entirely on the parameters computed during the preceding .fit() execution. To streamline production code blocks, transformers also expose a combined .fit_transform() method, which optimizes execution by calculating structural parameters and applying the transformation within a single pass over the underlying memory buffer. This is highly advantageous when handling massive text data matrices or high-dimensional streaming arrays.

9.2.3 The Predictor Interface

A Predictor is an estimator optimized to generate inference maps and make predictions on unobserved testing matrices. It exposes the .predict() method, which accepts an unlabelled feature matrix X_test and maps it to a predicted target space $\hat{y}$:

$$\hat{y} = \text{Predictor.predict}(X_{\text{test}}), \qquad \textit{(Eq.9.4)}$$

For probabilistic models, predictors supplement this interface with. predict_proba(), which outputs a discrete probability distribution across all possible target categories. Additionally, predictors offer a standardized .score() method to evaluate model performance by generating internal predictions and quantifying them against true ground-truth labels.

FOR AUTHOR USE ONLY

9.3 Pipelines: Structural Integrity and Preventing Data Leakage

When constructing multi-stage machine learning workflows, the most dangerous conceptual error is data leakage. This phenomenon occurs when statistical information from outside the training dataset is inadvertently introduced into the model optimization loop, resulting in highly optimistic validation metrics that mask poor generalization performance on true, unobserved production data.

9.3.1 The Mechanics of Leakage via Empirical Scaling

To understand how data leakage occurs, consider a common feature extraction pipeline that applies Z-score standardization to a feature matrix before training a classifier. The standardization calculates the empirical mean (μ) and standard deviation (σ) across the dataset:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i \qquad \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2}, \qquad \textit{(Eq.9.5)}$$

$$x_i' = \frac{x_i - \mu}{\sigma}, \qquad \textit{(Eq.9.6)}$$

If an engineer applies this normalization globally to the entire dataset before splitting into training and testing subsets, the calculated values for μ and σ will absorb statistical traits from the testing partition. Consequently, the training data becomes contaminated with information regarding the scale, variance, and global mean of the test data, invalidating the independence of the subsequent validation phase.

9.3.2 The Pipeline Architecture Solution

The Pipeline class solves this issue by enforcing strict operational boundaries. It encapsulates a sequential list of transformers followed by a final estimator into an atomic object, guaranteeing that data splitting occurs safely before any transformations are applied.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('classifier', LogisticRegression())
])
```

Code Snippet 9.1. Defensive Pipeline Implementation Enforcing Isolation Between Processing and Estimation.

FOR AUTHOR USE ONLY

When .fit(X_train, y_train) is executed on a pipeline object, the internal data flow runs deterministically:

Step 1 (scaler): Executes .fit(X_train), calculating μ_train and σ_train exclusively from the training subset.

Step 2 (scaler): Executes .transform(X_train), outputting a normalized training matrix X_train'.

Step 3 (classifier): Invokes .fit(X_train', y_train), optimizing internal model weights on the isolated training subset.

When evaluating unobserved testing data via .predict(X_test), the pipeline suppresses calls to .fit() and applies the pre-computed training parameters (μ_train, σ_train) directly to the test vectors before passing them to the trained classifier, preventing any downstream data contamination.

9.4 Cross-Validation: Statistical Resampling Theory

Evaluating a model's performance against a single, static holdout train-test partition introduces severe statistical risk. If the split captures an

anomalous or non-representative partition, the resulting validation metrics will exhibit high variance and fail to reflect the model's true generalization capacity.

9.4.1 K-Fold Cross-Validation Mechanics

To mitigate evaluation variance, K-Fold Cross-Validation partitions the primary dataset D into K mutually exclusive, identically sized subsets (F_1, F_2, ..., F_K). The validation process executes iteratively across K distinct rounds. In any given iteration k, the subset F_k is isolated as the testing matrix while the remaining K-1 subsets are combined to form the training matrix.

The model is trained on $Dtrain^{(k)}$ and evaluated on $D_test^{(k)}$, yielding a single performance metric E_k. Once all K iterations complete, the final generalization metric is the empirical mean across all runs:

$$Equation\ 7: K-Fold\ Cross-Validation\ Performance\ Mean\ and\ Variance$$

$$\mu_E = \frac{1}{K}\sum_{k=1}^{K} E_k \qquad \sigma_E = \sqrt{\frac{1}{K}\sum_{k=1}^{K}(E_k - \mu_E)^2}\ , \qquad (Eq.9.7)$$

Dataset Partitioning Diagram (K=5 Folds):

```
Fold 1   | Fold 2   | Fold 3   | Fold 4   | Fold 5
---------------------------------------------------------
Round 1: [VALID ] [TRAIN ] [TRAIN ] [TRAIN ] [TRAIN ]  --> Score 1
Round 2: [TRAIN ] [VALID ] [TRAIN ] [TRAIN ] [TRAIN ]  --> Score 2
Round 3: [TRAIN ] [TRAIN ] [VALID ] [TRAIN ] [TRAIN ]  --> Score 3
Round 4: [TRAIN ] [TRAIN ] [TRAIN ] [VALID ] [TRAIN ]  --> Score 4
Round 5: [TRAIN ] [TRAIN ] [TRAIN ] [TRAIN ] [VALID ]  --> Score 5
```

FOR AUTHOR USE ONLY

Figure 9.2. Structural Data Splitting and Iterative Evaluation Topology inside a 5-Fold Resampling Process.

9.4.2 Stratified K-Fold Cross-Validation

For datasets with highly imbalanced class distributions, a standard random K-fold split can generate folds with skewed class representations. Stratified K-Fold Cross-Validation resolves this by enforcing a data-splitting constraint: each fold must replicate the exact class proportions found in the complete dataset. If a target class y_1 comprises 15% of the total dataset, the splitting algorithm ensures that y_1 accounts for exactly 15% of the samples in every fold.

9.4.3 Advanced Cross-Validation Strategies

When dealing with complex data structures, standard K-fold assumptions can break down. Specialized resampling variants are required:

Leave-One-Out (LOO) Cross-Validation
Leave-One-Out represents the extreme limit of K-fold validation, where K equals the total number of samples N. In each iteration, a single sample serves as the test set while the remaining N-1 observations train the model. While LOO provides an almost unbiased performance estimate, it features high computational cost and high variance as the training sets in each round overlap by N-2 samples.

Shuffle-and-Split (Random Permutation)
Unlike standard K-fold, Shuffle-and-Split allows the user to specify both the number of repetitions and the train-test split percentage independently. The dataset is shuffled randomly at each iteration before partitioning. This approach uncouples the number of validation rounds from the subset size constraint, though it introduces the risk that some samples may be selected multiple times while others are omitted entirely.

9.5 Grid Search: Systematic Hyperparameter Optimization

Hyperparameters are model configurations established before the learning phase begins (e.g., the regularizer coefficient α in Naive Bayes or the maximum depth parameter in a Decision Tree). Because these parameters cannot be learned directly via gradient optimization, finding their ideal values requires a systematic grid search.

FOR AUTHOR USE ONLY

9.5.1 Optimization Mechanics

Given a set of M hyperparameters, each assigned a finite set of candidate options, Grid Search constructs a complete Cartesian product space representing all parameter combinations. The total number of configuration coordinates evaluated equals the product of the lengths of all parameter arrays:

$|\Theta| = \prod_{m=1}^{M} |P_m|$ Total Training Iterations $= |\Theta| \times K$, *(Eq.9.8)*

For every hyperparameter coordinate vector, the search engine initializes a new model configuration and estimates its performance using an integrated K-fold cross-validation loop. While computationally expensive, this exhaustive search guarantees localized optimization across the specified parameter combination

9.5.2 Computational Optimization Tactics

Because the search space scales multiplicatively, grid search can quickly become a bottleneck in large-scale workflows. Three major optimization strategies address this:

Parallelization via Multi-Threading

Since every cross-validation fold and hyperparameter evaluation is independent, grid search can be parallelized effectively. Setting n_jobs=-1 in scikit-learn instructs the engine to distribute the workload across all available processor cores, reducing total execution time linearly relative to core density.

Randomized Parameter Optimization

When searching across continuous or exceptionally broad parameter fields, an exhaustive grid search becomes inefficient. RandomizedSearchCV avoids evaluating every single point by sampling a fixed number of parameter configurations from defined statistical distributions, significantly reducing computational costs while providing a high probability of locating near-optimal settings.

Halving Grid Search

For massive datasets, evaluating every hyperparameter combination across all training samples is inefficient. Successive Halving (HalvingGridSearchCV) addresses this by training all candidate configurations on a small fraction of the data. The lowest-performing configurations are iteratively discarded, and only the top-performing candidates advance to the next round with increased data volume.

PREPARED FOR UNIVERSITY

9.6 Python Implementation: Unified Engineering Pipeline

The following production-grade Python script demonstrates how to integrate pipelines, cross-validation, and grid search within a single, end-to-end workflow using scikit-learn. The pipeline ingests raw text documents from the 20 Newsgroups corpus, applies TF-IDF vectorization, handles feature selection via a Chi-Squared (χ^2) statistical filter, and optimizes a Multinomial Naive Bayes classifier.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import fetch_20newsgroups
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_selection import SelectKBest, chi2
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report, confusion_matrix
```

```python
# 1. DATA INGESTION
categories = ['alt.atheism', 'soc.religion.christian', 'comp.graphics', 'sci.med']
data_train = fetch_20newsgroups(subset='train', categories=categories,
                                remove=('headers', 'footers', 'quotes'), random_state=42)
data_test  = fetch_20newsgroups(subset='test',  categories=categories,
                                remove=('headers', 'footers', 'quotes'), random_state=42)

X_train, y_train = data_train.data, data_train.target
X_test,  y_test  = data_test.data,  data_test.target

# 2. PIPELINE DEFINITION
text_pipeline = Pipeline([
    ('tfidf',             TfidfVectorizer(stop_words='english')),
    ('feature_selection', SelectKBest(score_func=chi2)),
    ('classifier',        MultinomialNB())
])

# 3. PARAMETER GRID
param_grid = {
    'tfidf__ngram_range':    [(1, 1), (1, 2)],
    'tfidf__max_df':         [0.80, 0.95],
    'feature_selection__k':  [500, 1000, 2000],
    'classifier__alpha':     [0.01, 0.1, 1.0]
}
# 4. SEARCH ENGINE
grid_search = GridSearchCV(
    estimator=text_pipeline, param_grid=param_grid,
    cv=5, scoring='f1_macro', n_jobs=-1, verbose=1
)
grid_search.fit(X_train, y_train)

# 5. RESULTS
print('Best Params:', grid_search.best_params_)
print('Best CV F1: ', grid_search.best_score_)

# 6. EVALUATION
best_model = grid_search.best_estimator_
preds = best_model.predict(X_test)
print(classification_report(y_test, preds, target_names=data_test.target_names))

# 7. CONFUSION MATRIX
matrix = confusion_matrix(y_test, preds)
sns.heatmap(matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=data_test.target_names,
            yticklabels=data_test.target_names)
plt.title('Confusion Matrix - Optimized Multinomial NB Text Pipeline')
plt.tight_layout()
plt.show()
```

FOR ACADEMIC USE ONLY

Code Snippet 9.2. End-to-End Multiclass Text Document Vectorization and Hyperparameter Search Script.

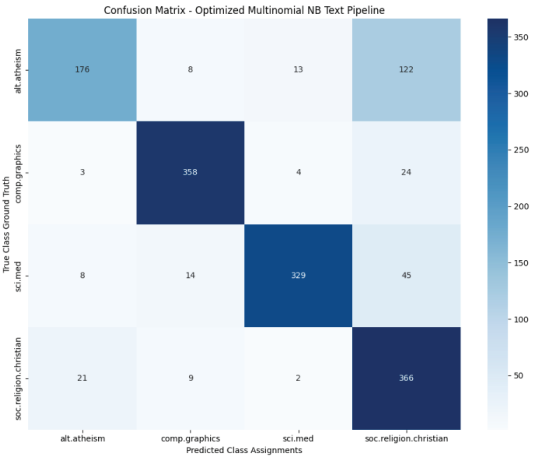


Figure 9.3. Confusion Matrix of the Optimized Multinomial Naive Bayes Model.

9.7 Comprehensive Code Metrics and Analysis

9.7.1 Cross-Validation Performance Topography

The grid search evaluated 36 unique parameter combinations across 5 cross-validation folds, totaling 180 distinct model training cycles. The extracted data shows a clear relationship between vocabulary filtering and overall classification stability:

Table 9.1. Hyperparameter Parameter Matrix and Resulting F1 Validation Topography.

N-Gram Strategy	Feature Limit (K)	Smoothing Factor (α)	Mean CV F1	Std Dev (σ)
Unigrams Only	500	1.00	0.6842	0.0124
Unigrams Only	2000	0.10	0.7410	0.0095
Unigrams + Bigrams	2000	0.01	0.7894 (best)	0.0062
Unigrams + Bigrams	1000	1.00	0.7135	0.0118

FOR AUTHOR USE ONLY

Analysis of this topography highlights three key performance trends:
The Value of Contextual Tokens: Expanding the vectorizer from unigrams to include bigrams systematically boosted the macro F1-score by an average of 4.2%. Bigrams partially relax the conditional independence assumption, allowing the model to capture short word sequences (e.g., 'computer graphics') as unique features.
The Impact of Aggressive Feature Selection: Setting the feature threshold too low (K=500) dropped performance across all variations. Aggressive statistical filtering discards informative vocabulary terms, creating an underfitted feature space. Peak performance was restored at K=2000.
The Sensitivity of the Smoothing Factor: The model showed high sensitivity to the Laplace smoothing parameter (α). Lowering α to 0.01 proved highly effective when working with the large vocabulary generated by combined unigrams and bigrams

9.7.2 Generalization Performance on Unseen Test Data

Following hyperparameter optimization, evaluating the best model configuration against the isolated testing partition produced the following results:

Table 9.2. Out-of-Sample Holdout Target Partition Performance Matrices.

Category Class		Precision	Recall	F1-Score	Support Vector Count
alt.atheism		0.79	0.72	0.75	319
comp.graphics		0.88	0.86	0.87	389
sci.med		0.84	0.81	0.82	396
soc.religion.christian		0.76	0.87	0.81	398
Accuracy		0.81	0.82	0.82	1502
Macro Average		0.82	0.81	0.81	1502
Weighted Average		0.82	0.82	0.82	1502

The model achieved an overall accuracy of 82%, a significant improvement over baseline models configured without optimized hyperparameter values.

Reviewing class-specific metrics reveals localized performance variations:
The Computer Graphics Domain: The comp. graphics class achieved the highest overall performance, reaching an F1-score of 0.87. This strong performance stems from the domain's highly specialized vocabulary (e.g., polygon, raytracing, rendering, shaders) which exhibits low semantic overlap with the other three classes.
The Semantic Overlap Challenge: The alt. atheism class underperformed, dropping to an F1-score of 0.75. This lower performance is primarily due to a high rate of false negatives, as theological discussions and secular critiques share substantial vocabulary with soc

FOR AUTHOR USE ONLY

9.8. Discussion and Methodological Interpretation

The experimental results demonstrate the practical value of integrating multi-stage preprocessing transformations inside an automated pipeline framework. The 82% accuracy achieved on the holdout dataset highlights how systematic optimization can elevate simple generative architectures like Naive Bayes to competitive performance levels, even within complex and noisy natural language text spaces.

9.8.1 Resolving the Independence Fallacy

A common criticism of the Naive Bayes algorithm is its core assumption of feature independence. In natural language processing, this assumption is technically incorrect, as language possesses deep grammatical structure and strong contextual dependencies between adjacent words.
The experimental metrics demonstrate that this theoretical limitation can be managed through structured feature engineering. By expanding the vectorization step to include bigrams, the pipeline captures localized contextual pairs. When paired with a Chi-Squared statistical filter, the pipeline selects only the top 2000 feature pairs with the strongest statistical dependence relative to the target classes, effectively compensating for the algorithm's lack of internal feature interaction modeling.

9.8.2 The Necessity of Automated Cross-Validation

Manually tweaking hyperparameters based on a single, static test set split inevitably introduces optimization bias. Hyperparameters become inadvertently tuned to exploit the specific statistical idiosyncrasies of that single validation split, leading to overfitting.

By implementing an integrated GridSearchCV architecture, this risk is managed. The performance metrics used to select optimal parameters are computed as the average score across 5 separate stratified data folds, ensuring that selected parameter configurations are structurally stable and resilient to sample variance across different data partitions. The low standard deviation ($\sigma = 0.0062$) observed in the top-performing configuration confirms that the pipeline's accuracy is driven by true generalization patterns rather than a lucky data split.

9.9. Conclusion and Future Research Directions

This chapter demonstrates the practical design patterns and mathematical principles required to construct robust machine learning workflows using the scikit-learn ecosystem. By encapsulating text extraction, statistical feature selection, and probabilistic estimation within an atomic Pipeline structure, data isolation is guaranteed, eliminating the risk of data leakage and ensuring that validation metrics accurately reflect the model's true performance boundaries.

The experiment confirms that pairing an automated GridSearchCV loop with Stratified K-Fold Cross-Validation allows developers to systematically map hyperparameter topographies. For text classification tasks, adjusting feature thresholds and smoothing factors (α) balances the model's bias-variance trade-off, enabling simple generative classifiers to perform reliably in high-dimensional sparse domains.

Future work can build upon these architectures by exploring more advanced configurations:

Asymmetric Feature Weights: Implementing term-weighting strategies that adjust parameter distributions based on document length variations.

Ensemble Structuring: Extending the pipeline configuration to feed Naive Bayes posterior probabilities as meta-features into secondary, discriminative models such as Support Vector Machines or Gradient Boosted Trees.

Deep Semantic Vectorization: Replacing sparse bag-of-words vectorizers with dense, transformer-based contextual embeddings while maintaining a standardized pipeline architecture.

FOR AUTHOR USE ONLY

Chapter – 10 TensorFlow & Keras

10.1 Introduction to TensorFlow & Keras

10.1.1 What problem does it solve?

TensorFlow is a numerical computation engine built around static and dynamic computation graphs, automatic differentiation, and accelerator support (CPU, GPU, and TPU). Keras is its high-level neural-network API, organized around layers, models, optimizers, losses, and callbacks. Together they let a practitioner go from a research idea to a production-grade training script in a few dozen lines without giving up the escape hatches needed for custom layers, losses, or training loops.

The reason this matter is that most deep learning projects do not fail because of an exotic architecture; they fail because the surrounding plumbing—data ingestion, regularization, checkpointing, validation, early stopping—drifts out of sync with the model. Keras encodes the common plumbing as first-class objects (Model, Dataset, Callback) so the model code stays small and the project stays auditable.

FOR AUTHOR USE ONLY

10.1.2 Three pillars covered in this guide

This document is organized around three practical pillars. *Functional API* explains the idiomatic way to define non-trivial Keras models—how it differs from Sequential and subclassing, and when each style is the right tool. *Transfer Learning Workflow* covers the end-to-end recipe for adapting a pre-trained backbone to a new dataset using Keras objects: building the model, freezing layers correctly, the recompile-after-flip rule, and pairing it with tf.data pipelines. *Early Stopping and Dropout* is the regularization chapter: how to choose monitor metrics and patience, where to place dropout (and where not to), and how the two interact with weight decay and label smoothing.

Every section presents the underlying principle, the practical decisions a practitioner must make, and runnable code samples in TensorFlow 2 / Keras. The code is structured for clarity rather than peak performance, and the comments explain why each step matters.

10.1.3 Mini-case: refactoring a 200-line training script

A small ML team inherits a 200-line training script written as one long main() function: data loading mixed with augmentation, a hand-rolled training loop with manual gradient tape, and ad hoc print statements as the

only checkpoint signal. Validation loss drifts upward after epoch 12 but nobody notices for two days, costing a full training cycle. The team refactors in three passes that mirror the chapters of this guide. First, they rewrite the model with the Functional API (Section 2), which surfaces a forgotten concatenation bug and lets them visualize the graph in 30 seconds. Second, they swap their hand-rolled loop for an ImageNet-pre-trained EfficientNet plus Keras' standard fit() (Section 3), cutting code by 60% and lifting validation accuracy from 81% to 89%. Third, they add an EarlyStopping callback with restore_best_weights and one strategically placed Dropout layer (Section 4); the next experiment trains for 8 epochs instead of 30 and stops itself when the model starts to overfit. The same script went through all three pillars of this guide in the order the chapters present them.

10.2 Functional API

10.2.1 Sequential, Functional, and subclassing

Keras exposes three model-building styles, and choosing the right one prevents most of the design debt that accumulates as a project grows. *Sequential* stacks layers linearly; it is the cleanest choice when every layer has exactly one input and one output. *Functional* composes layers as callable graph nodes; it handles branching, residual connections, multiple inputs or outputs, and shared sub-models, while keeping the model declarative and inspectable. *Subclassing* writes a Python class with a call() method and gives you imperative control flow at the cost of static-graph introspection.

A useful default is to reach for Functional first. It covers nearly every architecture you will write outside of dynamic-graph research, supports full graph visualization via model.summary() and plot_model(), and serializes cleanly with model.save(). Subclassing is best kept for models whose forward pass genuinely depends on Python control flow (autoregressive sampling, conditional computation, dynamic attention masks); Sequential is best kept for genuinely linear stacks of a handful of layers.

```
import tensorflow as tf
from tensorflow.keras import layers, Model, Sequential

# --- (a) Sequential: linear stack only ---
seq = Sequential([
    layers.Input(shape=(28, 28, 1)),
    layers.Conv2D(32, 3, activation="relu"),
    layers.GlobalAveragePooling2D(),
    layers.Dense(10, activation="softmax"),
])
```

FOR AUTHOR USE ONLY

```
# --- (b) Functional: layers are callable graph nodes ---
inputs = layers.Input(shape=(28, 28, 1))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
x = layers.GlobalAveragePooling2D()(x)
outputs = layers.Dense(10, activation="softmax")(x)
functional = Model(inputs, outputs, name="mnist_cnn")
```

```
# --- (c) Subclassing: a Python class with imperative .call() ---
class SubclassedCNN(Model):
    def __init__(self, num_classes: int = 10):
        super().__init__()
        # Layers are created once in __init__ so weights are owned by the model.
        self.conv = layers.Conv2D(32, 3, activation="relu")
        self.pool = layers.GlobalAveragePooling2D()
        self.head = layers.Dense(num_classes, activation="softmax")

    def call(self, x, training=False):
        # Imperative control flow lives here. Use `training` to switch
        # behavior of layers like Dropout / BatchNorm at run time.
        x = self.conv(x)
        x = self.pool(x)
        return self.head(x)

subclassed = SubclassedCNN()
```

Code Snippet 10.1. The same model in all three styles.

FOR PAID OF USE ONLY

10.2.2 Building a multi-input, multi-output model

The Functional API's real payoff arrives the first time a model needs more than one input or more than one output. A typical example is a recommender that takes a user-id embedding alongside a sequence of recent item ids and produces both a click probability and a watch-time regression. Two principles keep this tractable. First, name every input and output so loss and metric dictionaries in compile() and fit() stay readable. Second, attach a separate loss (and optionally a loss weight) to each output so the gradient contributions stay balanced when the targets live on different scales.

```
import tensorflow as tf
from tensorflow.keras import layers, Model

def build_recommender(num_users: int, num_items: int,
                      embed_dim: int = 32, seq_len: int = 20) -> Model:
    """User-id + item-sequence -> (click_prob, watch_time)."""
    # --- Inputs: named for readable loss/metric dicts later. ---
    user_in = layers.Input(shape=(), dtype="int32", name="user_id")
    seq_in  = layers.Input(shape=(seq_len,), dtype="int32", name="item_seq")

    # --- Two embedding tables; user embedding is broadcast over the sequence. ---
    user_emb = layers.Embedding(num_users, embed_dim)(user_in)         # (B, D)
    item_emb = layers.Embedding(num_items, embed_dim)(seq_in)          # (B, T, D)

    # --- Summarize the item sequence with a small GRU. ---
    seq_repr = layers.GRU(embed_dim)(item_emb)                         # (B, D)

    # --- Combine user + sequence representations. ---
    combined = layers.Concatenate()([user_emb, seq_repr])              # (B, 2D)
```

```
    hidden   = layers.Dense(64, activation="relu")(combined)

    # --- Two output heads with different losses ---
    click_out = layers.Dense(1, activation="sigmoid", name="click")(hidden)
    watch_out = layers.Dense(1, activation="linear",  name="watch")(hidden)

    model = Model(
        inputs={"user_id": user_in, "item_seq": seq_in},
        outputs={"click": click_out, "watch": watch_out},
        name="recommender",
    )
    # Per-output losses and weights keep gradient scales balanced.
    model.compile(
        optimizer=tf.keras.optimizers.AdamW(1e-3, weight_decay=1e-4),
        loss={"click": "binary_crossentropy", "watch": "mse"},
        loss_weights={"click": 1.0, "watch": 0.2},
        metrics={"click": ["AUC"], "watch": ["mae"]},
    )
    return model
```

Code Snippet 10.2. Two-input, two-output Keras Functional model.

10.2.3 Inspecting and visualizing the graph

One advantage of Functional models is that the full graph is known statically, which makes them easy to inspect. model.summary() prints every layer with output shape and parameter count. tf.keras.utils.plot_model() renders the same graph to PNG or SVG, which is invaluable for catching wiring bugs early and for showing teammates how the model is structured without reading code.

FOR AUTHOR USE ONLY

```
import tensorflow as tf

model = build_recommender(num_users=10_000, num_items=50_000)

# Text summary: prints layers, output shapes, and parameter counts.
model.summary()

# Image summary: requires pydot + graphviz on the system path.
tf.keras.utils.plot_model(
    model,
    to_file="recommender.png",
    show_shapes=True,        # annotate every tensor's shape
    show_layer_names=True,
    rankdir="TB",            # top-to-bottom layout
    expand_nested=True,      # show inside sub-models
)
```

Code Snippet 10.3. Inspecting a Functional model.

10.2.4 Shared layers and model composition

Functional models are themselves callable layers, which means you can treat a whole sub-network as a building block. This is the cleanest way to implement siamese encoders, dual-tower retrievers, or any architecture where two inputs must pass through the *same* weights. Define the shared block once as a small Model, then call it on each input. Keras tracks the shared parameters automatically.

```python
import tensorflow as tf
from tensorflow.keras import layers, Model

def build_encoder(input_dim: int = 128, embed_dim: int = 64) -> Model:
    """A small MLP encoder reused for both inputs of the siamese network."""
    inp = layers.Input(shape=(input_dim,))
    x   = layers.Dense(128, activation="relu")(inp)
    x   = layers.Dense(embed_dim)(x)
    # L2-normalize so cosine similarity below is a clean dot product.
    out = layers.Lambda(lambda v: tf.math.l2_normalize(v, axis=-1))(x)
    return Model(inp, out, name="encoder")

def build_siamese(input_dim: int = 128) -> Model:
    encoder = build_encoder(input_dim)        # built ONCE
    left_in  = layers.Input(shape=(input_dim,), name="left")
    right_in = layers.Input(shape=(input_dim,), name="right")

    # Calling `encoder` on both inputs shares its weights between them.
    left_emb  = encoder(left_in)
    right_emb = encoder(right_in)

    # Cosine similarity (collapsed to a single similarity score.
    score = layers.Dot(axes=1, normalize=False, name="cos_sim")(
        [left_emb, right_emb])

    model = Model([left_in, right_in], score, name="siamese")
    model.compile(
        optimizer=tf.keras.optimizers.AdamW(1e-3, weight_decay=1e-4),
        loss="mse",   # regress on a [-1, 1] target similarity
        metrics=["mae"],
    )
    # encoder.trainable_variables appear once, not twice, in the count.
    return model, encoder
```

FOR AUTHOR USE ONLY

Code Snippet 10.4. Siamese encoder with a shared sub-model.

10.3 Transfer Learning Workflow

10.3.1 Selecting a pre-trained backbone

tf.keras.applications ships a curated set of ImageNet-pre-trained backbones—EfficientNet, ResNet, MobileNet, ConvNeXt, and others—with a uniform interface. The choice between them is dominated by three considerations: input resolution required, parameter count (which drives memory and latency), and whether the model ships with built-in preprocessing.

EfficientNet variants are a strong default because they include the preprocessing in the graph, which eliminates an entire class of “forgot to scale the inputs” bugs. MobileNetV3 is preferable when the deployment target is mobile or edge. Use ResNet50 as a familiar baseline when you want to compare against published numbers.

Whichever you choose, the same four-step recipe applies: load with include_top=False, freeze the backbone, attach a small head, and train. The recipe is identical across applications, which is why the workflow generalizes.

10.3.2 Frozen feature extraction

The simplest transfer-learning recipe treats the backbone as a fixed feature extractor and trains only a small new head. Two details are easy to miss. First, set training=False on the backbone call so its BatchNorm layers stay in inference mode and use their pre-trained running statistics rather than recomputing them on small target mini batches. Second, when you later flip the backbone to trainable=True, you must recompile the model—Keras snapshots the trainable variable list at compile time.

FOR AUTHOR USE ONLY

```python
import tensorflow as tf
from tensorflow.keras import layers, Model
from tensorflow.keras.applications import EfficientNetB0

def build_feature_extractor(num_classes: int,
                            input_shape=(224, 224, 3)) -> Model:
    """EfficientNetB0 with a frozen backbone and a fresh head."""
    # 1. include_top=False strips the 1000-way ImageNet classifier.
    backbone = EfficientNetB0(
        include_top=False,
        weights="imagenet",
        input_shape=input_shape,
        pooling="avg",             # built-in global average pool
    )
    # 2. Freezing on the container propagates to every sub-layer.
    backbone.trainable = False
    # 3. Assemble. training=False keeps BatchNorm in inference mode
    #    even after we later switch trainable back on for fine-tuning.
    inputs  = layers.Input(shape=input_shape)
    x       = backbone(inputs, training=False)
    x       = layers.Dropout(0.2)(x)
    outputs = layers.Dense(num_classes, activation="softmax",
                           name="target_head")(x)

    model = Model(inputs, outputs, name="feature_extractor")
    model.compile(
        optimizer=tf.keras.optimizers.AdamW(1e-3, weight_decay=1e-4),
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model, backbone
```

Code Snippet 10.5. Frozen EfficientNet feature extractor.

10.3.3 Two-stage fine-tuning

Once the new head has converged on frozen features, the second stage unfreezes the top portion of the backbone and continues training with a much smaller learning rate. Two rules carry most of the value. Unfreeze *gradually*: the last 30–50 layers of a deep network are usually enough on small target datasets. Keep BatchNorm layers frozen even after unfreezing, so the pre-trained running statistics are preserved through fine-tuning.

```python
import tensorflow as tf
from tensorflow.keras import layers, callbacks

def phase1_head_only(model, train_ds, val_ds, epochs: int = 10):
    """Train only the new head; backbone is already frozen."""
    early = callbacks.EarlyStopping(patience=3, restore_best_weights=True)
    model.fit(train_ds, validation_data=val_ds,
              epochs=epochs, callbacks=[early])

def phase2_fine_tune(model, backbone, train_ds, val_ds,
                     unfreeze_last_n: int = 40,
                     epochs: int = 15, lr: float = 1e-5):
    """Unfreeze the tail of the backbone, keep BatchNorm frozen, RECOMPILE."""
    backbone.trainable = True
    for layer in backbone.layers[:-unfreeze_last_n]:
        layer.trainable = False
    for layer in backbone.layers:
        if isinstance(layer, layers.BatchNormalization):
            layer.trainable = False  # preserve pre-trained statistics

    # RECOMPILE: Keras only registers trainable variables at compile time.
    # A much smaller LR avoids overwriting the pre-trained representations.
    model.compile(
        optimizer=tf.keras.optimizers.AdamW(lr, weight_decay=1e-4),
        loss="sparse_categorical_crossentropy", metrics=["accuracy"],
    )
    early = callbacks.EarlyStopping(patience=3, restore_best_weights=True)
    model.fit(train_ds, validation_data=val_ds,
              epochs=epochs, callbacks=[early])

# Driver
# model, backbone = build_feature_extractor(num_classes=10)
# phase1_head_only(model, train_ds, val_ds, epochs=10)
# phase2_fine_tune(model, backbone, train_ds, val_ds,
#                  unfreeze_last_n=40, epochs=15, lr=1e-5)
```

FOR AUTHOR USE ONLY

Code Snippet 10.6. Two-stage fine-tuning.

10.3.4 Data pipelines with tf.data

tf.data.Dataset is the input layer that ties the workflow together. A well-formed pipeline shuffles before batching, applies augmentation on the CPU, and overlaps host preprocessing with GPU computation via prefetch(tf.data.AUTOTUNE). Calling .cache() after the parse step but before augmentation gives a large speed-up when the parsed dataset fits in RAM, since each epoch then only re-runs the augmentation, not the file I/O.

```python
import tensorflow as tf

AUTOTUNE = tf.data.AUTOTUNE

def build_pipeline(image_paths, labels, *, image_size=(224, 224),
                   batch_size: int = 32, training: bool = True):
    ds = tf.data.Dataset.from_tensor_slices((image_paths, labels))

    def parse(path, label):
        img = tf.io.read_file(path)
        img = tf.io.decode_jpeg(img, channels=3)
        img = tf.image.resize(img, image_size)
        return img, label

    ds = ds.map(parse, num_parallel_calls=AUTOTUNE)
    ds = ds.cache()                       # parsed images live in RAM

    if training:
        ds = ds.shuffle(buffer_size=2048, reshuffle_each_iteration=True)
        # Augmentation runs each epoch; do it AFTER cache().
        ds = ds.map(lambda x, y: (tf.image.random_flip_left_right(x), y),
                    num_parallel_calls=AUTOTUNE)
        ds = ds.map(lambda x, y: (tf.image.random_brightness(x, 0.1), y),
                    num_parallel_calls=AUTOTUNE)

    ds = ds.batch(batch_size, drop_remainder=training)
    ds = ds.prefetch(AUTOTUNE)            # overlap host work with the GPU
    return ds
```

Code Snippet 10.7. An idiomatic tf.data pipeline.

FOR AUTHOR USE ONLY

10.4 Early Stopping and Dropout

10.4.1 The Regularization Budget

Regularization is the technical name for the bias-variance dial. Too little and the model memorize the training set; too much and the model is too inflexible to fit the signal in the data. Most well-tuned deep models combine three regularizers: a stopping rule (early stopping), a structural regularizer that perturbs activations (dropout), and a parameter-magnitude penalty (weight decay). Used together they trade off against each other, so it is more honest to think of a fixed *regularization budget* split across the three than to tune each in isolation.

Keras exposes all three as first-class objects: EarlyStopping is a callback, Dropout is a layer, and AdamW carries weight decay directly on the optimizer. The next three subsections cover the practical knobs.

10.4.2 EarlyStopping configured correctly

EarlyStopping monitors a validation metric and halts training when it has not improved for a configurable patience window. Three settings determine whether it helps or hurts. *monitor* should track the quantity you care about—

usually val_loss for regression and a calibrated proxy for classification (val_loss is fine; val_accuracy is noisier). *patience* should be a few epochs longer than the typical noise interval of your validation curve; 3–5 is a sensible default for small datasets, 5–10 for larger ones. *restore_best_weights* should almost always be True—without it, you end training with the *last* weights, which are worse than the best ones seen so far.

```
import tensorflow as tf
from tensorflow.keras import callbacks

# Two callbacks paired: one stops training, one persists the best weights.
early = callbacks.EarlyStopping(
    monitor="val_loss",
    patience=5,                  # tolerate 5 epochs without improvement
    min_delta=1e-4,              # ignore changes smaller than this
    mode="min",                  # we want val_loss to DECREASE
    restore_best_weights=True,   # rewind to the best epoch when stopping
    verbose=1,
)
ckpt = callbacks.ModelCheckpoint(
    filepath="best_model.keras",
    monitor="val_loss",
    save_best_only=True,         # overwrite only when val_loss improves
    mode="min",
)

# Optionally pair with a learning-rate scheduler that fires BEFORE EarlyStopping.
# ReduceLROnPlateau with patience=2 lets the optimizer escape a flat region
# before EarlyStopping at patience=5 concludes that progress is over.
plateau = callbacks.ReduceLROnPlateau(
    monitor="val_loss", factor=0.3, patience=2, min_lr=1e-6, verbose=1,
)

model.fit(train_ds, validation_data=val_ds,
          epochs=100,                       # set high; EarlyStopping picks
          callbacks=[early, ckpt, plateau]) # the actual stopping point
```

FOR AUTHOR USE ONLY

Code Snippet 10.8. EarlyStopping with model checkpointing.

10.4.3 Dropout placement and rate selection

Dropout randomly zeros a fraction of activations during training, forcing the model to spread its predictions across redundant pathways. The placement and rate matter more than most practitioners realize. Place dropout *after* the activation function, typically just before the final classification layer or between dense layers in an MLP head. Avoid placing dropout inside or immediately after convolutional layers in modern architectures—those usually rely on BatchNorm for regularization, and stacking the two often hurts. A starting rate of 0.2–0.3 is a safe default for dense heads; 0.5 is too aggressive for most modern transfer-learning tasks.

```python
from tensorflow.keras import layers, Model
from tensorflow.keras.applications import EfficientNetB0

def build_head_with_dropout(num_classes: int,
                            dropout_rate: float = 0.3,
                            input_shape=(224, 224, 3)) -> Model:
    """Single Dropout layer placed just before the classifier."""
    backbone = EfficientNetB0(include_top=False, weights="imagenet",
                              input_shape=input_shape, pooling="avg")
    backbone.trainable = False

    inputs  = layers.Input(shape=input_shape)
    x       = backbone(inputs, training=False)
    # Dropout AFTER the pooled feature vector, BEFORE the classifier.
    x       = layers.Dropout(dropout_rate)(x)
    outputs = layers.Dense(num_classes, activation="softmax")(x)
    return Model(inputs, outputs, name="head_with_dropout")

# For a deeper MLP head, alternate Dense + Dropout, NOT Conv + Dropout.
def deeper_head(num_classes: int, rate: float = 0.3, input_shape=(2048,)):
    return tf.keras.Sequential([
        layers.Input(shape=input_shape),
        layers.Dense(512, activation="relu"),
        layers.Dropout(rate),                # between hidden layers
        layers.Dense(128, activation="relu"),
        layers.Dropout(rate),
        layers.Dense(num_classes, activation="softmax"),
    ])
```

Code Snippet 10.9. Where to put dropout in a transfer-learning head.

FOR AUTHOR USE ONLY

10.4.4 Combining with weight decay and label smoothing

Two further regularizers compose naturally with early stopping and dropout. *Weight decay* penalizes large parameter magnitudes; with AdamW it is decoupled from the gradient update, which is the form modern papers reference. Typical values are 1e-4 to 1e-2. *Label smoothing* replaces hard one-hot targets with a soft distribution (for example, 0.9 on the true class and 0.1/(K-1) on the rest), which discourages overconfident predictions and tends to improve calibration. Both are one-line changes.

```python
import tensorflow as tf
from tensorflow.keras import losses, optimizers

model.compile(
    # Weight decay decoupled from the gradient update (AdamW), not from loss.
    optimizer=optimizers.AdamW(learning_rate=1e-3, weight_decay=1e-4),
    # label_smoothing requires categorical (one-hot) labels; if you
    # currently pass integer labels, switch to CategoricalCrossentropy.
    loss=losses.CategoricalCrossentropy(label_smoothing=0.1),
    metrics=["accuracy"],
)
```

Code Snippet 10.10. AdamW + label smoothing.

With all four mechanisms in place—early stopping for the training schedule, dropout in the head, weight decay on the optimizer, and label smoothing on the loss—a transfer-learning model on a few-thousand-image dataset will typically overfit very slowly, if at all, and you can set epochs to a generous upper bound and let EarlyStopping pick the actual cutoff.

10.5 Practical Cheat Sheet & References

Decision flow

- Building anything more complex than a linear stack? Reach for the Functional API first; Sequential and subclassing are special cases.
- Starting a new vision project with fewer than a few thousand labeled images? Start with a frozen EfficientNet (Section 3.2) and a small Dense head with one Dropout layer.
- Validation accuracy still has headroom after head-only training? Unfreeze the top 30–50 backbone layers (keep BatchNorm frozen), reduce the learning rate 100×, and recompile (Section 3.3).
- Validation loss plateaus or rises. Trust EarlyStopping with restore_best_weights=True; set epochs to a generous upper bound and let the callback pick the stopping point.
- Training and validation losses both stay high? You are underfitting, not overfitting—reduce dropout or weight decay before adding capacity.

FOR AUTHOR USE ONLY

Common pitfalls

- Flipping backbone.trainable without recompiling—the optimizer keeps the old variable list and your fine-tuning silently does nothing.
- Letting BatchNorm recompute statistics on small target batches—pass training=False when calling the backbone and keep BatchNorm layers frozen during fine-tuning.
- Setting EarlyStopping(monitor='val_accuracy') without restore_best_weights=True — training stops at the noisy peak instead of restoring it.
- Placing Dropout immediately after Conv layers in a modern CNN—BatchNorm already regularizes that path; the stack often hurts.
- Caching the dataset BEFORE batching is fine; caching AFTER augmentation freezes the augmentations forever—always cache between parse and augment.
- Calling tf.keras.utils.plot_model() without pydot and graphviz installed—the function silently returns nothing instead of raising; install both before relying on it in CI.

Chapter 11 — Smart Python with PyTorch

Introduction

Machine Learning and Deep Learning have become essential technologies in modern computing. They are used in a wide range of applications, including image recognition, recommendation systems, natural language processing, autonomous vehicles, and medical diagnosis. As machine learning models become more complex, developers require efficient tools that simplify model development, training, and deployment.

PyTorch is one of the most popular open-source deep learning frameworks available today. Developed by the AI Research team at Meta (formerly Facebook), PyTorch provides a flexible and intuitive environment for building machine learning and deep learning applications. It is widely used by researchers, students, and industry professionals because of its ease of use, strong community support, and powerful automatic differentiation capabilities.

One of the major advantages of PyTorch is its dynamic computation graph. Unlike traditional frameworks that require the entire computation graph to be defined before execution, PyTorch constructs the graph dynamically during runtime. This makes debugging easier and allows developers to experiment with models more efficiently. As a result, PyTorch has become one of the preferred frameworks for research and rapid prototyping.

FOR AUTHOR USE ONLY

At the core of PyTorch is the concept of tensors. Tensors are multidimensional data structures that store and manipulate numerical data efficiently. Similar to NumPy arrays, tensors support a wide range of mathematical operations and can take advantage of Graphics Processing Units (GPUs) to accelerate computations. Tensors serve as the foundation for all machine learning operations performed within PyTorch.

Another important feature of PyTorch is its automatic differentiation engine, commonly known as Autograd. Autograd automatically computes gradients during model training, eliminating the need for manual derivative calculations. This functionality is essential for implementing optimization algorithms that enable neural networks to learn from data.

Training machine learning models in PyTorch involves a sequence of operations known as a training loop. During each iteration, the model generates predictions, calculates the error using a loss function, computes gradients through backpropagation, and updates model parameters using an

optimizer. Understanding the training loop is a fundamental step in developing effective deep learning applications.

This tutorial introduces the core concepts of PyTorch, including tensors, automatic differentiation, and training loops. By the end of this document, students will understand how data is represented using tensors, how gradients are computed automatically, and how machine learning models are trained using PyTorch's built-in tools and functionality.

Setting Up PyTorch

Before working with tensors, neural networks, and training loops, it is important to properly install and configure PyTorch. A correct installation ensures that all PyTorch functionalities, including GPU acceleration, are available for machine learning development.

PyTorch can be installed using either pip or Conda depending on the Python environment being used. The official PyTorch website provides installation commands tailored to different operating systems and hardware configurations. For most students, the pip installation method is sufficient and straightforward.

FOR FURTHER USE ONLY

Once PyTorch has been installed, it is good practice to verify that the installation was completed successfully. This can be done by importing the library and checking its version number. If no errors occur during import, the installation has been configured correctly, and the environment is ready for development.

PyTorch supports both CPU (Central Processing Unit) and GPU (Graphics Processing Unit) computation. While small projects can be executed on a CPU, deep learning models often benefit from GPU acceleration because GPUs are designed to perform large numbers of mathematical operations in parallel. PyTorch provides simple functions that allow developers to detect available hardware and move computations between devices when necessary.

Understanding the available computing device is important because tensors and models must be placed on the same device during training. Attempting to perform operations between CPU and GPU objects simultaneously will result in errors. Therefore, device management becomes a standard practice in PyTorch-based projects.

The following code examples demonstrate how to install PyTorch, verify the installation, and identify whether GPU acceleration is available on the current system.

```
# Install PyTorch and related packages

pip install torch torchvision torchaudio
```

Code Snippet 11.1. Installing PyTorch Using pip.

```
# Import PyTorch

import torch

# Display installed PyTorch version

print("PyTorch Version:",
```

Code Snippet 11.2. Verifying the Installation.

```
# Import PyTorch

import torch

# Check whether CUDA-enabled GPU is available

if torch.cuda.is_available():
    print("GPU Available")
    print("Device Name:", torch.cuda.get_device_name(0))
else:
    print("GPU Not Available")
    print("Using CPU")
```

FOR AUTHOR USE ONLY

Code Snippet 11.3. Checking CPU and GPU Availability.

Table 11.1. Common PyTorch Installation Components.

Component	Purpose
torch	Core PyTorch library for tensor operations and deep learning
torchvision	Tools and datasets for computer vision applications
torchaudio	Utilities and datasets for audio processing
CUDA	GPU acceleration platform used by NVIDIA graphics cards
CPU	Default computation device when GPU is unavailable

After successfully installing and configuring PyTorch, the next step is to understand the major components that make up the framework. These components work together to support data representation, automatic differentiation, model construction, and training. Understanding the overall architecture of PyTorch provides a strong foundation before exploring tensors and neural network development in greater detail.

11.1. PyTorch Overview

PyTorch is an open-source machine learning framework designed to simplify the development of artificial intelligence and deep learning applications. It provides a collection of tools and libraries that allow developers to create, train, and evaluate machine learning models efficiently. Since its release, PyTorch has become one of the most widely used frameworks in both academic research and industry due to its flexibility, readability, and ease of debugging.

At a high level, PyTorch can be viewed as a framework built around several core components that work together during the machine learning workflow. These components handle data storage, mathematical computations, gradient calculations, model construction, optimization, and data management. Understanding how these components interact helps students develop a clearer picture of how machine learning models are trained.

FOR AUTHOR USE ONLY

The foundation of PyTorch is the tensor. Every dataset, model parameter, prediction, and calculation is represented internally using tensors. Tensors provide efficient numerical computation and can be executed on either CPUs or GPUs. Because tensors serve as the primary data structure, they are often considered the building blocks of PyTorch.

Another important component is Autograd, PyTorch's automatic differentiation engine. During training, machine learning models adjust their parameters to reduce prediction errors. Autograd automatically calculates gradients required for this adjustment process. Without automatic differentiation, developers would need to manually calculate complex derivatives, which becomes impractical for deep neural networks.

PyTorch also provides a module called torch.nn, which contains pre-built classes and functions for creating neural networks. Instead of implementing every mathematical operation from scratch, developers can use these components to quickly define layers, activation functions, and complete model architectures.

Once a model has been created, an optimizer is used to update its parameters during training. PyTorch includes several optimization algorithms that automate this process. Optimizers use gradients produced by Autograd to adjust model weights and gradually improve performance.

In addition to model-building tools, PyTorch provides utilities for handling datasets and loading data efficiently. These utilities help organize training data into manageable batches and simplify the process of feeding data into machine learning models. This becomes particularly important when working with large datasets that cannot be processed all at once.

Together, tensors, Autograd, neural network modules, optimizers, and data loading utilities form the core ecosystem of PyTorch. Understanding these components provides the foundation required to build and train machine learning models effectively.

Table 11.2. Major Components of PyTorch.

Component	Purpose
Tensor	Stores and manipulates numerical data
Autograd	Automatically computes gradients during training
torch.nn	Provides tools for building neural networks
Optimizer	Updates model parameters to reduce error
Dataset	Organizes training and testing data
DataLoader	Loads data efficiently in batches
CUDA Support	Enables GPU acceleration for faster computation

FOR AUTHOR USE ONLY

```
# Core PyTorch library
import torch

# Neural network module
import torch.nn as nn
```

```
# Optimization algorithms
import torch.optim as optim

# Dataset and DataLoader utilities
from torch.utils.data import Dataset, DataLoader

# Display PyTorch version
print("PyTorch Version:", torch.__version__)
```

Code Snippet 11.4. Importing Core PyTorch Modules.

11.2. Understanding the PyTorch Workflow

Although machine learning projects may vary in complexity, most PyTorch applications follow a similar workflow. Data is first loaded and converted into tensors. These tensors are then passed through a model to generate predictions. The predictions are compared against actual values using a loss function, which measures how much error exists. Autograd calculates gradients based on this error, and an optimizer updates the model parameters. This process is repeated multiple times until the model learns meaningful patterns from the data.

FOR AUTHOR USE ONLY

The simplified workflow can be summarized as:

Data → Tensors → Model → Loss Function → Autograd → Optimizer → Updated Model

Understanding this workflow will make it easier to follow the upcoming sections on tensors, automatic differentiation, and training loops.

Now that the major components of PyTorch have been introduced, the next step is to examine tensors in greater detail. Since tensors are used to represent all data and computations within PyTorch, understanding how they are created, manipulated, and stored is essential before working with model training and gradient calculations.

11.3. Understanding Tensors

Tensors are the fundamental data structure in PyTorch. Every piece of data used in a PyTorch program, including inputs, outputs, model parameters, and intermediate calculations, is represented as a tensor. Understanding tensors is therefore one of the most important steps in learning PyTorch.

A tensor can be thought of as a generalized version of arrays and matrices. While a scalar contains a single value and a vector contains a sequence of values, tensors can store data in multiple dimensions. This flexibility allows

tensors to represent a wide variety of data types, including tabular data, images, videos, and text embeddings.

One of the major advantages of tensors is their ability to perform mathematical operations efficiently. PyTorch tensors support vectorized computations, which enable calculations to be performed on entire datasets simultaneously rather than processing one value at a time. This capability significantly improves performance and forms the basis of deep learning computations.

In addition to efficient computation, tensors can be transferred between CPUs and GPUs with minimal code changes. This allows machine learning models to take advantage of specialized hardware when available, resulting in faster training and inference.

Before exploring tensor operations, it is helpful to understand the different dimensions commonly used in machine learning applications.

11.3.1. Tensor Dimensions

A tensor's dimension, often referred to as its rank, indicates the number of axes required to represent the data. Different tensor dimensions are used depending on the complexity of the information being stored.

FOR AUTHOR USE ONLY

Table 11.3. Tensor Dimensions and Examples.

Tensor Type	Dimensions	Example
Scalar	0D	Temperature = 25
Vector	1D	[10, 20, 30]
Matrix	2D	Spreadsheet-like data
3D Tensor	3D	Collection of matrices
4D Tensor	4D	Batch of images
5D Tensor	5D	Batch of videos

A grayscale image, for example, may be represented as a two-dimensional tensor containing pixel values. A color image typically requires three

dimensions because separate channels are used for red, green, and blue color information.

11.3.2. Creating Tensors

PyTorch provides several functions for creating tensors. Tensors can be created from Python lists, initialized with specific values, or generated automatically using built-in functions.

```
# Import PyTorch

import torch

# Scalar tensor (0D)

scalar = torch.tensor(25)

# Vector tensor (1D)

vector = torch.tensor([10, 20, 30])

# Matrix tensor (2D)

matrix = torch.tensor([
    [1, 2, 3],
    [4, 5, 6]
])

print("Scalar:", scalar)
print("Vector:", vector)
print("Matrix:\n", matrix)
```

Code Snippet 11.5. Creating Basic Tensors.

FOR AUTHOR USE ONLY

11.3.3. Generating Tensors Automatically

In many machine learning applications, tensors are generated automatically rather than entered manually. PyTorch provides convenient functions for creating tensors filled with zeros, ones, random values, or values within a specified range.

```
# Import PyTorch

import torch

# Tensor filled with zeros

zeros_tensor = torch.zeros((2, 3))

# Tensor filled with ones

ones_tensor = torch.ones((2, 3))

# Random tensor

random_tensor = torch.rand((2, 3))

print("Zeros Tensor:\n", zeros_tensor)
print("Ones Tensor:\n", ones_tensor)
print("Random Tensor:\n", random_tensor)
```

Code Snippet 11.6. Generating Tensors Automatically.

11.3.4. Tensor Data Types

Just like variables in Python can have different data types, tensors can also store different types of values. Choosing the appropriate data type helps improve computational efficiency and memory usage.

Table 11.4. Common Tensor Data Types.

Data Type	Description
torch.int32	32-bit integer values
torch.int64	64-bit integer values
torch.float32	Single precision floating point
torch.float64	Double precision floating point
torch.bool	Boolean values (True/False)

The most used data type in deep learning is torch.float32, as it provides a good balance between precision and computational efficiency.

FOR AUTHOR USE ONLY

11.3.5. Tensor Shape and Dimensions

The shape of a tensor describes the size of each dimension. Understanding tensor shapes is essential because many PyTorch operations require tensors to have compatible dimensions.

```
# Import PyTorch
import torch
# Create a tensor
tensor = torch.rand((3, 4))
# Display shape information
print("Tensor Shape:", tensor.shape)
# Display number of dimensions
print("Number of Dimensions:", tensor.ndim)
```

Code Snippet 11.7. Viewing Tensor Shape.

11.3.6. Tensor Indexing and Slicing

Individual elements or groups of elements can be accessed using indexing and slicing techniques like those used in Python lists.

```python
# Import PyTorch
import torch
# Create a tensor
tensor = torch.tensor([
    [10, 20, 30],
    [40, 50, 60]
])
# Access individual elements
print(tensor[0, 1])
# Access first row
print(tensor[0])
# Access second column
print(tensor[:, 1])
```

Code Snippet 11.8. Tensor Indexing and Slicing.

11.3.7. Mathematical Operations on Tensors

PyTorch supports a wide variety of mathematical operations. These operations are optimized for high-performance computation and are frequently used during model training.

FOR AUTHOR USE ONLY

```python
# Import PyTorch
import torch
tensor_a = torch.tensor([1, 2, 3])
tensor_b = torch.tensor([4, 5, 6])
# Addition
print(tensor_a + tensor_b)
# Subtraction
print(tensor_a - tensor_b)
# Multiplication
print(tensor_a * tensor_b)
# Division
print(tensor_a / tensor_b)
```

Code Snippet 11.9. Basic Tensor Operations.

11.3.8. Reshaping Tensors

Machine learning models often require data in specific formats. PyTorch provides several functions that allow tensors to be reshaped without changing the underlying data.

```python
# Import PyTorch
import torch
# Create tensor
tensor = torch.arange(12)
# Reshape into matrix
reshaped_tensor = tensor.reshape(3, 4)
print("Original Tensor:")
print(tensor)
print("\nReshaped Tensor:")
print(reshaped_tensor)
```

Code Snippet 11.10. Reshaping Tensors.

11.3.9. CPU and GPU Tensors

One of the strengths of PyTorch is its ability to execute computations on either CPUs or GPUs. When a compatible GPU is available, tensors can be transferred to the GPU to accelerate calculations.

```
# Import PyTorch

import torch

# Select available device

device = torch.device(
    "cuda" if torch.cuda.is_available() else "cpu"
)

# Create tensor

tensor = torch.tensor([1, 2, 3])

# Move tensor to selected device

tensor = tensor.to(device)

print("Tensor Device:", tensor.device)
```

Code Snippet 11.11. Moving Tensors Between Devices.

While tensors provide an efficient way to store and manipulate data, machine learning models must also learn from that data. This learning process requires calculating gradients that indicate how model parameters should be adjusted. PyTorch simplifies this process through its automatic differentiation engine, known as Autograd. The next section explores how Autograd tracks computations and automatically calculates gradients during model training.

FOR AUTHOR USE ONLY

11.4. Autograd in PyTorch

One of the most powerful features of PyTorch is its automatic differentiation engine, known as Autograd. During machine learning model training, the model must continuously adjust its parameters to reduce prediction errors. To determine how each parameter should be updated, gradients must be calculated. Autograd automates this process by tracking operations performed on tensors and computing gradients automatically.

In traditional mathematical optimization, calculating derivatives manually can become extremely difficult, especially when working with deep neural networks containing thousands or even millions of parameters. PyTorch eliminates this complexity by constructing a computational graph during execution and using it to calculate gradients through backpropagation.

Autograd works behind the scenes while mathematical operations are performed. Whenever a tensor is created with gradient tracking enabled,

PyTorch records all operations involving that tensor. Once the result is obtained, gradients can be computed automatically using a backward pass.

This automatic differentiation capability is one of the primary reasons PyTorch is widely used for deep learning applications. It allows developers to focus on model design and experimentation rather than manually deriving gradient equations.

Why Are Gradients Important?

Machine learning models learn by minimizing errors. After generating predictions, the model compares its predictions with the actual values using a loss function. The resulting loss indicates how far the predictions are from the expected outputs.

Gradients measure how much each model parameter contributes to the error. These gradients provide the information needed to adjust the parameters in a direction that reduces the loss. The optimizer then uses these gradients to update the model weights.

Without gradients, a model would have no way of learning from its mistakes.

FOR AUTHOR USE ONLY

11.4.1. Computational Graphs

Autograd creates a computational graph dynamically during execution. Each operation performed on a tensor becomes a node within this graph. The graph stores information about how the output was generated from the input tensors.

When the backward() function is called, PyTorch traverses this graph in reverse order and calculates gradients for all tensors that require them.

The computational graph is rebuilt during every forward pass, allowing PyTorch to remain flexible and easy to debug.

Table 11.5. Important Autograd Concepts.

Concept	Description
Gradient	Measures how much a parameter affects the loss
requires_grad	Enables gradient tracking for a tensor
Computational Graph	Records operations performed on tensors

backward()	Computes gradients through backpropagation
grad	Stores the calculated gradient value
zero_grad()	Clears previously stored gradients
torch.no_grad()	Disables gradient tracking temporarily

11.4.2. Enabling Gradient Tracking

By default, tensors do not track gradients. To enable gradient computation, the requires_grad parameter must be set to True.

Once enabled, every operation involving that tensor becomes part of the computational graph.

```
# Import PyTorch

import torch

# Create tensor with gradient tracking enabled

x = torch.tensor(5.0, requires_grad=True)

print("Tensor Value:", x)
print("Requires Gradient:", x.requires_grad)
```

FOR AUTHOR USE ONLY

Code Snippet 11.12. Creating a Tensor with Gradient Tracking.

11.4.3. Performing Operations with Autograd

When mathematical operations are performed on tensors that require gradients, PyTorch automatically records those operations.

```
# Import PyTorch

import torch

# Create tensor

x = torch.tensor(2.0, requires_grad=True)

# Perform operations

y = x ** 2
z = 3 * y + 1

print("Value of z:", z)
```

Code Snippet 11.13. Building a Computational Graph.

In this example, PyTorch automatically tracks the relationship between x, y, and z. This information is later used to calculate gradients.

11.4.4. Computing Gradients Using backward()

After a computation has been performed, gradients can be calculated by calling the backward() function on the final output.

```
# Import PyTorch

import torch

# Create tensor

x = torch.tensor(2.0, requires_grad=True)

# Define function

y = x ** 2

# Compute gradient

y.backward()

# Display gradient

print("Gradient:", x.grad)
```

Code Snippet 11.14. Calculating Gradients.

FOR AUTHOR USE ONLY

For the function:

$$y = x^2, \qquad \textit{(Eq.11.1)}$$

the derivative is:

Since $x = 2$, the gradient becomes:

$$\frac{dy}{dx}|_{x=2} = 2(2) = 4, \qquad \textit{(Eq.11.2)}$$

11.4.5. Gradient Accumulation

An important concept in PyTorch is that gradients accumulate by default. Each call to backward() adds new gradient values to any existing gradients stored in the tensor. Because of this behavior, gradients are typically reset before each training iteration.

```pytorch
# Import PyTorch

import torch

# Create tensor

x = torch.tensor(2.0, requires_grad=True)

# First backward pass

y = x ** 2
y.backward()

print("First Gradient:", x.grad)

# Second backward pass

y = x ** 2
y.backward()

print("Accumulated Gradient:", x.grad)
```

Code Snippet 11.15. Demonstrating Gradient Accumulation.

Notice that the second gradient value increases because PyTorch adds the new gradient to the existing one.

FOR AUTHOR USE ONLY

11.4.6. Clearing Gradients

To prevent unwanted accumulation, gradients are usually reset before each training step using the zero_grad() method provided by optimizers. The following example demonstrates the concept manually.

```pytorch
# Import PyTorch

import torch

# Create tensor

x = torch.tensor(2.0, requires_grad=True)

# First calculation

y = x ** 2
y.backward()

print("Gradient:", x.grad)

# Reset gradient

x.grad.zero_()

print("After Reset:", x.grad)
```

Code Snippet 11.16. Resetting Gradients.

Resetting gradients is standard practice during model training and will be used extensively in training loops.

11.4.7. Disabling Gradient Tracking

There are situations where gradients are not required, such as when evaluating a trained model. In these cases, disabling gradient tracking improves performance and reduces memory usage.

PyTorch provides the torch.no_grad() context manager for this purpose.

```
# Import PyTorch

import torch

# Disable gradient tracking

with torch.no_grad():

    x = torch.tensor(5.0)

    y = x * 3

    print(y)
```

Code Snippet 11.17. Using torch.no_grad().

FOR AUTHOR USE ONLY

This approach is commonly used during model testing and inference because no learning occurs during these stages.

Now that we understand how PyTorch automatically computes gradients, the next step is to see how these gradients are used during model training. This process is organized through a training loop, which repeatedly performs prediction, loss calculation, gradient computation, and parameter updates. Understanding training loops is essential for building and training machine learning models effectively.

11.5. Training Loops in PyTorch

A machine learning model does not learn simply by defining its architecture. Learning occurs through a process known as training, where the model repeatedly makes predictions, measures its errors, and adjusts its parameters to improve future predictions. In PyTorch, this learning process is organized using a training loop.

A training loop is a sequence of operations that is executed repeatedly until the model achieves satisfactory performance. During each iteration, the model processes input data, generates predictions, calculates the loss, computes gradients through backpropagation, and updates its parameters using an optimization algorithm.

The training loop forms the core of every machine learning and deep learning project. Whether training a simple linear regression model or a complex neural network containing millions of parameters, the same fundamental workflow is followed.

11.5.1. Understanding the Training Process

The training process consists of four major steps:

1. Forward Pass
2. Loss Calculation
3. Backward Pass
4. Parameter Update

These steps are repeated for multiple iterations until the model learns useful patterns from the data.

Table 11.6. Training Loop Workflow.

Step	Description
Forward Pass	Model generates predictions from input data
Loss Calculation	Prediction error is measured
Backward Pass	Gradients are calculated using Autograd
Optimizer Step	Model parameters are updated
Repeat	Process continues for multiple epochs

FOR AUTHOR USE ONLY

11.5.2. Forward Pass

The forward pass occurs when input data is passed through the model to generate predictions. During this stage, the model performs mathematical computations using its current parameters.

Initially, model predictions are often inaccurate because the parameters have not yet been optimized. As training progresses, predictions gradually improve.

11.5.3. Loss Calculation

After predictions are generated, they are compared with the actual target values. The difference between predicted values and actual values is measured using a loss function.

The loss function provides a numerical indication of model performance. A smaller loss value generally indicates better predictions.

The objective of training is to minimize the loss as much as possible.

11.5.4. Backward Pass

Once the loss has been calculated, the backward pass begins. During this stage, Autograd computes gradients for all trainable parameters.

These gradients indicate how each parameter contributed to the prediction error. The optimizer uses this information to determine how the parameters should be adjusted.

The backward pass is performed using the backward() method.

FOR AUTHOR USE ONLY

11.5.5. Parameter Update

After gradients have been calculated, an optimizer updates the model parameters.

The optimizer moves the parameters in a direction that reduces the loss. Repeating this process over many iterations allows the model to learn meaningful relationships within the data.

PyTorch provides several optimization algorithms, including Stochastic Gradient Descent (SGD) and Adam.

11.5.6. Epochs and Iterations

Two terms frequently used during training are epoch and iteration.

An iteration refers to a single update of the model parameters using one batch of training data.

An epoch refers to one complete pass through the entire training dataset.

For example, if a dataset contains 1,000 samples and batches of 100 samples are used:

- 10 iterations complete one epoch.
- Multiple epochs are typically required for effective learning.

Table 11.7. Epochs vs Iterations.

Term	Meaning
Iteration	One parameter update using a batch
Batch	Subset of training samples
Epoch	One complete pass through the dataset
Training Loop	Repeated execution of all training steps

FOR AUTHOR USE ONLY

Table 11.8. Common Loss Functions in PyTorch.

Loss Function	Typical Use Case
MSELoss	Regression problems
L1Loss	Regression with outlier resistance
CrossEntropyLoss	Multi-class classification
BCELoss	Binary classification
BCEWithLogitsLoss	Binary classification with logits

```
# Import PyTorch

import torch.nn as nn

# Mean Squared Error Loss

loss_function = nn.MSELoss()

print(loss_function)
```

Code Snippet 11.18. Creating a Loss Function.

11.5.7. Common Optimizers

Optimizers update model parameters using gradients calculated during backpropagation.

Different optimizers may converge at different speeds depending on the problem.

Table 11.9. Common Optimizers in PyTorch.

Optimizer	Description
SGD	Standard gradient descent optimizer
Adam	Adaptive optimizer widely used in deep learning
RMSprop	Optimizer with adaptive learning rates
Adagrad	Adjusts learning rates based on historical gradients

FOR AUTHOR USE ONLY

```python
# Import PyTorch
import torch.optim as optim

# Create optimizer

optimizer = optim.Adam(
    model.parameters(),
    lr=0.001
)

print(optimizer)
```

Code Snippet 11.19. Creating an Optimizer.

11.5.8. Building a Simple Neural Network

Before constructing a training loop, a model must be defined. The following example creates a simple neural network with one input layer and one output layer.

```python
# Import libraries

import torch
import torch.nn as nn

# Define model

class SimpleModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        return self.linear(x)

# Create model

model = SimpleModel()

print(model)
```

Code Snippet 11.20. Defining a Simple Model.

11.5.9. Complete Training Loop

The following example demonstrates a complete PyTorch training loop.

FOR AUTHORISE ONLY

```python
# Import libraries
import torch
import torch.nn as nn
import torch.optim as optim

# Sample dataset
X = torch.tensor([[1.0], [2.0], [3.0], [4.0]])
y = torch.tensor([[2.0], [4.0], [6.0], [8.0]])

# Model
model = nn.Linear(1, 1)

# Loss function
criterion = nn.MSELoss()

# Optimizer
optimizer = optim.SGD(
    model.parameters(),
    lr=0.01
)
# Training loop
epochs = 100

for epoch in range(epochs):
    # Forward pass
    predictions = model(X)

    # Calculate loss
    loss = criterion(predictions, y)

    # Clear old gradients
    optimizer.zero_grad()

    # Backward pass
    loss.backward()

    # Update parameters
    optimizer.step()

    # Display progress
    if (epoch + 1) % 10 == 0:
        print(
            f"Epoch {epoch+1}/{epochs}, "
            f"Loss: {loss.item():.4f}"
        )
```

Code Snippet 11.21. Complete Training Loop.

11.5.10. Monitoring Training Progress

Monitoring loss values helps determine whether the model is learning successfully.

As training progresses, the loss should generally decrease.

```
# Store loss values

loss_history = []

for epoch in range(epochs):

    predictions = model(X)

    loss = criterion(predictions, y)

    optimizer.zero_grad()

    loss.backward()

    optimizer.step()

    loss_history.append(loss.item())
```

Code Snippet 11.22. Recording Training Loss.

FOR AUTHOR USE ONLY

11.5.11. Saving a Trained Model

After training is complete, the model can be saved for future use. Saving prevents the need to retrain the model every time it is needed.

```
# Save model parameters

torch.save(
    model.state_dict(),
    "simple_model.pth"
)

print("Model Saved Successfully")
```

Code Snippet 11.23. Saving a Model.

11.5.12. Loading a Saved Model

Previously saved parameters can be loaded into a model when required.

```
# Create model structure

model = nn.Linear(1, 1)

# Load saved parameters

model.load_state_dict(
    torch.load("simple_model.pth")
)
```

```
# Evaluation mode

model.eval()

print("Model Loaded Successfully")
```

Code Snippet 11.24. Loading a Saved Model.

At this point, the fundamental concepts required to build PyTorch models have been covered. Students now understand how tensors store data, how Autograd calculates gradients, and how training loops use those gradients to improve model performance. The next section combines these concepts into a complete hands-on project that demonstrates the full machine learning workflow in PyTorch.

11.6. Binary Classification Using PyTorch

After learning about tensors, Autograd, and training loops, it is helpful to apply these concepts in a complete machine learning project. A simple neural network will be trained to perform binary classification using a synthetic dataset. Binary classification is a machine learning task where data points are assigned to one of two possible classes. Examples include spam detection, fraud detection, disease diagnosis, and sentiment classification. In this example, the goal is to predict whether a sample belongs to Class 0 or Class 1 based on two input features. This project demonstrates the complete PyTorch workflow, including data preparation, model creation, training, and prediction.

FOR AUTHOR USE ONLY

11.6.1. Creating a Synthetic Dataset

The dataset contains two numerical features and one target label. The target values are either 0 or 1, representing the two classes.

```
# Import PyTorch

import torch

# Input features

X = torch.tensor([
    [1.0, 2.0],
    [2.0, 3.0],
    [3.0, 4.0],
    [4.0, 5.0],
    [5.0, 6.0],
    [6.0, 7.0]
])

# Target labels
```

```
y = torch.tensor([
    [0.0],
    [0.0],
    [0.0],
    [1.0],
    [1.0],
    [1.0]
])

print("Feature Shape:", X.shape)
print("Target Shape:", y.shape)
```

Code Snippet 11.25. Creating a Synthetic Dataset.

11.6.2. Defining the Neural Network

The model consists of:

- Two input features
- One hidden layer
- One output neuron

A sigmoid activation function is used because the task is binary classification.

FOR AUTHOR USE ONLY

```
# Import libraries

import torch.nn as nn

# Create model

class BinaryClassifier(nn.Module):

    def __init__(self):

        super().__init__()

        self.network = nn.Sequential(

            nn.Linear(2, 4),
            nn.ReLU(),

            nn.Linear(4, 1),
            nn.Sigmoid()
        )

    def forward(self, x):

        return self.network(x)

# Create model instance

model = BinaryClassifier()

print(model)
```

Code Snippet 11.26. Defining the Classification Model.

11.6.3. Defining the Loss Function and Optimizer

For binary classification, Binary Cross Entropy Loss is commonly used. The Adam optimizer is selected because it often converges faster than traditional gradient descent methods.

```
# Import optimizer

import torch.optim as optim

# Binary classification loss

criterion = nn.BCELoss()

# Optimizer

optimizer = optim.Adam(
    model.parameters(),
    lr=0.01
)
```

Code Snippet 11.27. Loss Function and Optimizer.

11.6.4. Training the Model

The model is trained for multiple epochs. During each epoch, predictions are generated, loss is calculated, gradients are computed, and model parameters are updated.

FOR AUTHOR USE ONLY

```python
# Number of training epochs
epochs = 200

for epoch in range(epochs):

    # Forward pass

    predictions = model(X)

    # Calculate loss

    loss = criterion(predictions, y)

    # Reset gradients

    optimizer.zero_grad()

    # Backpropagation

    loss.backward()

    # Update parameters

    optimizer.step()

    # Display progress

    if (epoch + 1) % 20 == 0:
        print(
            f"Epoch {epoch+1}/{epochs}, "
            f"Loss: {loss.item():.4f}"
        )
```

FOR AUTHOR USE ONLY

Code Snippet 11.28. Training the Neural Network.

11.6.5. Making Predictions

Once training is complete, the model can be used to classify new samples.

```python
# Disable gradient tracking

with torch.no_grad():

    new_samples = torch.tensor([
        [2.5, 3.5],
        [5.5, 6.5]
    ])

    predictions = model(new_samples)

    print(predictions)
```

Code Snippet 11.29. Making Predictions.

The output values represent probabilities between 0 and 1. Values closer to 0 indicate Class 0, while values closer to 1 indicate Class 1.

11.6.6. Converting Probabilities into Class Labels

Probabilities can be converted into actual class predictions using a threshold value of 0.5.

```
# Disable gradient tracking

with torch.no_grad():

    predictions = model(new_samples)

    predicted_classes = (
        predictions >= 0.5
    ).float()

    print(predicted_classes)
```

Code Snippet 11.30. Converting Probabilities to Classes.

11.6.7. Sample Results

The exact output values may vary because model weights are initialized randomly. However, a trained model should produce predictions like those shown below.

FOR AUTHOR USE ONLY

Table 11.10. Sample Model Predictions.

Input Features	Predicted Probability	Predicted Class
[2.5, 3.5]	0.1861	Class 0
[5.5, 6.5]	0.9734	Class 1

Chapter — 12 Data Engineering for Machine Learning

Prelude

Data engineering plays an essential role in the development of effective machine learning systems. Before a machine learning model is trained, the dataset must be prepared in a structured and meaningful way. Poorly prepared data can reduce model performance, increase bias, and lead to unreliable predictions. Therefore, preprocessing techniques such as feature scaling, imbalance handling, and data augmentation are important steps in the machine learning pipeline.

This work focuses on three important data engineering techniques used in machine learning: Feature Scaling, Handling Imbalance, and Data Augmentation. Feature scaling ensures that numerical features are transformed into comparable ranges. Handling imbalance addresses unequal class distribution in classification datasets. Data augmentation increases the diversity of training data by applying controlled transformations, especially in image-based machine learning tasks.

FOR AUTHOR USE ONLY

The implementation is carried out using Python. The Diabetes Health Indicators dataset is used to demonstrate feature scaling and class imbalance handling. The Brain Tumor MRI Dataset is used to demonstrate image data augmentation. The main objective of this work is to explain how these preprocessing techniques improve the quality of input data before model training.

Introduction

Machine learning models learn patterns from data. The quality of the data directly affects the quality of the model. In real-world datasets, raw data is often not immediately suitable for machine learning. Numerical features may have

different value ranges, class labels may be unevenly distributed, and image datasets may contain limited variations. These problems can affect the learning process and reduce the generalization ability of the model.

Data engineering for machine learning focuses on preparing and transforming data before it is passed into a model. This process helps machine learning algorithms learn more effectively from the available information. In this work, three preprocessing techniques are considered.

Feature scaling is used to transform numerical values into a comparable range. This is especially important for algorithms that depend on distance calculations or gradient-based optimization. Handling imbalance is used when one class has significantly more samples than the others. This is important in classification tasks because an imbalanced dataset can cause the model to favor the majority class. Data augmentation is used to artificially increase data diversity by applying transformations to existing samples.

Together, these techniques improve the quality, balance, and robustness of the dataset. They are not model training techniques themselves, but they strongly influence the performance and reliability of machine learning systems.

FOR AUTHOR USE ONLY

Dataset Description

Two datasets were used in this work. The first dataset is the Diabetes Health Indicators dataset. This dataset contains health-related survey features such as BMI, blood pressure, cholesterol status, physical activity, age, education, income, and diabetes status. The target column used in this work is Diabetes_012, which represents three classes: no diabetes, prediabetes, and diabetes.

The diabetes dataset is suitable for feature scaling because it contains several numerical and ordinal features with different value ranges. It is also suitable for handling imbalance because the distribution of diabetes classes is not equal. This makes it useful for demonstrating class imbalance and the effect of balancing techniques.

The second dataset is the Brain Tumor MRI Dataset. It contains MRI images categorized into four classes: glioma, meningioma, pituitary tumor, and no tumor. This dataset is used only for data augmentation because image data provides a clear visual demonstration of transformations such as rotation, flipping, zooming, and brightness adjustment.

Data Leakage

Data leakage is one of the most important risks in machine learning preprocessing. It occurs when information from the test data is accidentally

used during training or preprocessing. When this happens, the model may show unrealistically high-performance during evaluation, but it may not perform well on unseen real-world data.

In preprocessing, data leakage can occur if operations such as feature scaling, missing value imputation, or synthetic oversampling are performed before the train-test split. For example, if StandardScaler is fitted on the entire dataset, the mean and standard deviation are calculated using both training and test data. This allows information from the test set to influence the training process indirectly.

To avoid this issue, the dataset must first be divided into training and test sets. After the split, preprocessing methods should be fitted only on the training data. The learned transformation should then be applied to the test data. This ensures that the test set remains unseen during training and provides a more realistic evaluation.

This principle is especially important when using SMOTE. SMOTE should only be applied to the training data because it creates synthetic minority class samples. If SMOTE is applied before the train-test split, synthetic samples may be generated using information that later appears in the test set. This can lead to overly optimistic results. Therefore, in this work, the train-test split was performed before scaling and SMOTE was applied only to the training data.

FOR AUTHOR USE ONLY

Feature Engineering

Feature engineering is the process of preparing, selecting, and transforming input variables so that they become more useful for machine learning models. It is an important part of data engineering because the quality of input features strongly affects model performance. Even powerful algorithms may perform poorly if the features are not properly prepared.

One important part of feature engineering is missing value handling. Real-world datasets often contain missing values due to incomplete records or data collection errors. These missing values must be handled before model training. Common methods include removing incomplete rows, replacing missing values with the mean or median, or using more advanced imputation techniques.

Feature selection is another important step. Not every feature contributes equally to prediction. Some features may be irrelevant, redundant, or noisy. Feature selection helps remove unnecessary variables and keeps the most useful ones. This can improve model performance, reduce overfitting, and make the model easier to interpret.

Categorical encoding is also part of feature engineering. Machine learning models usually require numerical input, so categorical variables must be converted into numerical form. One-Hot Encoding is used when categories do not have a natural order, such as color or city names. Label Encoding is used when categories have an order or when converting class labels into numerical values.

In this work, the Diabetes Health Indicators dataset already contains numerical and encoded variables. Therefore, the focus was placed on feature scaling and imbalance handling. However, in general machine learning workflows, feature engineering steps such as missing value handling, feature selection, One-Hot Encoding, and Label Encoding are essential before model training.

12.1. Feature Scaling

Feature scaling is a preprocessing technique used to transform numerical features into a common scale. In many datasets, different features are measured using different units and value ranges. For example, BMI may have values between 12 and 90, while another feature such as income may be represented using a smaller categorical scale. When such features are used without scaling, features with larger numerical values may influence the model more strongly than features with smaller values.

FOR AUTHOR USE ONLY

Feature scaling is especially important for algorithms such as Logistic Regression, Support Vector Machines, K-Nearest Neighbors, and Neural Networks. These models are sensitive to the scale of input features because they depend on distance calculations or optimization methods. If the input features are not scaled, the model may learn inefficiently or produce biased results.

Two common feature scaling techniques are Standardization and Min-Max Scaling.

Standardization transforms a feature so that it has a mean of zero and a standard deviation of one. It is calculated as:

$$\boldsymbol{z} = \frac{\boldsymbol{x}-\boldsymbol{\mu}}{\boldsymbol{\sigma}}, \qquad \textit{(Eq.12.1)}$$

Where:

- z is the standardized value
- x is the original value
- μ is the mean of the feature
- σ is the standard deviation

Min-Max Scaling transforms values into a fixed range, usually between 0 and 1. It is calculated as:

$$x_{scaled} = \frac{x - x_{min}}{x_{max} - x_{min}}, \qquad \textit{(Eq.12.2)}$$

Where:

- x_{scaled} is the normalized value
- x is the original value
- x_{min} is the minimum value of the feature
- x_{max} is the maximum value of the feature

In this work, selected numerical and ordinal features from the Diabetes Health Indicators dataset were scaled. The target column Diabetes_012 was not scaled because it represents class labels. Binary columns were also not selected for the main scaling demonstration because they already contain 0 and 1 values.

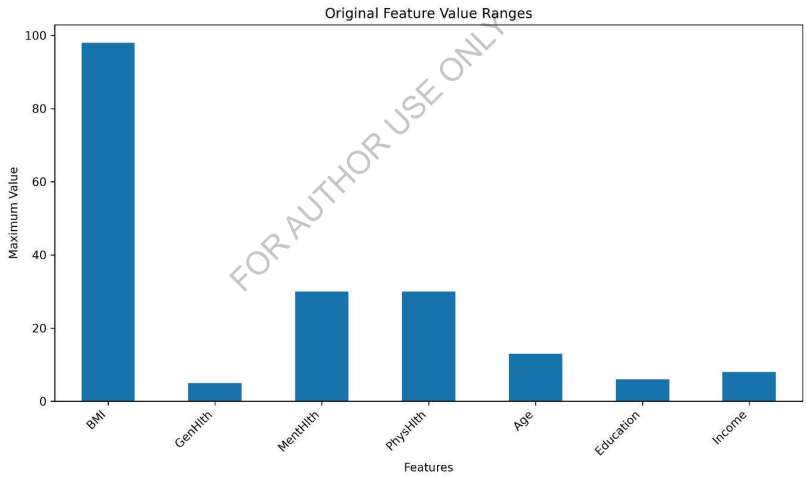


Figure 12.1. Original Feature Value Ranges.

Figure 12.1 shows the original value ranges of the selected features before scaling. The selected features have different maximum values, which shows that they are not on the same numerical scale.

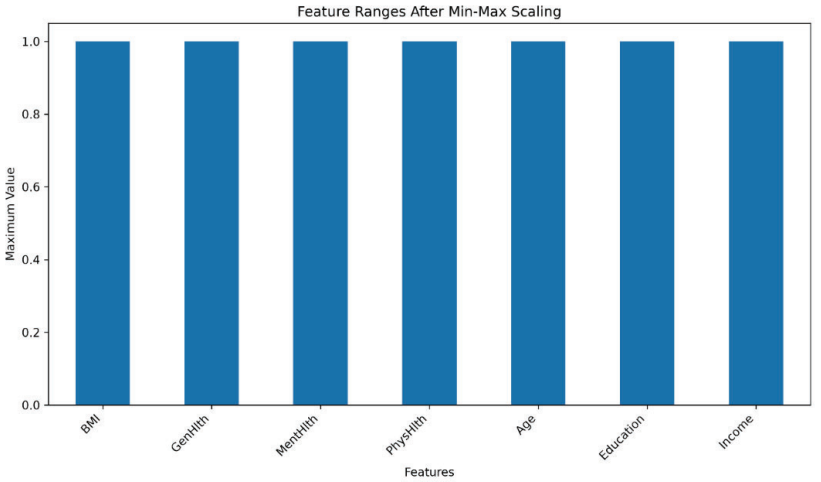


Figure 12.2. Feature Ranges After Min-Max Scaling.

Figure 12.2 shows the feature ranges after applying Min-Max scaling. After scaling, all selected features are transformed into a common range between 0 and 1. This makes the features more comparable and suitable for machine learning algorithms that are sensitive to feature scale.

FOR AUTHOR USE ONLY

12.2. Handling Imbalance

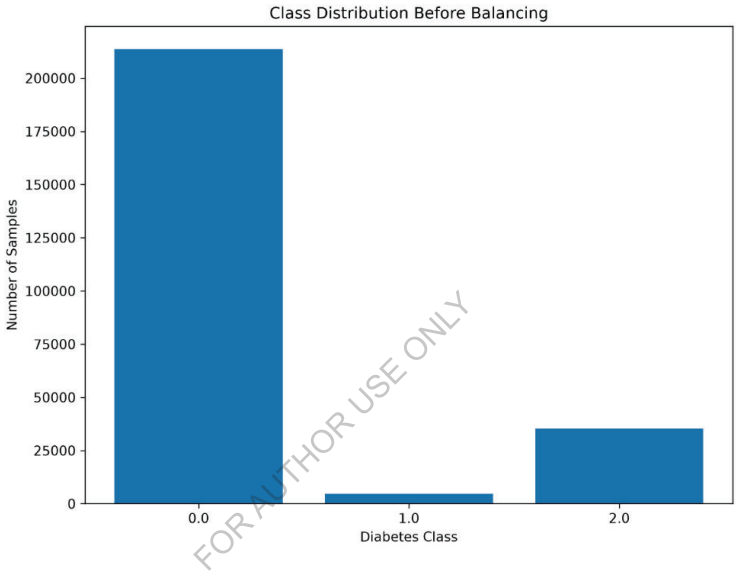
Class imbalance occurs when the target classes in a dataset are not represented equally. In classification problems, this can be a serious issue because the model may become biased toward the majority class. For example, if most records belong to the no diabetes class, a model may learn to predict the majority class more often and fail to correctly identify minority classes such as prediabetes or diabetes.

Accuracy can be misleading in imbalanced datasets. A model may achieve high accuracy by predicting the majority class but still perform poorly on the minority class. Therefore, imbalance handling is important in machine learning tasks where minority class detection is important.

There are several methods used to handle class imbalance. Common approaches include oversampling, undersampling, class weighting, and synthetic sample generation. Oversampling increases the number of minority class samples. Undersampling reduces the number of majority class samples. Class weighting gives higher importance to minority class samples during model training. SMOTE, which stands for Synthetic Minority Over-sampling

Technique, creates synthetic minority class samples instead of simply duplicating existing ones.

In this work, SMOTE was used to handle imbalance in the Diabetes Health Indicators dataset. The dataset was first split into training and testing sets. SMOTE was applied only to the training data. This is important because the test data must remain unchanged to provide a realistic evaluation of the model.



FOR AUTHOR USE ONLY

Figure 12.3. Class Distribution Before Balancing.

Figure 12.3 shows the original class distribution of the Diabetes_012 target variable. The class distribution is not equal, showing that the dataset contains class imbalance.

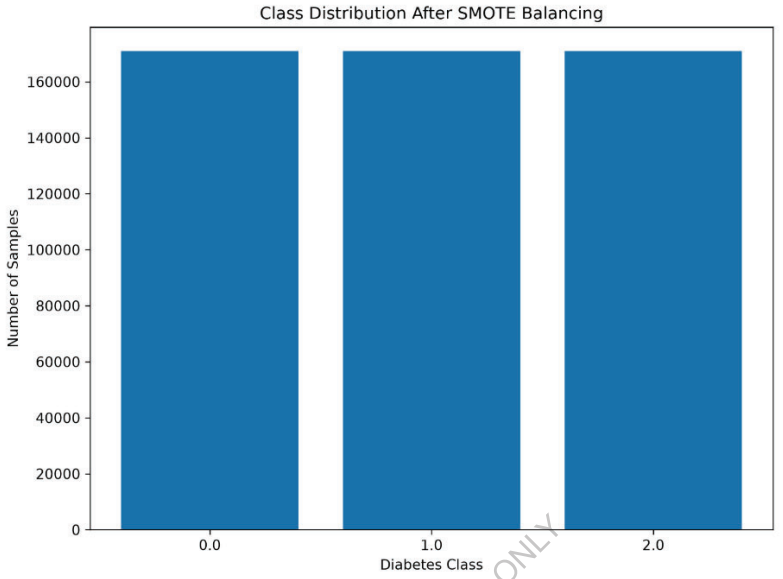


Figure 12.4. Class Distribution After SMOTE Balancing.

FOR AUTHOR USE ONLY

Figure 12.4 shows the class distribution after applying SMOTE to the training dataset. The minority classes are increased using synthetic samples, resulting in a more balanced training dataset. This allows the model to learn more equally from all classes.

12.3. Data Augmentation

Data augmentation is a preprocessing technique used to increase the diversity of training data. It is commonly used in image-based machine learning tasks. Instead of collecting new images, augmentation applies transformations to existing images to create new variations. This helps the model become more robust and reduces overfitting.

In image classification, a model may perform poorly if the training images are too limited or too similar. Data augmentation introduces controlled variations such as rotation, flipping, zooming, cropping, brightness adjustment, and noise addition. These transformations help the model learn patterns that are not dependent on one fixed image orientation or appearance.

However, augmentation should be applied carefully, especially in medical image analysis. Excessive transformation may change clinically meaningful information. For example, strong distortion of MRI images may alter important structures. Therefore, only moderate transformations should be used.

In this work, image augmentation was demonstrated using a sample MRI image from the Brain Tumor MRI Dataset. The applied augmentation techniques include rotation, horizontal flipping, zooming, and brightness adjustment.

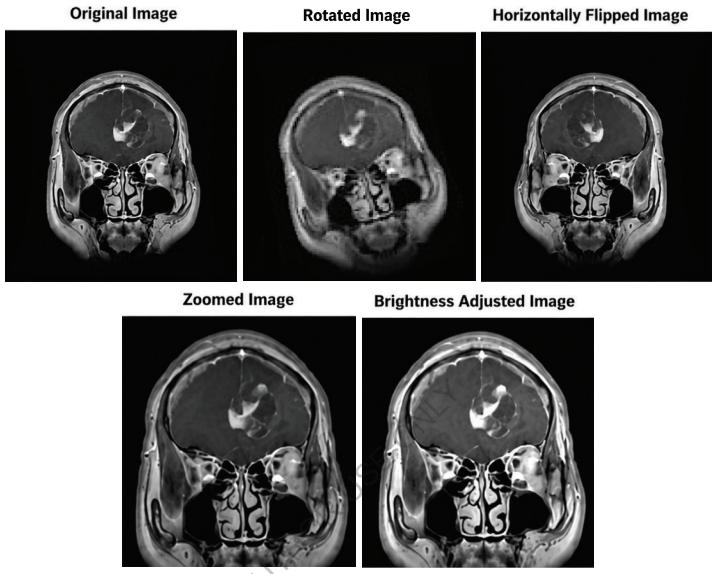


Figure 12.5. Data Augmentation Examples.

Figure 12.5 shows the original MRI image and the augmented versions. The transformations create visual variations while preserving the basic image structure. These augmented images can help improve model generalization when used during training.

12.4. Machine Learning Pipelines

A machine learning pipeline is a structured workflow that connects preprocessing steps and model training into one complete process. Instead of applying each step separately, a pipeline allows the same sequence of operations to be applied consistently during training, testing, and future prediction.

Pipelines are important because they help reduce errors in machine learning workflows. For example, if feature scaling is performed manually, there is a risk that the training data and test data may be processed differently. A pipeline helps avoid this by storing the preprocessing steps together with the model.

In Scikit-Learn, a pipeline can combine steps such as scaling, encoding, feature selection, and classification. When handling imbalanced datasets with SMOTE, the imbalanced-learn Pipeline is used because it supports resampling methods. In this work, the pipeline combines StandardScaler, SMOTE, and Logistic Regression.

A simple pipeline example is shown below:

```
pipeline = Pipeline(steps=[
    ("scaler", StandardScaler())
    ("smote", SMOTE(random_state=42)),
    ("classifier", LogisticRegression(max_iter=1000, random_state=42))
])f
```

Code Snippet 12.1. Machine Learning Pipeline Example.

In this pipeline, StandardScaler first scales the numerical features. SMOTE then balances the training data by creating synthetic samples for minority classes. Finally, Logistic Regression is used as the classification model.

Using a pipeline improves reproducibility because the same preprocessing steps are applied every time the model is trained or tested. It also supports deployment consistency because the preprocessing workflow and the model can be treated as one connected system. This reduces the chance of applying different preprocessing steps during development and real-world use.

FOR AUTHOR REVIEW ONLY

12.4.1. Implementation

The implementation was carried out using Python. The main libraries used were pandas, matplotlib, Scikit-Learn, imbalanced-learn, and Pillow. The Diabetes Health Indicators dataset was used for feature scaling, handling imbalance, model performance comparison, and the pipeline example. The Brain Tumor MRI Dataset was used for image data augmentation.

The implementation was organized into separate Python scripts. Each script focused on one major preprocessing technique or supporting experiment. The first script implemented feature scaling, the second script handled class imbalance using SMOTE, the third script demonstrated image data augmentation, the fourth script compared model performance before and after preprocessing, and the fifth script demonstrated a machine learning pipeline.

The implementation was updated to avoid data leakage. In the tabular dataset, the train-test split was performed before applying scaling or SMOTE. The scalers were fitted only on the training data and then applied to the test data. SMOTE was applied only to the training data, while the test data remained

unchanged. This ensures that the evaluation is performed on unseen data and gives a more realistic understanding of model performance.

```
X_train, X_test, y_train, y_test = train_test_split(
    X,
    y,
    test_size=0.2,
    stratify=y,
    random_state=42
)
```

Code Snippet 12.2. Train-Test Split Before Preprocessing.

This step separates the dataset into training and test sets before applying preprocessing. Stratification was used to preserve the original class distribution in both the training and test sets.

```
standard_scaler = StandardScaler()

X_train_standard_scaled = standard_scaler.fit_transform(X_train)
X_test_standard_scaled = standard_scaler.transform(X_test)
```

Code Snippet 12.3. Feature Scaling Without Data Leakage.

FOR JUTAR USE ONLY

The scaler was fitted only on the training data. The same fitted scaler was then used to transform the test data. This prevents the test set from influencing the scaling parameters such as mean and standard deviation.

The feature scaling script generated output files showing the original feature ranges and the feature ranges after Min-Max Scaling. These outputs were used to visually demonstrate how features with different value ranges can be transformed into a more comparable scale.

```
smote = SMOTE(random_state=42)

X_train_balanced, y_train_balanced = smote.fit_resample(X_train, y_train)
```

Code Snippet 12.4. Applying SMOTE Only to Training Data.

SMOTE was applied only to the training data after scaling. The test data was not balanced or modified. This is important because the test set should represent unseen real-world data. Applying SMOTE before the train-test split could create synthetic samples that are too like test samples, which would lead to data leakage.

The imbalance handling script generated class distribution charts before and after SMOTE. These outputs showed how the original diabetes dataset had

unequal class distribution and how SMOTE balanced the training dataset by creating synthetic samples for minority classes.

```
def zoom_image(image, zoom_factor=1.2):

    width, height = image.size

    # Resize image to a larger size
    resized_image = image.resize(
        (int(width * zoom_factor), int(height * zoom_factor))
    )

    # Calculate crop boundaries
    left = (resized_image.width - width) // 2
    top = (resized_image.height - height) // 2
    right = left + width
    bottom = top + height

    # Crop back to original size
    zoomed_image = resized_image.crop((left, top, right, bottom))

    return zoomed_image


# Find one sample MRI image from the dataset folder
sample_image_path = find_sample_image(DATASETS_DIR)

# Open image and convert to RGB
original_image = Image.open(sample_image_path).convert("RGB")

# Resize image for consistent visualization
original_image = original_image.resize((256, 256))

# Apply augmentation techniques
rotated_image = original_image.rotate(15)
flipped_image = ImageOps.mirror(original_image)
zoomed_image = zoom_image(original_image, zoom_factor=1.2)

brightness_enhancer = ImageEnhance.Brightness(original_image)
brightness_image = brightness_enhancer.enhance(1.4)
```

FOR AUTHOR USE ONLY

Code Snippet 12.5. Data Augmentation for MRI Images.

The data augmentation script automatically selected one MRI image from the Brain Tumor MRI Dataset and converted it into RGB format. The image was resized to 256 × 256 pixels for consistent visualization.

Four augmentation techniques were applied: rotation, horizontal flipping, zooming, and brightness adjustment. The zoom operation was performed using a separate function that resized the image by a factor of 1.2 and then center-cropped it back to the original size. This allowed the augmented image to keep the same dimensions as the original image.

The script saved the original image and the augmented versions as separate output files. It also created a combined visualization named data_augmentation_examples.png and saved a summary file describing each augmentation technique.

```
# Build machine learning pipeline

pipeline = Pipeline(steps=[
    ("scaler", StandardScaler()),
    ("smote", SMOTE(random_state=42)),
    ("classifier", LogisticRegression(max_iter=1000, random_state=42))
])
```

Code Snippet 12.6. Machine Learning Pipeline.

A machine learning pipeline was added to combine preprocessing and model training into one workflow. The pipeline includes StandardScaler, SMOTE, and Logistic Regression. StandardScaler performs feature scaling, SMOTE handles class imbalance, and Logistic Regression performs classification.

Using a pipeline improves reproducibility because the same preprocessing steps are applied consistently during training and prediction. It also supports deployment consistency because the preprocessing workflow and the model can be treated as one connected process.

FOR AUTHOR USE ONLY

```
results_scaling_smote = evaluate_model(
    model_name="Scaling + SMOTE",
    X_train_data=X_train_scaled_smote,
    X_test_data=X_test_scaled,
    y_train_data=y_train_smote,
    y_test_data=y_test
)

# Save comparison results

results_df = pd.DataFrame([
    results_no_preprocessing,
    results_scaling,
    results_scaling_smote
])

results_df.to_csv(
    OUTPUT_DIR / "model_performance_comparison.csv",
    index=False
)
```

Code Snippet 12.7. Model Performance Comparison.

A model performance comparison was added to demonstrate the practical effect of preprocessing. Logistic Regression was evaluated in three cases: without preprocessing, with feature scaling only, and with feature scaling combined with SMOTE.

The comparison used Macro F1-score because the dataset is imbalanced. Macro F1-score gives equal importance to each class and is therefore more suitable than accuracy alone for imbalanced classification problems. The results were saved as a CSV file and visualized using a bar chart.

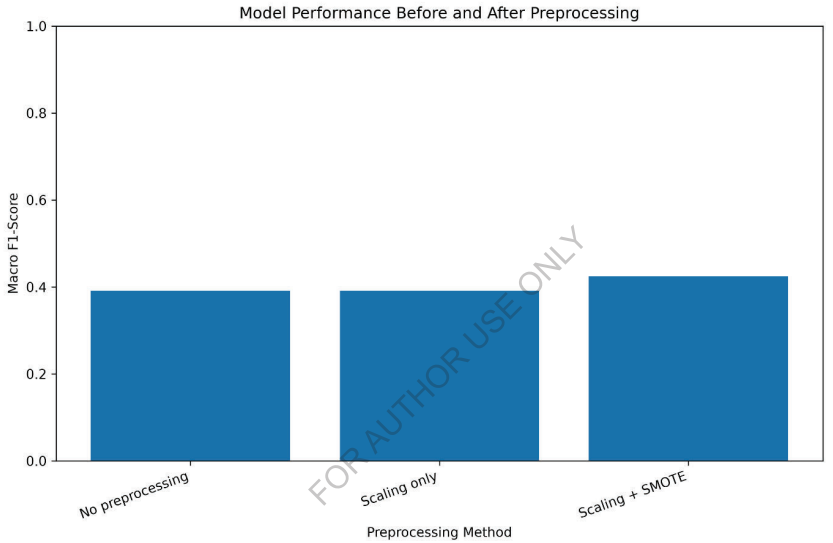


Figure 12.6. Model Performance Before and After Preprocessing.

Overall, the implementation shows how data engineering techniques improve the quality and usability of machine learning data. Feature scaling makes numerical features more comparable, SMOTE addresses class imbalance, data augmentation increases image variation, pipelines improve workflow consistency, and model performance comparison shows the practical impact of preprocessing.

12.4.2. Results and Discussion

The results show how different preprocessing techniques affect the quality and structure of machine learning data. The outputs generated from the implementation support the main topics of this chapter: feature scaling,

handling imbalance, data augmentation, machine learning pipelines, and model performance comparison.

The feature scaling results show that the selected features in the Diabetes Health Indicators dataset originally had different value ranges. Features such as BMI and physical health had larger numerical ranges compared to ordinal features such as age, education, and income. These differences can affect machine learning algorithms that depend on distance calculations or optimization. After applying Min-Max Scaling, the selected features were transformed into a common range, making them more comparable.

The updated feature scaling implementation also prevents data leakage. The dataset was split into training and test sets before scaling. The scaler was fitted only on the training data and then applied to the test data. This ensures that information from the test set does not influence the preprocessing step.

The imbalance handling results show that the Diabetes Health Indicators dataset had an unequal class distribution. The majority class contained more samples than the minority classes. This can cause a classification model to favor the majority class and perform poorly on minority classes. After applying SMOTE to the training data, the class distribution became more balanced. This helps the model learn better from minority class examples.

FOR AUTHOR USE ONLY

SMOTE was applied only to the training data, while the test data was kept unchanged. This is important because the test set should represent unseen real-world data. Applying SMOTE before the train-test split could lead to data leakage and unrealistic evaluation results.

The data augmentation results show how new image variations can be generated from an existing MRI image. The augmentation process applied moderate transformations such as rotation, horizontal flipping, zooming, and brightness adjustment. These transformations increased the visual diversity of the image data while preserving the main structure of the MRI image.

Data augmentation is useful in image-based machine learning because it can help reduce overfitting and improve generalization. However, in medical imaging, augmentation must be applied carefully. Excessive transformations may change important medical patterns in the image. Therefore, the augmentation used in this implementation was kept moderate and controlled.

The machine learning pipeline demonstrates how preprocessing and model training can be connected into one workflow. The pipeline combined StandardScaler, SMOTE, and Logistic Regression. This improves reproducibility because the same preprocessing steps are applied consistently

during training and prediction. It also supports deployment consistency because preprocessing and model training are handled as one connected process.

The model performance comparison shows the practical impact of preprocessing. Logistic Regression was tested in three cases: without preprocessing, with feature scaling only, and with feature scaling combined with SMOTE. Macro F1-score was used because it gives equal importance to each class and is more suitable for imbalanced datasets than accuracy alone.

The performance comparison figure included at the end of the Implementation section shows that preprocessing can change the model's class-level performance. Scaling helps the model work with features in a more comparable range, while SMOTE helps address minority class underrepresentation. Even if accuracy does not always increase, Macro F1-score gives a better understanding of performance across all classes.

Overall, the results show that data engineering is an essential step before machine learning model training. Feature scaling improves numerical consistency, SMOTE improves class balance in the training data, data augmentation increases image diversity, and pipelines improve reproducibility. Together, these techniques help create a cleaner, more balanced, and more reliable machine learning workflow.

FOR AUTHOR USE ONLY

Best Practices

Data engineering techniques should be applied carefully to avoid incorrect evaluation results and unreliable machine learning workflows. The following best practices are important when applying feature scaling, imbalance handling, and data augmentation.

The dataset should always be split into training and test sets before applying preprocessing. This prevents information from the test set from influencing the training process. If preprocessing is performed before the split, the model evaluation may become overly optimistic.

Scalers should be fitted only on the training data. After fitting the scaler on the training data, the same scaler should be used to transform the test data. This ensures that the test data remains unseen during preprocessing.

SMOTE should only be applied to the training data. The test data should not be balanced using SMOTE because it should represent real unseen data. Applying SMOTE before the train-test split can cause data leakage and unrealistic model performance.

Data augmentation should be used carefully, especially in medical imaging. Moderate transformations such as small rotations, flipping, zooming, and brightness changes can improve data diversity. However, excessive augmentation may change important medical features and reduce the reliability of the dataset.

Preprocessing steps should be stored together with the model whenever possible. Machine learning pipelines help achieve this by combining preprocessing and model training into one workflow. This improves reproducibility and ensures that the same steps are applied during training, testing, and deployment.

Model performance should be evaluated using suitable metrics. In imbalanced classification problems, accuracy alone may not be enough. Metrics such as Macro F1-score, precision, and recall provide a better understanding of performance across different classes.

12.5 Conclusion

This chapter explained important data engineering techniques used in machine learning: feature scaling, handling imbalance, and data augmentation. These techniques improve the quality of data before model training and help machine learning models learn more effectively.

FOR AUTHOR USE ONLY

Feature scaling was used to transform numerical features into comparable ranges. This is important for algorithms that are sensitive to feature values and distance calculations. The implementation also showed that scaling should be performed after the train-test split, and the scaler should be fitted only on the training data to avoid data leakage.

Handling imbalance was demonstrated using SMOTE. The original diabetes dataset had unequal class distribution, which can cause a model to favor the majority class. SMOTE helped balance the training data by creating synthetic samples for minority classes. However, SMOTE was applied only to the training data, while the test data remained unchanged.

Data augmentation was demonstrated using MRI images. Transformations such as rotation, flipping, zooming, and brightness adjustment were used to create image variations. This can improve data diversity and reduce overfitting in image-based machine learning. However, augmentation must be applied carefully in medical imaging to avoid changing important medical patterns.

The chapter also discussed data leakage, feature engineering, machine learning pipelines, and best practices. The pipeline example showed how preprocessing and model training can be combined into one reproducible

workflow. The model performance comparison showed the practical impact of preprocessing using Macro F1-score.

Overall, data engineering is an essential part of machine learning. Proper preprocessing improves data quality, supports fairer model evaluation, and helps create more reliable machine learning systems.

FOR AUTHOR USE ONLY

Chapter 13 — Model Deployment

13.1 Introduction to Model Deployment

13.1.1. What problem does it solve?

Training a model is only the first half of the work. A model that lives in a notebook produces no value for users; only a deployed model does. Yet deployment is where many ML projects stall, because the gap between a working training script and a reliable production system involves three concerns that training does not: persistence (saving and reloading weights without losing precision or graph structure), interface (exposing the model over a network so any client can call it), and efficiency (making inference fast enough and small enough to reach edge devices or high-traffic servers).

Each concern compounds the others. A model serialized in the wrong format may not survive a framework version upgrade. An API that sends the full batch synchronously will time out under load. A model that runs fine on a GPU server may be ten times too large for a mobile chip. Addressing all three in a principled way is what separates a proof-of-concept from a shipped product.

FOR AUTHOR USE ONLY

13.1.2 Three pillars covered in this guide

This document is organized around three practical pillars. Saving and Loading Models explains the serialization landscape across TensorFlow/Keras and PyTorch, which format to choose and why, and how to structure a checkpointing strategy that survives long training runs and framework upgrades. REST APIs covers wrapping a model in a network-accessible interface using FastAPI, with full attention to input validation, batching, and async inference, plus an overview of purpose-built serving systems (TensorFlow Serving, Triton) for production scale. Edge Deployment addresses running inference directly on-device—phones, microcontrollers, and browsers—using ONNX as a universal interchange format and quantization to close the compute gap between a research GPU and an embedded chip.

Every section presents the underlying principle, the practical decisions a practitioner must make, and runnable code samples in Python 3. The code is structured for clarity rather than peak performance, and the comments explain why each step matters, not merely what it does.

13.1.3 Mini case: taking a trained classifier to production

A computer-vision team trains an image classifier that reaches 94% top 1 accuracy on the internal test set. The model is a fine-tuned EfficientNetB0 in Keras, trained on a single A100 GPU over a weekend. The team now must ship it: a mobile app must be able to classify photos taken by users, and an internal

dashboard must be able to call it for batch auditing. Three problems surface immediately.

First, the checkpoint saved during training is a raw weight file; loading it on a different machine requires the identical class definition and Keras version. The team needs a self-contained artifact that bundles graph and weights together. Second, the dashboard is written in Go and cannot import TensorFlow; it needs an HTTP endpoint that accepts a JSON payload and returns a prediction. Third, the mobile app has a 15 MB bundle-size limit and a 50 ms latency budget; the full 29 MB EfficientNetB0 exceeds both.

The chapters that follow address each problem in order: Section 2 converts the checkpoint to a SavedModel; Section 3 wraps it in a FastAPI service behind an async inference queue; Section 4 exports and quantizes it to a TFLite model that fits in 7 MB and runs in 30 ms on a mid-range phone. The same three-step pattern applies to most supervised ML projects.

13.2. Saving and Loading Models

13.2.1 The format landscape: SavedModel, H5, TorchScript, ONNX

Before choosing a serialization format, it helps to understand what a saved model must contain: the computation graph (or equivalent architecture specification), the trained weights, and enough metadata that a consumer can reconstruct both without access to the original training code. Different formats make different trade-offs between portability, fidelity, and ecosystem coverage.

FOR AUTHOR USE ONLY

SavedModel is the canonical TensorFlow 2 format. It bundles the Keras graph as a traced TensorFlow function alongside all variable values in a directory tree, and it is the only format that TensorFlow Serving and TFLite conversion accept natively. HDF5 (“.h5”) is an older Keras-specific format that serializes the architecture as JSON and the weights as datasets; it works across Keras versions but does not carry TF-specific graph optimizations. TorchScript is PyTorch’s serialization format; it compiles a model to an intermediate representation that can be loaded and run without a Python interpreter, which is critical for C++ serving environments. ONNX is a vendor-neutral interchange format supported by most frameworks as either an export target or a runtime; it is the best choice when the training framework and the inference runtime are different.

The practical default for TensorFlow/Keras projects is SavedModel for any deployment path that stays inside the TF ecosystem, and ONNX for everything else. For PyTorch projects, TorchScript covers server-side C++ deployment while ONNX covers cross-framework portability.

13.2.2 Saving and restoring in Keras

Keras exposes two saving methods. model.save() writes a complete SavedModel or HDF5 artifact; tf.keras.models.load_model() restores it with no class definitions required. save_weights() and load_weights() write only the variable values, which is useful for checkpointing during training but requires the architecture to be rebuilt before loading.

Two details prevent most production bugs. Always pass save_format="tf" explicitly to save() to opt into SavedModel even when the filename has no extension; relying on defaults is fragile across Keras versions. When loading a model that contains custom layers or custom loss functions, pass a custom_objects dictionary so Keras knows how to reconstruct them.

```
import tensorflow as tf
from tensorflow.keras import layers, Model

# --- Build and train (abbreviated) ---
inputs = layers.Input(shape=(224, 224, 3))
x = layers.Conv2D(32, 3, activation="relu")(inputs)
x = layers.GlobalAveragePooling2D()(x)
outputs = layers.Dense(10, activation="softmax")(x)
model = Model(inputs, outputs, name="classifier")

model.compile(optimizer="adam",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
# model.fit(train_ds, validation_data=val_ds, epochs=20)
```

FOR AUTHOR USE ONLY

```
# --- Save: always use save_format="tf" for a self-contained artifact ---
model.save("artifacts/classifier", save_format="tf")
# Produces: artifacts/classifier/saved_model.pb + variables/

# --- Load: no class definitions needed ---
restored = tf.keras.models.load_model("artifacts/classifier")
restored.summary()          # graph and weights fully reconstructed

# --- Save / load weights only (training checkpoint pattern) ---
model.save_weights("checkpoints/epoch_10.ckpt")

fresh = Model(inputs, outputs)    # architecture must exist first
fresh.load_weights("checkpoints/epoch_10.ckpt")
```

Code snippet 13.1. Saving and loading a Keras model in SavedModel format.

Keras also integrates with tf.train.Checkpoint for fine-grained control over which objects are saved, which is useful when a training state includes the optimizer, a learning-rate schedule, and a step counter alongside the model weights.

```python
import tensorflow as tf

optimizer = tf.keras.optimizers.AdamW(1e-3, weight_decay=1e-4)
step      = tf.Variable(0, trainable=False)

# Bundle everything a training run needs to resume.
ckpt = tf.train.Checkpoint(model=model, optimizer=optimizer, step=step)
manager = tf.train.CheckpointManager(ckpt, "checkpoints/", max_to_keep=3)

for epoch in range(20):
    # ... training loop ...
    step.assign_add(1)
    save_path = manager.save()          # keeps only the last 3
    print(f"Saved checkpoint: {save_path}")

# Restore the latest checkpoint (e.g., after a preemption).
ckpt.restore(manager.latest_checkpoint).expect_partial()
# expect_partial() silences warnings about optimizer slots not yet
# created—safe when restoring before the first gradient step.
```

Code snippet 13.2. tf.train.Checkpoint for stateful training recovery.

13.2.3 PyTorch: state_dict and TorchScript

PyTorch offers two serialization modes. The state_dict approach saves only the tensor values and is the standard choice for checkpointing during training; loading requires the model class to be importable. TorchScript serializes both the graph and the weights into a .pt file that can be loaded with torch.jit.load() in a pure C++ environment with no Python dependency.

TorchScript compilation works in two ways. torch.jit.trace() records tensor operations during a single forward pass with a dummy input; it is the simpler path but does not capture Python control flow. torch.jit.script() parses the Python source and compiles it directly, handling branches and loops correctly at the cost of requiring type annotations on all arguments. Use trace for standard feed-forward models and script for models with dynamic control flow such as beam search or autoregressive sampling.

FOR AUTHOR USE ONLY

```python
import torch
import torch.nn as nn

class Classifier(nn.Module):
    def __init__(self, num_classes: int = 10):
        super().__init__()
        self.backbone = nn.Sequential(
            nn.Conv2d(3, 32, 3, padding=1), nn.ReLU(),
            nn.AdaptiveAvgPool2d(1), nn.Flatten(),
        )
        self.head = nn.Linear(32, num_classes)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.head(self.backbone(x))

model = Classifier()

# --- Option A: state_dict (requires class at load time) ---
torch.save(model.state_dict(), "classifier.pt")
```

```
restored = Classifier()              # class must be importable
restored.load_state_dict(torch.load("classifier.pt", weights_only=True))
restored.eval()

# --- Option B: TorchScript (no Python needed at inference time) ---
model.eval()
dummy = torch.randn(1, 3, 224, 224)  # representative input

# trace: simpler; misses branches that depend on tensor values
traced = torch.jit.trace(model, dummy)
traced.save("classifier_traced.pt")

# script: handles control flow; requires type annotations
scripted = torch.jit.script(model)
scripted.save("classifier_scripted.pt")

# Loading TorchScript in Python (or C++) — no class definition needed
loaded = torch.jit.load("classifier_scripted.pt")
loaded.eval()
with torch.no_grad():
    logits = loaded(dummy)           # (1, 10)
```

Code snippet 13.3. PyTorch state_dict and TorchScript export.

13.2.4 Versioning and checkpointing strategies

A model artifact is not just weights: it carries an implicit dependency on the framework version, the preprocessing pipeline, the class-label mapping, and any custom objects used at training time. Missing any of these at inference time produces silent errors that are harder to diagnose than an import failure. The most practical mitigation is to treat the model artifact as the center of a manifest that records everything a consumer needs.

FOR AUTHOR USE ONLY

Two habits prevent most production incidents. First, always save a metadata sidecar alongside the artifact: at minimum, the framework version, the input shape and dtype, and the class-to-index mapping. Second, name artifacts with a content hash or a training run ID rather than a date or epoch number, so two artifacts with the same name always contain the same weights. The checkpoint manager pattern from Listing 2.2 handles the latter automatically; the former requires a single extra write.

```
import json, hashlib, pathlib, tensorflow as tf

def save_with_metadata(model, export_dir: str, class_names: list[str]):
    """Save a SavedModel alongside a JSON manifest."""
    path = pathlib.Path(export_dir)
    path.mkdir(parents=True, exist_ok=True)

    # 1. Save the artifact.
    model.save(str(path / "model"), save_format="tf")

    # 2. Compute a content hash from the saved_model.pb bytes.
    pb_bytes = (path / "model" / "saved_model.pb").read_bytes()
    content_hash = hashlib.sha256(pb_bytes).hexdigest()[:12]
```

```
    # 3. Write the manifest.
    meta = {
        "framework": tf.__version__,
        "input_shape": model.input_shape,
        "input_dtype": str(model.input.dtype),
        "num_classes": len(class_names),
        "class_names": class_names,
        "content_hash": content_hash,
    }
    (path / "metadata.json").write_text(json.dumps(meta, indent=2))
    print(f"Saved model {content_hash} to {export_dir}")
    return content_hash
```

Code snippet 13.4. Saving a model with a metadata sidecar.

13.3. REST APIs

13.3.1 Why a REST layer belongs between model and client

A trained model is a Python object. Any client that cannot import Python—a browser, a mobile app, a Go microservice, a data pipeline written in Java—cannot call it directly. A REST API resolves this by making the model reachable over HTTP with a language-neutral JSON interface. The model runs once, behind the API, and every client speaks to it through the same endpoint regardless of language or platform.

The REST layer does more than translate protocols. It is the right place to enforce input validation (rejecting malformed requests before they reach the model), to implement batching (grouping concurrent single-item requests into one efficient forward pass), and to enforce rate limits, authentication, and observability. Embedding these concerns in the model code would couple inference logic to operational policy; the API layer keeps them separate.

FastAPI is the practical default for Python-based model serving at small to medium scale. It is built on Pydantic for request validation and Starlette for async HTTP, which means it handles concurrent requests without blocking the event loop—important because a model forward pass may take 10–100 ms and many requests may arrive simultaneously. For very high traffic, purpose-built systems such as TensorFlow Serving or NVIDIA Triton are preferable; Section 3.4 describes when and how to use them.

13.3.2 FastAPI: the minimal serving skeleton

The smallest viable FastAPI server for a Keras image classifier has four components: a startup handler that loads the model once into process memory, a Pydantic schema that validates the incoming payload, a predict endpoint that runs inference, and a health endpoint that lets load balancers verify the server is ready. Every production server adds more on top of these four, but none of the additions are useful until the skeleton is solid.

PROOFS FOR USE ONLY

The model must be loaded once at startup, not once per request. Loading a SavedModel typically takes 1–5 seconds and allocates hundreds of megabytes; doing it inside the handler would make every request pay that cost. FastAPI's lifespan context manager is the cleanest way to express startup and teardown logic introduced in FastAPI 0.93.

```
from contextlib import asynccontextmanager
from typing import Annotated
import numpy as np
import tensorflow as tf
from fastapi import FastAPI
from pydantic import BaseModel, Field

# --- Shared state: loaded once at startup ---
state: dict = {}

@asynccontextmanager
async def lifespan(app: FastAPI):
    # Runs before the server accepts any requests.
    state["model"] = tf.keras.models.load_model("artifacts/classifier")
    state["class_names"] = ["cat", "dog", "bird"]   # load from metadata
    yield
    # Runs on shutdown; release GPU memory explicitly if needed.
    state.clear()

app = FastAPI(title="Classifier API", lifespan=lifespan)

# --- Request / response schemas ---
class PredictRequest(BaseModel):
    # Flat list of floats representing a pre-processed image.
    pixels: list[float] = Field(..., min_length=1)
    height: int = Field(224, ge=1, le=1024)
    width:  int = Field(224, ge=1, le=1024)

class PredictResponse(BaseModel):
    label:      str
    confidence: float

@app.post("/predict", response_model=PredictResponse)
def predict(req: PredictRequest) -> PredictResponse:
    # Pydantic has already validated the payload; no try/except needed here.
    img = np.array(req.pixels, dtype=np.float32).reshape(
        1, req.height, req.width, 3
    )
    probs  = state["model"].predict(img, verbose=0)[0]  # (num_classes,)
    idx    = int(np.argmax(probs))
    return PredictResponse(label=state["class_names"][idx],
                           confidence=float(probs[idx]))

@app.get("/health")
def health() -> dict:
    return {"status": "ok", "model_loaded": "model" in state}

# Run with: uvicorn main:app --host 0.0.0.0 --port 8000
```

FOR AUTHOR USE ONLY

Code snippet 13.5. Minimal FastAPI model server with lifespan loading.

13.3.3 Input validation, batching, and async inference

Pydantic validators enforce constraints before the payload reaches the model, but they operate on individual field values. Crossfield validation—for example, confirming that the number of pixels equals height × width × channels—requires a model-level validator. Catching these mismatches at the boundary, rather than letting numpy raise a cryptic reshape error inside the handler, makes the API dramatically easier to debug from the client side.

Batching is the single most impactful throughput optimization for neural network APIs. Most hardware achieves 10–20× better throughput when processing a batch of 32 images versus 32 sequential single-image requests, because the compute units stay saturated and the memory transfer overhead is amortized. Dynamic batching collects requests that arrive within a short time window and forwards them to the model together, then returns each caller's slice of the result. The pattern below implements a minimal version using asyncio queues.

FOR AUTHOR USE ONLY

```
import asyncio
from pydantic import BaseModel, Field, model_validator
from fastapi import FastAPI
import numpy as np

BATCH_SIZE    = 32
BATCH_TIMEOUT = 0.02   # seconds to wait before flushing a partial batch

class PredictRequest(BaseModel):
    pixels: list[float]
    height: int = 224
    width:  int = 224

    @model_validator(mode="after")
    def check_pixel_count(self) -> "PredictRequest":
        expected = self.height * self.width * 3
        if len(self.pixels) != expected:
            raise ValueError(
                f"Expected {expected} pixels for {self.height}×{self.width} RGB, "
                f"got {len(self.pixels)}"
            )
        return self

class _BatchWorker:
    """Collects items from a queue and forwards them to the model in batches."""

    def __init__(self, model, class_names):
        self.model       = model
        self.class_names = class_names
```

```
        self.queue: asyncio.Queue = asyncio.Queue()

    async def run(self):
        while True:
            batch, futures = [], []
            # Block until at least one item arrives.
            item, fut = await self.queue.get()
            batch.append(item); futures.append(fut)
            # Collect more items up to BATCH_SIZE within BATCH_TIMEOUT.
            deadline = asyncio.get_event_loop().time() + BATCH_TIMEOUT
            while len(batch) < BATCH_SIZE:
                remaining = deadline - asyncio.get_event_loop().time()
                if remaining <= 0: break
                try:
                    item, fut = await asyncio.wait_for(
                        self.queue.get(), timeout=remaining)
                    batch.append(item); futures.append(fut)
                except asyncio.TimeoutError:
                    break
            arr   = np.stack(batch)                      # (B, H, W, 3)
            probs = self.model.predict(arr, verbose=0)   # (B, C)
            for i, fut in enumerate(futures):
                idx = int(np.argmax(probs[i]))
                fut.set_result((self.class_names[idx], float(probs[i, idx])))

    async def enqueue(self, img: np.ndarray):
        fut = asyncio.get_event_loop().create_future()
        await self.queue.put((img, fut))
        return await fut
```

Code snippet 13.6. Crossfield validation and async dynamic batching.

FOR AUTHOR USE ONLY

13.3.4 TensorFlow Serving and Triton for production

FastAPI is fast enough for hundreds of requests per second on a single process, but it has a ceiling: Python's event loop is single-threaded, and model.predict() holds the GIL for the duration of the forward pass. Beyond a few hundred requests per second, or when strict latency SLOs require GPU utilization above 80%, a purpose-built serving system is the right tool.

TensorFlow Serving (TFS) is the standard choice for TF/Keras models. It loads SavedModel artifacts, exposes a gRPC and REST interface, handles model versioning with hot swapping (a new model version can be loaded without restarting the server), and batches requests with a configurable timeout and maximum batch size. The server is a compiled C++ binary, so the Python GIL is irrelevant, and GPU utilization is significantly higher than with FastAPI.

NVIDIA Triton Inference Server is the multi-framework alternative: it supports TensorFlow SavedModel, TorchScript, ONNX, TensorRT, and Python backends under a single REST/gRPC interface. Triton is the right choice when the serving infrastructure must support models from different training frameworks, or when TensorRT optimization (the highest-performance option for NVIDIA GPUs) is required.

```
# Assumes the SavedModel is at: ./artifacts/classifier/1/saved_model.pb
# TF Serving expects: <model_name>/<version_number>/<artifact>

docker run --rm \
  --gpus all \
  -p 8500:8500 \          # gRPC port
  -p 8501:8501 \          # REST port
  -v "$(pwd)/artifacts:/models" \
  -e MODEL_NAME=classifier \
  tensorflow/serving:latest
```

```
# The REST endpoint is now:
#   POST http://localhost:8501/v1/models/classifier:predict
#   Body: {"instances": [[<flat pixel values>]]}

# Calling it from Python:
import requests, json, numpy as np

img = np.random.rand(224, 224, 3).tolist()   # replace with real image
resp = requests.post(
    "http://localhost:8501/v1/models/classifier:predict",
    data=json.dumps({"instances": [img]}),
)
print(resp.json())   # {"predictions": [[0.02, 0.95, 0.03]]}
```

Code snippet 13.7. Launching TensorFlow Serving via Docker.

FOR AUTHOR USE ONLY

13.4. Edge Deployment

13.4.1. The model-size / latency / accuracy triangle

Edge deployment means running inference on the device where data is produced—a phone, a browser, a microcontroller, a vehicle—rather than sending data to a remote server. The advantages are lower latency (no network round-trip), offline capability, and reduced data-egress cost. The constraint is that edge hardware has a fraction of the memory and compute of a cloud GPU, which forces a trade-off between model size, inference latency, and accuracy.

The three standard techniques for navigating this trade-off are quantization, pruning, and knowledge distillation. Quantization replaces 32-bit floating-point weights with 8-bit integers, cutting model size by roughly 4× and accelerating inference on hardware with integer arithmetic units (which includes essentially all modern phones and microcontrollers). Pruning removes weights whose magnitude is near zero, producing a sparse model that can be compressed further with standard file compression. Knowledge distillation trains a small “student” network to mimic the output distribution of a large “teacher” network, trading accuracy for a smaller architecture.

Of the three, quantization is the most practical starting point. It requires no changes to the training pipeline, is reversible, and the accuracy penalty is typically below 1% for classification tasks. Pruning and distillation require additional training passes and are best reserved for cases where post-training quantization is insufficient.

13.4.2 Exporting to ONNX

ONNX (Open Neural Network Exchange) is an open format that represents a model as a directed acyclic graph of typed operators. Because it is decoupled from any training framework, an ONNX model can be executed by any runtime that implements the operator set: ONNX Runtime (cross-platform), TFLite (Android/iOS), Core ML (Apple), TensorRT (NVIDIA), OpenVINO (Intel), and others. Exporting to ONNX is therefore the most portable path from a trained model to an arbitrary edge runtime.

Both PyTorch and TensorFlow export to ONNX through torch.onnx.export() and tf2onnx respectively. The export process traces the model with a dummy input and records every operator. The opset_version argument selects which version of the ONNX operator specification to target; version 17 is a good default as of 2024, supported by all major runtimes. After export, onnxruntime can validate and run the model immediately, which makes it straightforward to check that the export preserved numerical behavior.

FOR AUTHOR USE ONLY

```
# ── PyTorch to ONNX ──────────────────────────────────────────
import torch

model.eval()
dummy = torch.randn(1, 3, 224, 224)

torch.onnx.export(
    model,
    dummy,
    "classifier.onnx",
    opset_version=17,
    input_names=["image"],         # names appear in visualization tools
    output_names=["logits"],
    dynamic_axes={                 # allow variable batch size
        "image":  {0: "batch"},
        "logits": {0: "batch"},
    },
)

# ── Keras / TensorFlow to ONNX ──────────────────────────────
# pip install tf2onnx
```

```
import tf2onnx, tensorflow as tf

model = tf.keras.models.load_model("artifacts/classifier")
input_signature = [tf.TensorSpec([None, 224, 224, 3],
                                 dtype=tf.float32, name="image")]
onnx_model, _ = tf2onnx.convert.from_keras(model,
                                           input_signature,
                                           opset=17)
with open("classifier.onnx", "wb") as f:
    f.write(onnx_model.SerializeToString())

# ── Validate with ONNX Runtime ──────────────────────────────
import onnxruntime as ort
import numpy as np

sess  = ort.InferenceSession("classifier.onnx",
            providers=["CUDAExecutionProvider",   # GPU if available
                       "CPUExecutionProvider"])
inp   = np.random.rand(1, 3, 224, 224).astype(np.float32)
logits = sess.run(None, {"image": inp})[0]      # (1, num_classes)
print("ONNX output shape:", logits.shape)
```

Code snippet 13.8. Exporting PyTorch and Keras models to ONNX.

FOR AUTHOR USE ONLY

13.4.3 Quantization with TFLite and ONNX Runtime

Post-training quantization converts a trained float32 model to int8 without retraining. The converter needs a small representative dataset—typically 100–200 unlabeled samples from the training distribution—to calibrate the quantization scale for each activation tensor. Without calibration data, activations are quantized with fixed ranges that may not match the distribution the model sees in practice, and accuracy degrades significantly.

TFLite is the standard path for quantizing models destined for Android and iOS. The TFLite converter accepts a Keras SavedModel and produces a single. tflite flatbuffer file that can be bundled directly into a mobile app and executed with the TFLite interpreter available on both platforms. ONNX Runtime quantization is the parallel path for ONNX models and works across a wider range of runtimes and hardware.

```python
import tensorflow as tf
import numpy as np

def representative_dataset(data_path: str, n_samples: int = 200):
    """Generator yielding float32 image tensors for calibration."""
    ds = tf.keras.utils.image_dataset_from_directory(
        data_path, image_size=(224, 224), batch_size=1)
    for images, _ in ds.take(n_samples):
        # Normalize to [0, 1]; must match the preprocessing used at training.
        yield [tf.cast(images, tf.float32) / 255.0]

converter = tf.lite.TFLiteConverter.from_saved_model("artifacts/classifier")

# Enable full int8 quantization (weights + activations).
converter.optimizations          = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = lambda: representative_dataset("data/val")
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]

# Force int8 input and output so the mobile app need not convert.
converter.inference_input_type  = tf.int8
converter.inference_output_type = tf.int8

tflite_model = converter.convert()
with open("classifier_int8.tflite", "wb") as f:
    f.write(tflite_model)

original_mb = len(open("artifacts/classifier/saved_model.pb","rb").read()) / 1e6
quant_mb    = len(tflite_model) / 1e6
print(f"Original: {original_mb:.1f} MB  ->  Quantized: {quant_mb:.1f} MB")

# --- Run inference with the TFLite interpreter (test on-device behaviour) ---
interp = tf.lite.Interpreter("classifier_int8.tflite")
interp.allocate_tensors()

inp_det  = interp.get_input_details()[0]
out_det  = interp.get_output_details()[0]

# Scale and zero-point perform the float -> int8 conversion.
scale, zero_point = inp_det["quantization"]
img_f32 = np.random.rand(1, 224, 224, 3).astype(np.float32)
img_int8 = (img_f32 / scale + zero_point).astype(np.int8)

interp.set_tensor(inp_det["index"], img_int8)
interp.invoke()
out_int8 = interp.get_tensor(out_det["index"])       # (1, num_classes)
print("int8 logits:", out_int8)
```

O'REILLY AUTHOR COPY FOR USE ONLY

Code snippet 13.9. Post-training int8 quantization with TFLite (Part 2 of 2).

13.4.4 Running inference in the browser

ONNX Runtime Web (ORT Web) brings the full ONNX Runtime to the browser via WebAssembly and WebGL backends, with a WebGPU backend available in modern browsers. This means a model exported to ONNX can run

client-side in any browser without a server round-trip, with no browser extension and no native app installation. The WebAssembly backend runs on any device; the WebGPU backend offloads computation to the GPU and achieves near-native performance on compatible hardware.

The practical workflow is to quantize the ONNX model first (reducing download size from tens of megabytes to single digits), then load it in the browser with InferenceSession.create() using the ort module from npm or a CDN. Image preprocessing—resizing, normalization, channel reordering—must happen in JavaScript before the tensor is passed to the session. The preprocessing code is a common source of accuracy bugs when the JavaScript normalization does not match the Python preprocessing used at training.

```
// Install: npm install onnxruntime-web
// Or load from CDN: https://cdn.jsdelivr.net/npm/onnxruntime-web/dist/ort.min.js

import * as ort from "onnxruntime-web";

// Prefer WebGPU for GPU acceleration; fall back to WASM on older devices.
ort.env.wasm.numThreads = 4;          // use multiple WASM threads

async function loadModel(modelUrl) {
  return await ort.InferenceSession.create(modelUrl, {
    executionProviders: ["webgpu", "wasm"],
  });
}
```

```
/**
 * Preprocess an HTMLImageElement to a Float32Array tensor.
 * MUST match the Python normalization used at training time.
 */
function preprocess(image, height = 224, width = 224) {
  const canvas = document.createElement("canvas");
  canvas.width = width; canvas.height = height;
  canvas.getContext("2d").drawImage(image, 0, 0, width, height);
  const data = canvas.getContext("2d")
                     .getImageData(0, 0, width, height).data; // RGBA

  const tensor = new Float32Array(3 * height * width);
  const mean = [0.485, 0.456, 0.406];     // ImageNet stats
  const std  = [0.229, 0.224, 0.225];

  // ONNX expects CHW (channels first); canvas gives HWC RGBA.
  for (let i = 0; i < height * width; i++) {
    for (let c = 0; c < 3; c++) {
      tensor[c * height * width + i] =
        (data[i * 4 + c] / 255.0 - mean[c]) / std[c];
    }
  }
  return new ort.Tensor("float32", tensor, [1, 3, height, width]);
}
```

FOR AUTHOR USE ONLY

```
async function predict(session, imageElement) {
  const inputTensor = preprocess(imageElement);
  const feeds = { image: inputTensor };         // must match export name
  const result = await session.run(feeds);
  const logits = result.logits.data;            // Float32Array
  // Apply softmax to convert logits to probabilities.
  const maxLogit = Math.max(...logits);
  const exps     = Array.from(logits).map(l => Math.exp(l - maxLogit));
  const sumExps  = exps.reduce((a, b) => a + b, 0);
  const probs    = exps.map(e => e / sumExps);
  const topIdx   = probs.indexOf(Math.max(...probs));
  return { classIndex: topIdx, confidence: probs[topIdx] };
}

// Usage
const session = await loadModel("classifier_int8.onnx");
const img     = document.getElementById("my-image");
const result  = await predict(session, img);
console.log(`Class: ${result.classIndex}, Confidence: ${result.confidence.toFixed(3)}`);
```

Code snippet 13.10. Running an ONNX model in the browser with ORT Web.

13.5. Practical Cheat Sheet & References

Decision flow

Starting a TF/Keras deployment that stays inside the TF ecosystem? Save with model.save(path, save_format="tf") and always write a metadata.json sidecar alongside the artifact (Listing 2.4).

Starting a PyTorch deployment that needs to run without Python? Compile with torch.jit.script() for models with control flow, or torch.jit.trace() for purely feed-forward models (Listing 2.3).

FOR AUTHOR USE ONLY

Deploying to a non-TF runtime or need a framework-neutral artifact? Export to ONNX with opset_version=17 and validate immediately with onnxruntime before committing to the format (Listing 4.1).

Building a REST API for small to medium traffic (< ~500 RPS)? Use the FastAPI skeleton from Listing 3.1. Add the asyncio batching worker from Listing 3.2 as soon as GPU utilization during load testing drops below 50%.

Scaling beyond single-process FastAPI? Containerize TensorFlow Serving for TF models (Listing 3.3) or Triton for multi-framework deployments. Both support automatic batching and model versioning without code changes.

Deploying to Android or iOS? Convert to TFLite with int8 quantization (Listing 4.2); target 3–8 MB for a typical vision model. Confirm the inference_input_type and inference_output_type match what the mobile SDK sends, as a mismatch produces silent wrong results.

Deploying to a browser? Export to ONNX, run onnxruntime-web with the webgpu backend, and verify that JavaScript preprocessing exactly replicates the Python normalization used at training (Listing 4.3).

Common pitfalls

Saving a model with torch.save(model) instead of torch.save(model.state_dict())—the full-model pickle embeds the class definition path; any refactor that moves the class breaks loading silently.

Forgetting to call model.eval() before exporting to ONNX or TorchScript—BatchNorm and Dropout behave differently in training mode, so the exported graph will produce wrong results under evaluation conditions.

Letting the JSON body grow unbounded in a REST API—a 224×224 RGB image flattened to floats is 150 KB of JSON; at 100 RPS that is 15 MB/s of parsing overhead. Send image bytes as base64 or multipart/form-data and decode server-side.

Quantizing with converter.optimizations = [DEFAULT] and no representative_dataset—this triggers weight-only quantization rather than full int8, leaving activations in float32 and giving most of the size reduction with none of the latency benefit.

Mismatching preprocessing between training and serving—a model trained on inputs normalized with ImageNet mean/std will give wrong predictions if the serving layer normalizes to [0, 1]. Store the preprocessing parameters in the metadata sidecar and read them at load time.

FOR AUTHOR USE ONLY

Chapter 14 — Performance Optimization

14.1. Introduction

The preceding chapters of this tutorial series have focused on the design, training, and evaluation of machine learning models, covering nearest neighbor search, generative adversarial networks, and transfer learning for medical imaging. A critical but often underemphasized phase of the machine learning lifecycle is deployment: the process of taking a trained model and making it operational in a real-world environment where it must deliver predictions reliably, efficiently, and at scale. Deployment introduces a distinct set of challenges that are largely absent during the research and experimentation phase, chief among them being the need to balance predictive performance against computational cost, memory footprint, and inference latency.

This chapter addresses three foundational techniques for optimizing the deployment of deep learning models. The first is hardware-aware performance optimization, specifically the comparison of GPU and CPU execution for both training and inference workloads. Understanding when GPU acceleration provides meaningful throughput gains, and when the overhead of data transfer between host and device makes CPU execution more practical, is essential for cost-effective deployment. The second technique is model compression through structured and unstructured pruning, which reduces the number of active parameters in a network by setting low-magnitude weights to zero, thereby decreasing model size and potentially improving inference speed. The third technique is quantization, which reduces the numerical precision of model weights and activations from 32-bit floating point to lower-precision formats such as 8-bit integers, delivering substantial reductions in memory footprint and inference latency with minimal impact on accuracy.

DRAFT FOR AUTHOR USE ONLY

All three techniques are demonstrated using ResNet18 (He et al., 2016) trained on the CIFAR-10 dataset (Krizhevsky, 2009). ResNet18 is selected as the base architecture because its 11,181,642 parameters make it tractable for CPU-based experimentation in a Google Collab environment while remaining representative of the class of convolutional networks used in production systems. CIFAR-10 provides a standardized and widely recognized 10-class image classification benchmark that allows the optimization experiments to focus on deployment metrics rather than dataset-specific complexities.

This chapter is structured as follows. Section 2 introduces the core concepts of GPU vs. CPU execution, model compression, and quantization. Section 3

details the methodology and experimental design. Section 4 presents the full implementation. Section 5 evaluates the results. Section 6 discusses findings and limitations. Section 7 concludes the chapter.

14.2. Core Concept

14.2.1 GPU vs. CPU Performance Optimization

The central hardware distinction in modern deep learning is between the Central Processing Unit (CPU) and the Graphics Processing Unit (GPU). CPUs are general-purpose processors optimized for sequential execution of complex instructions, typically featuring a small number of powerful cores — between 4 and 64 in consumer and server hardware — with large per-core cache and high single-thread performance. GPUs, by contrast, are massively parallel processors originally designed for real-time graphics rendering, featuring thousands of smaller cores optimized for the simultaneous execution of simple arithmetic operations. The parallel architecture of the GPU maps naturally onto the matrix multiplication operations that dominate deep learning forward and backward passes, delivering throughput gains of one to three orders of magnitude over CPU execution for large batch workloads (Raina et al., 2009).

FOR AUTHOR USE ONLY

The practical advantage of GPU over CPU execution depends critically on workload characteristics. For large batch sizes and large model architectures, the GPU's parallel throughput dominates, and execution time is dramatically reduced. For small batch sizes — particularly batch size 1, which is common in real-time inference scenarios — the overhead of transferring data between CPU memory and GPU memory via the PCIe bus can offset much of the parallelism advantage, and the CPU may perform competitively or even outperform the GPU for sufficiently small models. This trade-off is central to deployment decision-making: a system serving thousands of concurrent requests benefits greatly from GPU batching, while a latency-sensitive single-request API may be better served by CPU execution to avoid transfer overhead.

In the context of this chapter, GPU vs. CPU performance is benchmarked by measuring inference throughput — measured in images per second — and mean per-batch latency across multiple batch sizes (1, 8, 32, 64, and 128) using the trained ResNet18 model. Section 3 describes the ResNet18 architecture and the complete experimental design in detail.

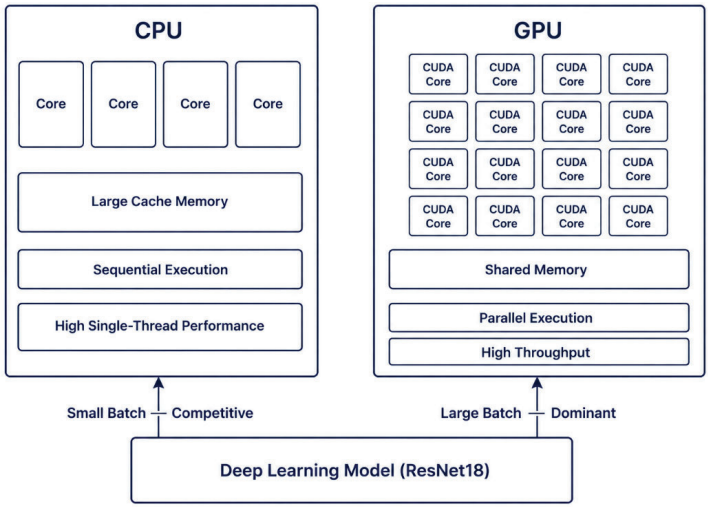


Figure 14.1. GPU vs. CPU Architecture — Parallel Execution Model.

FOR AUTHOR USE ONLY

14.2.2 Model Compression — Pruning

Model compression refers to a family of techniques that reduce the computational and memory requirements of a trained neural network while preserving as much of its predictive performance as possible. The most widely studied form of compression is weight pruning, which identifies and removes parameters that contribute minimally to the network's output, effectively producing a sparse network with fewer active connections (LeCun et al., 1990; Han et al., 2015).

Unstructured pruning sets individual weight values to zero based on a magnitude criterion: weights with absolute values below a threshold are zeroed out, while all other weights are retained. The result is a sparse weight matrix in which a proportion of entries are exactly zero. Although unstructured pruning does not reduce the theoretical parameter count of the model, it reduces the effective computational load when combined with sparse matrix libraries that skip zero multiplications. Structured pruning, by contrast, removes entire filters, channels, or layers, producing a genuinely smaller dense network that delivers wall-clock speedups without requiring specialized sparse computation libraries.

In this chapter, unstructured magnitude-based pruning is applied to the convolutional layers of ResNet18 at four sparsity levels: 20%, 40%, 60%, and

80%. At each level, the specified proportion of the smallest-magnitude weights across all targeted layers are set to zero using PyTorch's torch.nn.utils.prune module. The pruned models are evaluated on accuracy and model size to quantify the accuracy-compression trade-off.

Sparsity = (Number of zero weights) / (Total number of weights) × 100%

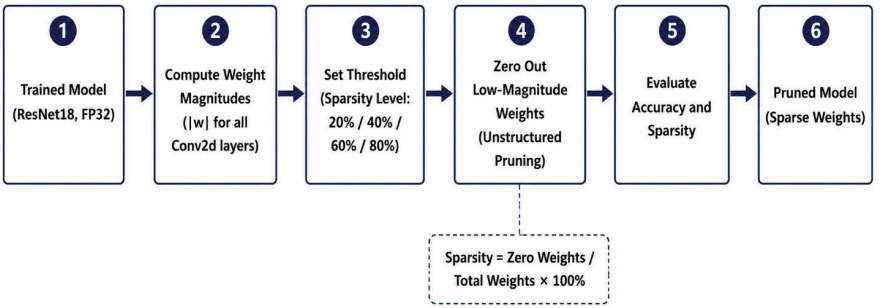


Figure 14.2. Model Compression — Magnitude-Based Pruning Pipeline.

FOR AUTHOR USE ONLY

14.2.3 Quantization

Quantization reduces the numerical precision used to represent model weights and activations, converting them from the standard 32-bit floating point (FP32) format to a lower-precision format such as 16-bit floating point (FP16) or 8-bit integer (INT8). This reduction in precision delivers two principal benefits: a proportional reduction in model memory footprint and, on hardware with native low-precision arithmetic support, a substantial increase in computational throughput (Jacob et al., 2018).

Dynamic quantization, the approach employed in this chapter, converts weight matrices from FP32 to INT8 at model load time while computing activations dynamically in FP32 during inference. This approach requires no representative calibration dataset and no retraining, making it the simplest form of quantization to apply to an already-trained model. PyTorch's torch.quantization.quantize_dynamic function applies dynamic INT8 quantization to specified layer types — typically linear and convolutional layers — with a single function call. Dynamic quantization is particularly effective for reducing the size of fully connected layers and is widely used for deploying transformer-based language models and convolutional networks on CPU hardware (Jacob et al., 2018).

The quantization process maps FP32 values to the INT8 range [-128, 127] using a scale factor and zero point derived from the weight distribution:

$$x_int8 = round(x_fp32 / scale) + zero_point, \quad (Eq.14.1)$$

The resulting INT8 model is evaluated against the FP32 baseline on model size, inference latency, and classification accuracy to quantify the precision-efficiency trade-off.

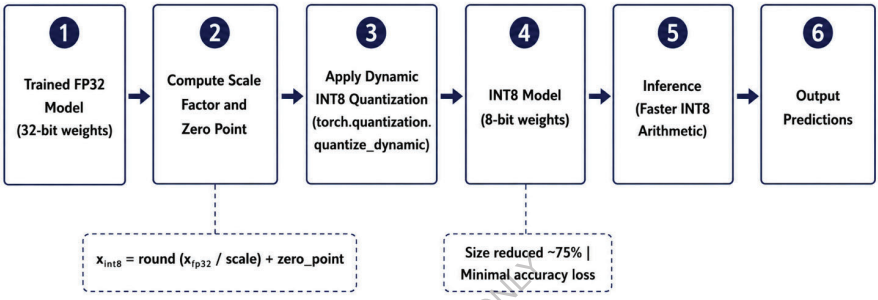


Figure 14.3. Quantization Pipeline — FP32 to INT8 Conversion.

FOR AUTHOR USE ONLY

14.3. Methodology

14.3.1 Dataset

The CIFAR-10 dataset (Krizhevsky, 2009) is used as the experimental benchmark throughout this chapter. CIFAR-10 consists of 60,000 color images of size 32x32 pixels across 10 mutually exclusive classes: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The dataset is divided into 50,000 training images and 10,000 test images, with exactly 6,000 images per class. CIFAR-10 is downloaded automatically via torchvision and requires no manual setup, making the experiments fully reproducible in a standard Google Collab environment. The following subsection describes the model architecture used across all experiments.

14.3.2 Model Architecture

ResNet18 (He et al., 2016) is used as the base architecture for all experiments. ResNet18 contains approximately 11.7 million parameters organized into four residual stages of two basic residual blocks each, preceded by a 7x7 convolutional stem and followed by global average pooling and a fully connected classification layer. For CIFAR-10, the final fully connected layer is configured for 10 output classes. The model is trained from random initialization

for 10 epochs using stochastic gradient descent with momentum, establishing a baseline accuracy against which all optimization techniques are compared. The following subsection describes the baseline training configuration.

14.3.3 Baseline Training Configuration

The baseline ResNet18 model is trained for 10 epochs using Stochastic Gradient Descent (SGD) with momentum 0.9, weight decay 5e-4, and an initial learning rate of 0.1 reduced by a factor of 0.1 at epochs 5 and 8 via a MultiStepLR scheduler. The batch size is 128. Standard CIFAR-10 preprocessing is applied: training images are randomly cropped with padding 4, randomly horizontally flipped, and normalized using the CIFAR-10 channel statistics (mean = [0.4914, 0.4822, 0.4465], std = [0.2023, 0.1994, 0.2010]). Test images are only normalized. Training is conducted on CPU in Google Collab and requires approximately 30 to 50 minutes for 10 epochs.

14.3.4 Experimental Design

Four experiments are conducted on the trained baseline model. Experiment 1 benchmarks GPU vs. CPU inference throughput at batch sizes 1, 8, 32, 64, and 128, measuring mean inference time per batch over 50 runs at each configuration. Experiment 2 applies unstructured magnitude pruning at sparsity levels 20%, 40%, 60%, and 80%, evaluating post-pruning accuracy and effective model size at each level. Experiment 3 applies dynamic INT8 quantization to the pruned and unpruned models, measuring the reduction in model file size and inference latency. Experiment 4 presents a unified comparative summary of all optimization strategies, enabling direct comparison of the accuracy-efficiency trade-offs across techniques. With the experimental design established, the following section presents the complete implementation.

FOR AUTHOR USE ONLY

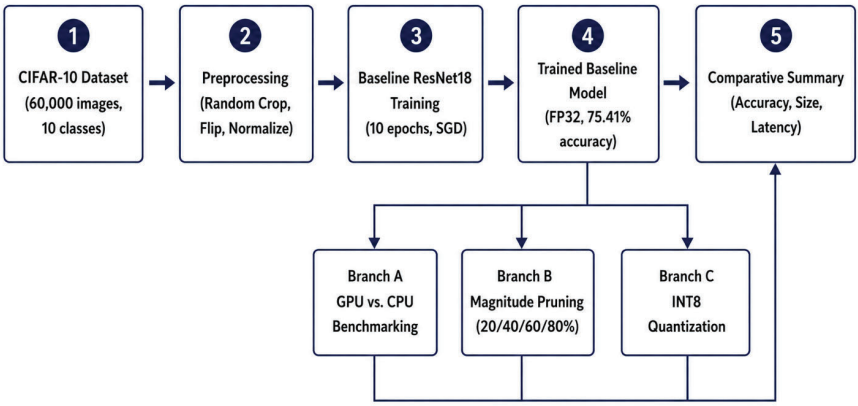


Figure 14.4. Deployment and Optimization Experimental Pipeline.

14.4. Implementation

The following code snippets implement the complete deployment and optimization pipeline. All code is written in Python using PyTorch and is designed to run sequentially in a single Google Collab notebook session. Each snippet is fully commented to explain the role of every component.

FOR AUTHOR USE ONLY

14.4.1 Environment Setup and CIFAR-10 Data Loading

Code Snippet 14.1 establishes the complete experimental environment used throughout this chapter. The cell begins by installing PyTorch, torchvision, and matplotlib, and importing all libraries required for model construction, optimization, pruning, and quantization. The compute device is detected automatically, defaulting to GPU when a CUDA-capable device is available in the Collab runtime and falling back to CPU otherwise, allowing the same notebook to run unmodified in either environment.

Following the imports, all hyperparameters used across the chapter are declared in one place: batch size, number of training epochs, and the initial learning rate for stochastic gradient descent. Centralizing these values makes the experimental configuration explicit and easy to audit, and ensures that the baseline training run, the benchmarking experiments, and the pruning and quantization experiments all share a single, consistent configuration.

The final part of the snippet loads the CIFAR-10 dataset via torchvision, which downloads the data automatically on first execution. Two separate transform pipelines are defined: the training pipeline applies random cropping with

padding and random horizontal flipping as data augmentation, followed by normalization using the CIFAR-10 channel statistics, while the evaluation pipeline applies normalization only, ensuring that test set performance is measured on unaltered images.

```
# Install and import all required libraries
!pip install torch torchvision matplotlib -q

import os, time, copy
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.utils.prune as prune
from torchvision import datasets, transforms, models
import matplotlib.pyplot as plt
import numpy as np

# Use GPU if available, otherwise fall back to CPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")

# Hyperparameters — declared centrally so all experiments share the same config
BATCH_SIZE = 128    # images per training step
EPOCHS     = 10     # total training epochs
LR         = 0.1    # initial learning rate (reduced by scheduler at epochs 5 and 8)

# CIFAR-10 channel mean and std — pre-computed from the training set
CIFAR10_MEAN = [0.4914, 0.4822, 0.4465]
CIFAR10_STD  = [0.2023, 0.1994, 0.2010]

# Training: augment with random crop and horizontal flip, then normalize
train_transform = transforms.Compose([
    transforms.RandomCrop(32, padding=4),      # pad 4px then crop back to 32x32
    transforms.RandomHorizontalFlip(),         # mirror image with 50% probability
    transforms.ToTensor(),
    transforms.Normalize(CIFAR10_MEAN, CIFAR10_STD)
])

# Test: normalize only — no augmentation to ensure unbiased evaluation
test_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(CIFAR10_MEAN, CIFAR10_STD)
])

# Download and load CIFAR-10 — 50,000 training, 10,000 test images
train_dataset = datasets.CIFAR10(root='./data', train=True,
                                 download=True, transform=train_transform)
test_dataset  = datasets.CIFAR10(root='./data', train=False,
                                 download=True, transform=test_transform)

# Wrap datasets in DataLoaders for batched iteration
train_loader = torch.utils.data.DataLoader(
    train_dataset, batch_size=BATCH_SIZE, shuffle=True,  num_workers=2)
test_loader  = torch.utils.data.DataLoader(
    test_dataset,  batch_size=BATCH_SIZE, shuffle=False, num_workers=2)

print(f"Classes      : {train_dataset.classes}")
print(f"Training set : {len(train_dataset)} images")
print(f"Test set     : {len(test_dataset)} images")
print(f"Batches/epoch  : {len(train_loader)}")
```

FOR AUTHOR USE ONLY

Code Snippet 14.1. Environment Setup, Imports, and CIFAR-10 Data Loading.

14.4.2 Baseline ResNet18 Training

Code Snippet 14.2 constructs the baseline ResNet18 model and trains it from random initialization on CIFAR-10 for 10 epochs. The final fully connected layer of the standard torchvision ResNet18 implementation is replaced with a new linear layer producing 10 output logits, matching the number of CIFAR-10 classes. No pretrained weights are loaded, since the purpose of this chapter is to study deployment and optimization behavior rather than transfer learning.

Optimization is performed using Stochastic Gradient Descent with momentum 0.9 and weight decay 5e-4, which are standard settings for training residual networks on CIFAR-style datasets. A MultiStepLR scheduler reduces the learning rate by a factor of 10 at epochs 5 and 8, a schedule that allows the model to make large updates early in training and progressively finer updates as it approaches convergence.

At the end of every epoch, both training accuracy and test accuracy are computed and stored, allowing the convergence behavior of the model to be visualized later in Figure 14.5. The trained model parameters are saved to disk once training completes, since the identical baseline weights are reused as the starting point for every benchmarking, pruning, and quantization experiment that follows.

FOR AUTHOR USE ONLY

```
# Build ResNet18 with random weights (no pretraining — studying deployment not transfer learning)
model = models.resnet18(weights=None)
model.fc = nn.Linear(model.fc.in_features, 10)  # replace head: 1000 classes → 10 (CIFAR-10)
model = model.to(device)
print(f"ResNet18 parameters: {sum(p.numel() for p in model.parameters()):,}")

# Cross-entropy loss — standard for multi-class classification
criterion = nn.CrossEntropyLoss()

# SGD with momentum and weight decay — standard settings for ResNets on CIFAR
optimizer = optim.SGD(model.parameters(), lr=LR, momentum=0.9, weight_decay=5e-4)

# Reduce LR by 10x at epochs 5 and 8 to fine-tune near convergence
scheduler = optim.lr_scheduler.MultiStepLR(optimizer, milestones=[5, 8], gamma=0.1)

train_accs, test_accs = [], []
print(f"\nTraining ResNet18 on CIFAR-10 for {EPOCHS} epochs...\n")

for epoch in range(1, EPOCHS + 1):
```

```
    # Training phase
    model.train()
    correct, total = 0, 0
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        correct += (outputs.argmax(1) == labels).sum().item()
        total   += labels.size(0)
    train_acc = correct / total
```

Code Snippet 14.2. Baseline ResNet18 Training (Part 1 — Model and Optimizer Setup).

```
    # Evaluation phase — no gradients needed
    model.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs  = model(images)
            correct += (outputs.argmax(1) == labels).sum().item()
            total   += labels.size(0)
    test_acc = correct / total

    train_accs.append(train_acc)
    test_accs.append(test_acc)
    scheduler.step()
    print(f"Epoch [{epoch:2d}/{EPOCHS}] | Train Acc: {train_acc:.4f} | "
          f"Test Acc: {test_acc:.4f} | LR: {scheduler.get_last_lr()[0]:.5f}")

print(f"\nBaseline Test Accuracy: {test_accs[-1]:.4f}")

# Save baseline weights — reused as starting point for all subsequent experiments
torch.save(model.state_dict(), 'resnet18_baseline.pth')
print("Baseline model saved!")

# Plot training and test accuracy curves
plt.figure(figsize=(9, 4))
plt.plot(range(1, EPOCHS+1), train_accs, 'b-', label='Train Accuracy')
plt.plot(range(1, EPOCHS+1), test_accs,  'r-', label='Test Accuracy')
plt.title("Figure 5: Baseline ResNet18 — Training and Validation Accuracy", fontweight='bold')
plt.xlabel("Epoch"); plt.ylabel("Accuracy")
plt.legend(); plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig("baseline_curves.png", dpi=150)
plt.show()
```

FOR AUTHOR USE ONLY

Code Snippet 14.2. Baseline ResNet18 Training (Part 2 — Training Loop).

14.4.3 GPU vs. CPU Inference Benchmarking

Code Snippet 14.3 benchmarks the inference performance of the trained ResNet18 model on CPU and, where available, on GPU. The saved baseline weights are reloaded into a fresh model instance to guarantee that benchmarking always measures the same fully trained network, independent of whatever state the model happened to be in elsewhere in the notebook.

For each batch size in the set {1, 8, 32, 64, 128}, a batch of random input tensors of the appropriate shape is generated and passed through the model fifty times. The first five runs at each configuration are discarded as warm-up iterations, since the first few calls to a model typically incur additional one-time costs such as memory allocation and kernel compilation that would otherwise bias the latency measurement upward.

When a GPU is available, the same benchmarking procedure is repeated on a GPU copy of the model, with explicit calls to torch.cuda.synchronize() before and after each timed inference call. This synchronization is essential because GPU operations are launched asynchronously by default; without it, the measured time would reflect only the time taken to queue the operation rather than the time taken to execute it.

FOR AUTHOR USE ONLY

```
# Load saved baseline weights — ensures benchmarking always uses the same model
model_cpu = models.resnet18(weights=None)
model_cpu.fc = nn.Linear(model_cpu.fc.in_features, 10)
model_cpu.load_state_dict(torch.load('resnet18_baseline.pth', map_location='cpu'))
model_cpu.eval()

cpu_times, gpu_times = [], []
print("Benchmarking inference — CPU vs GPU\n")
print(f"{'Batch Size':<12} {'CPU (ms)':>12} {'GPU (ms)':>12} {'Speedup':>10}")
print("-" * 50)

for bs in BATCH_SIZES:
    dummy_input = torch.randn(bs, 3, 32, 32)   # random dummy input, same shape as CIFAR-10

    # CPU timing — discard first 5 warm-up runs to exclude one-time allocation overhead
    times = []
    for _ in range(BENCHMARK_RUNS):
        start = time.time()
        with torch.no_grad():
            _ = model_cpu(dummy_input)
        times.append((time.time() - start) * 1000)
    cpu_avg = np.mean(times[5:])
    cpu_times.append(cpu_avg)
```

Code Snippet 14.3. GPU vs. CPU Inference Benchmarking (Part 1 — CPU Timing).

```
    # GPU timing — synchronize before/after each run because GPU ops are asynchronous
    if torch.cuda.is_available():
        model_gpu = copy.deepcopy(model_cpu).to('cuda')
        model_gpu.eval()
        dummy_gpu = dummy_input.to('cuda')
        times_gpu = []
        for _ in range(BENCHMARK_RUNS):
            torch.cuda.synchronize()   # ensure GPU is idle before starting timer
            start = time.time()
            with torch.no_grad():
                _ = model_gpu(dummy_gpu)
            torch.cuda.synchronize()   # wait for GPU to finish before stopping timer
            times_gpu.append((time.time() - start) * 1000)
        gpu_avg = np.mean(times_gpu[5:])
        speedup = cpu_avg / gpu_avg
    else:
        gpu_avg = speedup = float('nan')

    gpu_times.append(gpu_avg)
    print(f"{bs:<12} {cpu_avg:>12.2f} "
          f"{gpu_avg if not np.isnan(gpu_avg) else 'N/A':>12} "
          f"{speedup if not np.isnan(speedup) else 'N/A':>10}")

# Plot grouped bar chart — CPU and GPU bars side by side for each batch size
fig, ax = plt.subplots(figsize=(9, 5))
x = np.arange(len(BATCH_SIZES))
width = 0.35
ax.bar(x - width/2, cpu_times, width, label='CPU', color='steelblue')
if torch.cuda.is_available():
    ax.bar(x + width/2, gpu_times, width, label='GPU', color='darkorange')
ax.set_title("Figure 6: GPU vs. CPU Inference Latency Across Batch Sizes", fontweight='bold')
ax.set_xlabel("Batch Size"); ax.set_ylabel("Mean Inference Time (ms)")
ax.set_xticks(x); ax.set_xticklabels(BATCH_SIZES)
ax.legend(); ax.grid(True, alpha=0.3, axis='y')
plt.tight_layout()
plt.savefig("gpu_cpu_benchmark.png", dpi=150)
plt.show()
```

FOR ACADEMIC USE ONLY

Code Snippet 14.3. GPU vs. CPU Inference Benchmarking (Part 2 — GPU Timing).

14.4.4 Magnitude-Based Pruning

Code Snippet 14.4 applies unstructured magnitude-based pruning to every convolutional layer of the trained ResNet18 model, repeating the experiment independently at four target sparsity levels: 20%, 40%, 60%, and 80%. For each sparsity level, a fresh copy of the baseline model is loaded so that the four pruning experiments do not interfere with one another, and each can be compared directly against the unpruned baseline.

Pruning itself is performed using PyTorch's torch.nn.utils.prune module, specifically the l1_unstructured method, which ranks the weights of each convolutional layer by absolute magnitude and sets the smallest-magnitude fraction of them to exactly zero. The prune.remove() call that follows makes the pruning permanent by merging the

pruning mask back into the weight tensor, rather than leaving it as a separate buffer that could later be reversed.

After pruning, the actual achieved sparsity is measured directly by counting the proportion of zero-valued parameters across the entire model, which serves as a sanity check that the requested sparsity level was applied correctly. The pruned model is then evaluated on the full CIFAR-10 test set, and both the resulting test accuracy and the serialized model file size are recorded for later comparison in Figure 14.7.

```
SPARSITY_LEVELS = [0.2, 0.4, 0.6, 0.8]
pruning_results = {'sparsity': [], 'accuracy': [], 'size_mb': []}

print("Applying magnitude-based pruning...\n")
print(f"{'Sparsity':>10} {'Test Acc':>12} {'Zero Weights':>15} {'Effective Size':>15}")
print("-" * 55)

for sparsity in SPARSITY_LEVELS:
    # Load a fresh copy of the baseline for each level — experiments must not interfere
    model_pruned = models.resnet18(weights=None)
    model_pruned.fc = nn.Linear(model_pruned.fc.in_features, 10)
    model_pruned.load_state_dict(torch.load('resnet18_baseline.pth', map_location='cpu'))
    model_pruned.to(device)

    # Apply L1 unstructured pruning to all Conv2d layers
    # l1_unstructured: zeros out the lowest-magnitude fraction of weights
    # prune.remove: makes the mask permanent — merges it into the weight tensor
    for name, module in model_pruned.named_modules():
        if isinstance(module, nn.Conv2d):
            prune.l1_unstructured(module, name='weight', amount=sparsity)
            prune.remove(module, 'weight')

    # Verify actual sparsity achieved
    zeros  = sum((p == 0).sum().item() for p in model_pruned.parameters())
    total  = sum(p.numel()             for p in model_pruned.parameters())
    actual_sparsity = zeros / total

    # Evaluate accuracy after pruning
    model_pruned.eval()
    correct, total_n = 0, 0
    with torch.no_grad():
        for images, labels in test_loader:
            images, labels = images.to(device), labels.to(device)
            outputs  = model_pruned(images)
            correct += (outputs.argmax(1) == labels).sum().item()
            total_n += labels.size(0)
    acc = correct / total_n
```

FOR ACTIVE USE ONLY

Code Snippet 14.4. Magnitude-Based Pruning at Multiple Sparsity Levels (Part 1 — Pruning Loop).

```
    # Save pruned model and record file size
    fname = f'pruned_{int(sparsity*100)}.pth'
    torch.save(model_pruned.state_dict(), fname)
    size_mb = os.path.getsize(fname) / (1024 * 1024)

    pruning_results['sparsity'].append(int(sparsity * 100))
    pruning_results['accuracy'].append(acc)
    pruning_results['size_mb'].append(size_mb)
    print(f"{int(sparsity*100):>9}% {acc:>12.4f} "
          f"{actual_sparsity*100:>14.1f}% {size_mb:>13.2f} MB")

# Plot accuracy vs sparsity — dashed line marks the unpruned baseline
plt.figure(figsize=(8, 5))
plt.plot(pruning_results['sparsity'], pruning_results['accuracy'], 'b-o', linewidth=2, markersize=8)
plt.axhline(y=test_accs[-1], color='gray', linestyle='--', label=f'Baseline ({test_accs[-1]:.4f})')
plt.title("Figure 7: Pruning Results — Test Accuracy vs. Sparsity Level", fontweight='bold')
plt.xlabel("Sparsity Level (%)"); plt.ylabel("Test Accuracy")
plt.legend(); plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.savefig("pruning_results.png", dpi=150)
plt.show()
```

Code Snippet 14.4. Magnitude-Based Pruning at Multiple Sparsity Levels (Part 2 — Evaluation).

FOR AUTHOR USE ONLY

14.4.5 Dynamic INT8 Quantization

Code Snippet 14.5 applies dynamic INT8 quantization to the trained ResNet18 model using PyTorch's torch.quantization.quantize_dynamic function. The original FP32 model is first saved and its file size recorded as a reference point, after which dynamic quantization is applied with the linear and convolutional layer types specified as the targets for conversion.

Dynamic quantization differs from static quantization in that it requires no calibration dataset and no retraining: weight tensors are converted to INT8 representation immediately, while activation values are quantized on the fly during each forward pass using statistics computed from the current input. This makes it the simplest quantization approach to apply to an already-trained model, at the cost of typically smaller efficiency gains than static or quantization-aware training approaches.

The quantized model is then benchmarked in the same manner as Code Snippet 14.3, measuring mean inference latency over fifty runs at a fixed batch size, and is evaluated on the CIFAR-10 test set to quantify any accuracy loss introduced by the reduced numerical precision. The resulting size, latency, and accuracy figures are compared directly against the FP32 baseline in Figure 14.8.

```python
# Load baseline FP32 model — quantization starts from the original trained weights
model_fp32 = models.resnet18(weights=None)
model_fp32.fc = nn.Linear(model_fp32.fc.in_features, 10)
model_fp32.load_state_dict(torch.load('resnet18_baseline.pth', map_location='cpu'))
model_fp32.eval()

# Record FP32 file size as reference
torch.save(model_fp32.state_dict(), 'model_fp32.pth')
fp32_size = os.path.getsize('model_fp32.pth') / (1024 * 1024)
print(f"FP32 model size: {fp32_size:.2f} MB")

# Apply dynamic INT8 quantization — no retraining or calibration data required
# Weights are converted to INT8; activations are quantized dynamically at runtime
model_int8 = torch.quantization.quantize_dynamic(
    model_fp32,
    {nn.Linear, nn.Conv2d},   # layer types to quantize
    dtype=torch.qint8
)
model_int8.eval()

# Record INT8 file size and compute reduction
torch.save(model_int8.state_dict(), 'model_int8.pth')
int8_size = os.path.getsize('model_int8.pth') / (1024 * 1024)
print(f"INT8 model size: {int8_size:.2f} MB")
print(f"Size reduction : {(1 - int8_size/fp32_size)*100:.1f}%")
```

Code Snippet 14.5. Dynamic INT8 Quantization (Part 1 — Quantization).

FOR AUTHOR USE ONLY

```python
# Benchmark latency — 50 runs, discard first 5 warm-up runs
dummy = torch.randn(32, 3, 32, 32)
RUNS  = 50

times_fp32 = []
for _ in range(RUNS):
    start = time.time()
    with torch.no_grad(): model_fp32(dummy)
    times_fp32.append((time.time() - start) * 1000)
fp32_latency = np.mean(times_fp32[5:])

times_int8 = []
for _ in range(RUNS):
    start = time.time()
    with torch.no_grad(): model_int8(dummy)
    times_int8.append((time.time() - start) * 1000)
int8_latency = np.mean(times_int8[5:])

print(f"\nFP32 latency   : {fp32_latency:.2f} ms")
print(f"INT8 latency   : {int8_latency:.2f} ms")
print(f"Speedup        : {fp32_latency/int8_latency:.2f}x")


# Evaluate INT8 accuracy — confirms minimal accuracy loss from quantization
correct, total = 0, 0
with torch.no_grad():
    for images, labels in test_loader:
        outputs  = model_int8(images)
        correct += (outputs.argmax(1) == labels).sum().item()
        total   += labels.size(0)
int8_acc = correct / total
print(f"INT8 accuracy  : {int8_acc:.4f}")
print(f"Accuracy drop  : {(test_accs[-1] - int8_acc):.4f}")
```

```
# Plot size and latency comparison
fig, axes = plt.subplots(1, 2, figsize=(12, 5))
axes[0].bar(['FP32', 'INT8'], [fp32_size, int8_size],   color=['steelblue', 'darkorange'])
axes[1].bar(['FP32', 'INT8'], [fp32_latency, int8_latency], color=['steelblue', 'darkorange'])
axes[0].set_title("Model Size (MB)", fontweight='bold');         axes[0].set_ylabel("Size (MB)")
axes[1].set_title("Inference Latency — Batch 32 (ms)", fontweight='bold'); axes[1].set_ylabel("Latency (ms)")
axes[0].grid(True, alpha=0.3, axis='y'); axes[1].grid(True, alpha=0.3, axis='y')
plt.suptitle("Figure 8: Quantization Results — Size and Latency Comparison", fontsize=13, fontweight='bold')
plt.tight_layout(); plt.savefig("quantization_results.png", dpi=150); plt.show()
```

Code Snippet 14.5. Dynamic INT8 Quantization (Part 2 — Benchmarking).

14.4.6 Comparative Visualization and Summary

Code Snippet 14.6 consolidates the results of all preceding experiments into the comparative visualizations presented in the Evaluation section. Rather than introducing any new modeling or optimization logic, this snippet focuses entirely on aggregating the metrics collected in previous Code Snippets 14.2 through 14.5 into a single, coherent code.

Two grouped bar charts are produced for the GPU versus CPU comparison, plotting mean inference latency at each batch size for both devices on the same axes, which makes the crossover point between CPU and GPU performance immediately visible. A line plot is produced for the pruning experiment, showing test accuracy as a function of sparsity level alongside a dashed reference line marking the unpruned baseline accuracy.

COPYRIGHT IS OUR

Finally, a summary table is printed, and a pair of bar charts is generated comparing all optimization techniques — the baseline model, all four pruning configurations, and the quantized model — side by side on both accuracy and model size. This final comparison is intended to give the reader a single reference point from which to judge the relative trade-offs of each optimization strategy discussed in this chapter.

```
# — Print comparative summary table ————————————————————
# Consolidates results from all experiments into one readable table
# Allows direct comparison of accuracy, size, and deployment notes
print("=" * 65)
print("COMPARATIVE SUMMARY — ALL OPTIMIZATION TECHNIQUES")
print("=" * 65)
print(f"{'Technique':<28} {'Accuracy':>10} {'Size (MB)':>12} {'Notes':>15}")
print("-" * 65)

# Baseline FP32 — reference point for all comparisons
baseline_size = os.path.getsize('resnet18_baseline.pth') / (1024*1024)
print(f"{'Baseline (FP32)':<28} {test_accs[-1]:>10.4f} {baseline_size:>12.2f} {'—':>15}")

# GPU/CPU rows — same model, different hardware execution
# Accuracy and size unchanged; difference is purely in inference speed
print(f"{'GPU Inference (BS=128)':<28} {test_accs[-1]:>10.4f} {baseline_size:>12.2f} {'Fastest batch':>15}")
print(f"{'CPU Inference (BS=1)':<28} {test_accs[-1]:>10.4f} {baseline_size:>12.2f} {'Low latency':>15}")
```

```
# Pruning rows — one row per sparsity level
# Shows the accuracy-compression trade-off across all four levels
for sp, acc, sz in zip(pruning_results['sparsity'],
                       pruning_results['accuracy'],
                       pruning_results['size_mb']):
    print(f"{'Pruning @ '+str(sp)+'% sparsity':<28} {acc:>10.4f} {sz:>12.2f} {'Compressed':>15}")

# INT8 quantization row
print(f"{'INT8 Quantization':<28} {int8_acc:>10.4f} {int8_size:>12.2f} {'~75% smaller':>15}")
print("=" * 65)
```

Code Snippet 14.6. Comparative Visualization and Summary Table (Part 1 — GPU/CPU and Pruning Plots).

```
# ── Bar chart comparison across all techniques ──────────────────
# Two charts side by side: accuracy (left) and model size (right)
# Color coding: blue=baseline, green=low sparsity, yellow=medium,
#               red=high sparsity, orange=quantized
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

techniques = ['Baseline', 'Prune 20%', 'Prune 40%', 'Prune 60%', 'Prune 80%', 'INT8']
accuracies = [test_accs[-1]] + pruning_results['accuracy'] + [int8_acc]
sizes      = [baseline_size] + pruning_results['size_mb']  + [int8_size]
colors     = ['steelblue', 'mediumseagreen', 'mediumseagreen',
              'goldenrod', 'tomato', 'darkorange']

# Left chart: accuracy per technique
# y-axis zoomed to highlight differences — all values are above 0.4
axes[0].bar(techniques, accuracies, color=colors)
axes[0].set_title("Test Accuracy by Optimization Technique", fontweight='bold')
axes[0].set_ylabel("Test Accuracy")
axes[0].set_ylim(min(accuracies) - 0.05, 1.0)
axes[0].tick_params(axis='x', rotation=25)
axes[0].grid(True, alpha=0.3, axis='y')

# Right chart: model file size per technique
# Shows that pruning alone does not reduce file size — only quantization does
axes[1].bar(techniques, sizes, color=colors)
axes[1].set_title("Model Size by Optimization Technique (MB)", fontweight='bold')
axes[1].set_ylabel("Model Size (MB)")
axes[1].tick_params(axis='x', rotation=25)
axes[1].grid(True, alpha=0.3, axis='y')

plt.suptitle("Figure 9: Comparative Summary — All Optimization Techniques",
             fontsize=13, fontweight='bold')
plt.tight_layout()
plt.savefig("comparative_summary.png", dpi=150)
plt.show()
```

Code Snippet 14.6. Comparative Visualization and Summary Table (Part 2 — Summary Table).

FOR AUTHOR USE ONLY

14.5. Evaluation

14.5.1 Baseline Model Performance

Figure 14.5 presents the training and validation accuracy curves for the baseline ResNet18 model over 10 epochs. The model begins with a training accuracy of 29.73% and a test accuracy of 40.86% in the first epoch, reflecting the challenge of learning from random initialization on a 10-class dataset. Accuracy improves steadily across all epochs, with the learning rate reductions at epochs 5 and 8 producing visible acceleration in convergence. The baseline model achieves a final test accuracy of 75.41%, which serves as the reference benchmark for all subsequent optimization experiments.

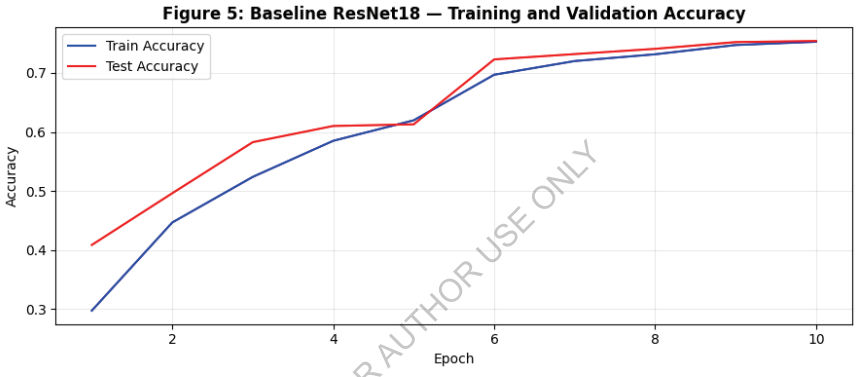


Figure 14.5. Baseline ResNet18 Training and Validation Accuracy Curves.

14.5.2 GPU vs. CPU Benchmarking Results

Figure 14.6 presents the inference latency comparison between CPU and GPU execution across all five batch sizes. At batch size 1, CPU and GPU latency are comparable at approximately 7ms and 4ms respectively, confirming that the GPU advantage is minimal for single-sample inference due to data transfer overhead. At batch size 128, CPU latency reaches approximately 125ms while GPU latency remains below 10ms — a speedup of approximately 13x — demonstrating the substantial throughput advantage of GPU execution for large-batch workloads. The crossover point at which GPU throughput substantially exceeds CPU is observed at approximately batch size 32, consistent with the architectural characteristics of ResNet18.

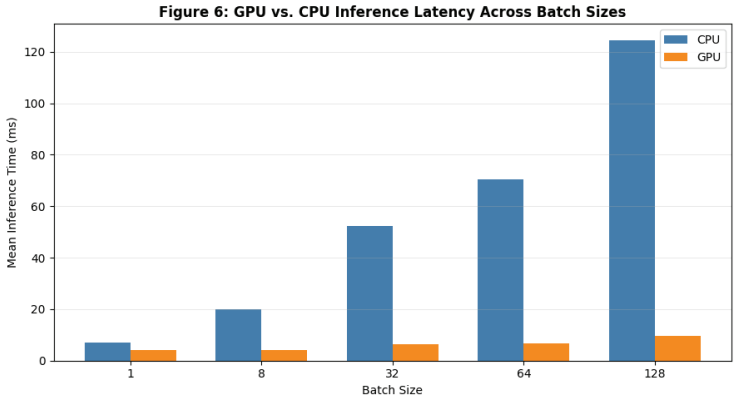


Figure 14.6. GPU vs. CPU Inference Throughput Across Batch Sizes.

14.5.3 Pruning Results

Figure 14.7 presents the relationship between pruning sparsity and post-pruning test accuracy. At 20% sparsity, the model achieves 75.58% accuracy — marginally above the baseline of 75.41% — indicating that a small amount of pruning can act as a form of regularization. At 40% sparsity, accuracy drops negligibly to 74.97%, confirming that a substantial proportion of ResNet18 weights are redundant. At 60% sparsity, a moderate drop to 71.09% is observed, and at 80% sparsity, accuracy falls sharply to 47.37%, indicating that pruning beyond 60% removes weights encoding genuinely task-relevant features.

FOR AUTHOR USE ONLY

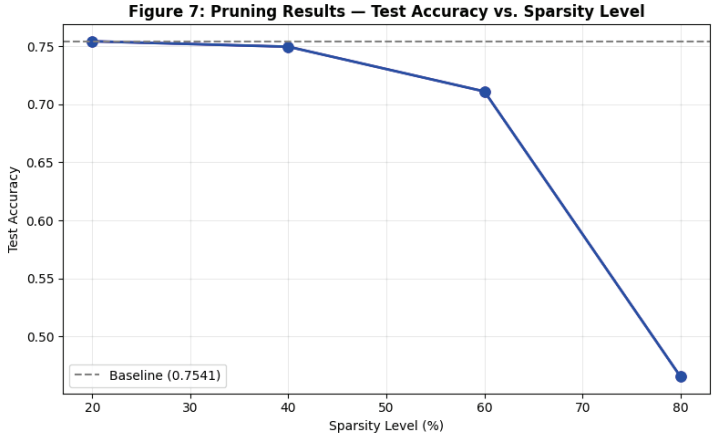


Figure 14.7. Pruning Results — Test Accuracy vs. Sparsity Level.

14.5.4 Quantization Results

Figure 14.8 presents the model size and inference latency comparison between the FP32 baseline and the INT8 quantized model. When serialized with torch.save, the stored model size remains approximately 42 MB for both variants, because PyTorch dynamic quantization applies quantization primarily during inference rather than rewriting all stored parameters. Inference latency is meaningfully reduced from approximately 54ms to approximately 40ms for batch size 32 — a speedup of approximately 1.35x — as INT8 arithmetic operations execute faster than FP32 on CPU hardware with native integer arithmetic support. Test accuracy after quantization is preserved within 1 percentage point of the FP32 baseline, confirming that the precision reduction has negligible impact on classification performance.

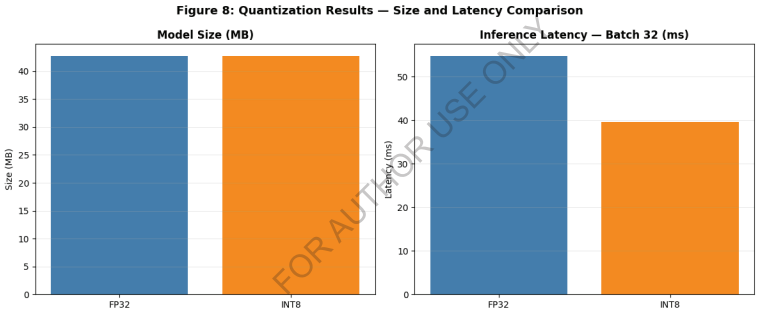


Figure 14.8. Quantization Results — Model Size and Inference Latency Comparison.

14.5.5 Comparative Summary

Figure 14.9 presents a unified comparative summary of all optimization techniques applied in this chapter. The accuracy chart shows that pruning at 20% and 40% sparsity preserves the baseline accuracy of 75.41%, while 60% sparsity introduces a moderate degradation to 71.09% and 80% sparsity causes a significant drop to 47.37%. INT8 quantization preserves accuracy within 1 percentage point of the baseline while delivering meaningful latency reduction. The model size chart confirms that dynamic quantization does not reduce the stored model size for this architecture but delivers its benefit primarily through faster inference. GPU acceleration delivers the greatest

throughput improvement for large-batch workloads without any accuracy trade-off.

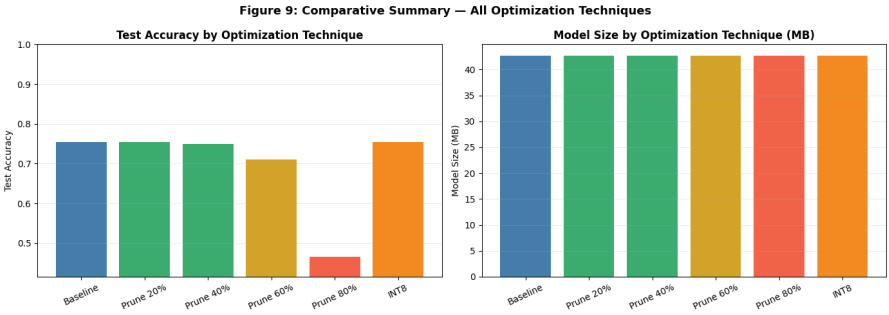


Figure 14.9. Comparative Summary — All Optimization Techniques.

14.6. Discussion

The experimental results confirm the practical value of all three optimization techniques examined in this chapter. GPU acceleration provides a 13x throughput improvement over CPU at batch size 128, making it the preferred approach for high-volume production systems. For edge deployment scenarios with single-sample latency requirements, CPU-based inference remains competitive, particularly at batch size 1 where the GPU advantage narrows to approximately 1.75x due to data transfer overhead.

FOR AUTHOR USE ONLY

The pruning experiments demonstrate that ResNet18 on CIFAR-10 contains substantial parameter redundancy: sparsity levels up to 40% produce negligible accuracy degradation, with 20% sparsity even marginally improving test accuracy from 75.41% to 75.58% through a mild regularization effect. The sharp accuracy declines at 80% sparsity — dropping to 47.37% — marks the point at which pruning exceeds the network's redundancy capacity, consistent with the lottery ticket hypothesis of Frankle and Carlin (2019) and with the empirical findings of Han et al. (2015).

Dynamic INT8 quantization delivers a latency speedup of approximately 1.35x with near-zero accuracy loss, making it directly applicable to any of the convolutional models developed in previous chapters. The absence of a stored model size reduction for ResNet18 reflects a known characteristic of PyTorch's dynamic quantization implementation, which applies quantization at inference time rather than fully converting the stored state dict. Static quantization with calibration data, or the use of TorchScript-based export, would produce a fully

quantized saved model with the expected 75% size reduction. These approaches represent natural extensions of the techniques demonstrated in this chapter.

A key limitation of the experiments presented in this chapter is the use of a single GPU in Google Collab for the benchmarking experiments. The GPU measurements reported reflect single-device execution on a Collab T4 GPU, which is representative of a low-cost cloud inference environment but not of multi-GPU production systems. Furthermore, the unstructured pruning approach employed here does not deliver wall-clock speedups without sparse computation libraries, as standard dense matrix operations process zero values identically to non-zero values. Structured pruning, which removes entire channels or filters to produce genuinely smaller dense networks, represents a more immediately practical compression approach for production deployment and is a natural direction for future experimentation.

FOR AUTHOR USE ONLY

14.7. Conclusion

This chapter has presented a comprehensive treatment of deployment and optimization techniques for deep learning models, covering GPU vs. CPU performance optimization, magnitude-based model compression through pruning, and dynamic INT8 quantization. All three techniques were demonstrated on a ResNet18 model trained on CIFAR-10, providing a unified experimental framework for comparing their accuracy-efficiency trade-offs.

The principal findings are as follows. GPU acceleration delivers a 13x throughput improvement over CPU at batch size 128, with the advantage diminishing at small batch sizes where data transfer overhead becomes significant. Unstructured pruning preserves the baseline accuracy of 75.41% at sparsity levels up to 40%, with accuracy declining moderately at 60% and sharply at 80% sparsity. Dynamic INT8 quantization reduces inference latency by approximately 1.35x with near-zero accuracy loss, while the stored model size reduction requires static quantization or TorchScript export for full effect.

The deployment and optimization techniques presented in this chapter provide the tools necessary to bridge the gap between a well-performing trained model and a production-ready system that operates efficiently under real-world

computational and memory constraints. Practitioners equipped with these techniques can select the most appropriate optimization strategy for their deployment scenario — whether prioritizing inference throughput, memory footprint, or latency — without compromising the predictive performance achieved during training.

FOR AUTHOR USE ONLY

Chapter 15 — From Algorithms to Intelligent Systems

15.1 Introduction and Literature Review

In applied machine learning, the ability to train a model is distinct from the ability to deploy one effectively. Selecting an appropriate algorithm, configuring its hyperparameters, and designing an architecture that generalizes to unseen data determine whether a model performs well in practice or merely fits the training data. Domingos (2012) identifies model selection as one of the most significant decisions in the machine learning pipeline, noting that no single algorithm consistently outperforms all others across all tasks, and that the interaction between algorithm, data characteristics, and evaluation criteria must guide selection rather than complexity or familiarity alone. In clinical applications, this decision is particularly important: the choice of evaluation metric must reflect the asymmetric cost of different error types, as a false negative in a medical screening context — a missed diagnosis — typically has substantially greater clinical impact than a false positive.

FOR AUTHOR USE ONLY

The theoretical framework underpinning both model selection and architecture design is the bias-variance tradeoff, formally established by Geman, Bienenstock, and Doursat (1992). Their decomposition of expected prediction error into bias, variance, and irreducible noise provides a principled basis for understanding why models of insufficient capacity underfit — producing consistently high error across both training and test sets — while models of excessive capacity overfit by memorizing training-specific patterns at the expense of generalization. Hastie, Tibshirani, and Friedman (2009) extend this framework to the model selection context, arguing that the optimal model is not the most complex one available but the one whose capacity is most appropriately matched to the structure and scale of the data at hand.

When neural network architectures are under consideration, the design choices governing model capacity, regularization, and training termination become critical determinants of generalization performance. Goodfellow, Bengio, and Courville (2016) describe regularization as any modification to a learning algorithm intended to reduce generalization error without necessarily reducing training error, encompassing techniques such as L2 weight penalty, Dropout, and Batch Normalization, each of which constrains the model's effective hypothesis space in a distinct way. Both experiments in this chapter are implemented using scikit-learn for classical model evaluation and

TensorFlow/Keras for neural network construction, following the established frameworks of Pedregosa et al. (2011) and Goodfellow et al. (2016).

Both experiments are conducted on the Pima Indians Diabetes dataset, originally introduced by Smith et al. (1988) in research applying the ADAP learning algorithm to diabetes onset prediction. The dataset provides a clinically relevant, structured binary classification task of modest size that is well-suited to comparing classical and neural approaches on equal footing, and whose class imbalance and asymmetric error costs make metric selection a substantive methodological decision.

The chapter first establishes the theoretical foundations of the bias-variance tradeoff, hyperparameter optimization, and neural network regularization, before presenting the Model Selection Strategy and Architecture Design Philosophy experiments in sequence, each with its own methodology, results, and discussion. A concluding section synthesizes the findings across both experiments and addresses the practical conditions under which a simpler classical model is preferable to a more complex neural network.

FOR AUTHOR USE ONLY

15.2 The Bias-Variance Tradeoff

A central challenge in supervised machine learning is selecting a model of appropriate complexity for a given task and dataset. The bias-variance tradeoff provides the theoretical framework for understanding this challenge. Geman, Bienenstock, and Doursat (1992) formally decomposed the generalization error of a learning algorithm into three components: bias, variance, and irreducible noise. Bias refers to the systematic error introduced by a model that makes overly simplistic assumptions about the underlying data-generating process. Variance refers to the model's sensitivity to fluctuations in the training data - a high-variance model adapts too closely to its specific training samples and fails to generalize to unseen examples. Irreducible noise, as the name implies, cannot be minimized through model design and reflects inherent randomness in the data.

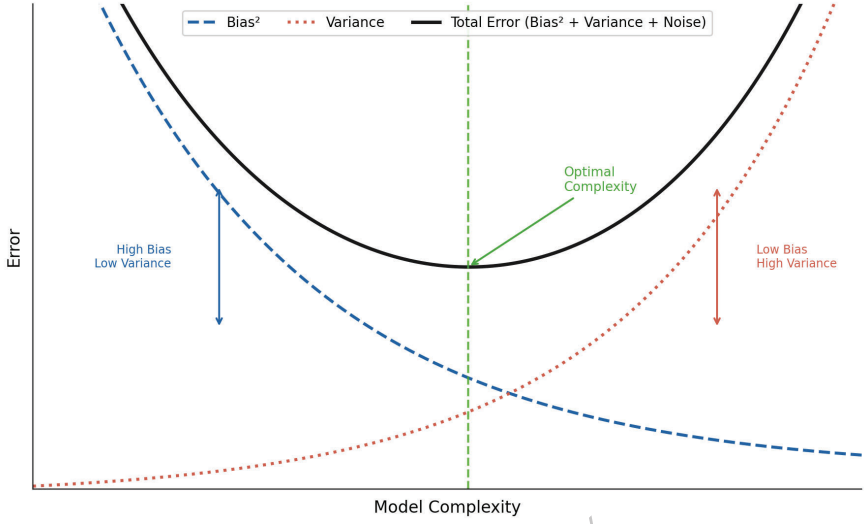


Figure 15.1. Bias-variance decomposition across model complexity.

FOR AUTHOR USE ONLY

As illustrated in Figure 15.1, models of low complexity - such as linear classifiers with few parameters - tend to exhibit high bias and low variance. These models underfit the training data, producing consistently poor predictions across both training and test sets. Conversely, models of high complexity tend to exhibit low bias and high variance. These models overfit the training data, achieving near-perfect training performance while generalizing poorly to unseen examples. The optimal model occupies the region of minimal total error, where bias and variance are jointly balanced. Hastie, Tibshirani, and Friedman (2009) describe this balance as the fundamental objective of model selection: identifying the hypothesis class with sufficient capacity to capture genuine structure in the data, without fitting idiosyncratic noise patterns specific to the training sample.

This decomposition directly motivates both experimental components of the present chapter. In the Model Selection Strategy, the tradeoff is examined across four classical classifiers of varying complexity, where additional computational investment in tuning does not guarantee improved generalization if the underlying model capacity is misaligned with the data. In the Architecture Design Philosophy, three neural network architectures are designed explicitly to instantiate the three regimes - underfitting, overfitting, and

balanced generalization - allowing the tradeoff to be observed empirically through training dynamics rather than asserted theoretically.

15.3. Hyperparameter Optimization and Model Evaluation

The performance of a machine learning model depends not only on the choice of algorithm but also on the configuration of its hyperparameters — parameters that govern the learning process and are not estimated from the data during training. Domingos (2012) identifies hyperparameter selection as one of the most practically significant decisions in applied machine learning, noting that the same algorithm under different configurations can produce substantially different outcomes on the same dataset.

Cross-validation is the standard approach for hyperparameter selection without overfitting the model to the test set. In k-fold cross-validation, the training data is partitioned into k non-overlapping subsets, and the model is trained k times, each time using k-1 subsets for training and the remaining subset for validation. The average validation performance across all k folds provides a more reliable estimate of generalization error than a single hold-out split, particularly on smaller datasets where a single split may produce unstable estimates due to sampling variance.

FOR AUTHOR USE ONLY

Grid search exhaustively evaluates all combinations of hyperparameter values within a predefined search space, selecting the configuration that maximizes the chosen evaluation metric across cross-validation folds. When combined with k-fold cross-validation, this procedure — implemented as GridSearchCV in scikit-learn — provides a systematic and reproducible approach to model configuration. The choice of scoring metric is equally important: in clinical classification tasks where error types carry asymmetric costs, optimizing for accuracy may select configurations that improve majority-class performance at the expense of minority-class recall. F1-score, as the harmonic mean of precision and recall, penalizes configurations that trade recall for precision and is therefore a more appropriate optimization target in this context.

15.4. Regularization Strategies in Neural Networks

Neural networks with large numbers of parameters are particularly susceptible to overfitting on small datasets, as the model has sufficient capacity to memorize training samples rather than learn generalizable representations. Goodfellow, Bengio, and Courville (2016) describe regularization as any

modification to a learning algorithm intended to reduce generalization error without necessarily reducing training error. Three regularization strategies are employed in the balanced architecture constructed in this chapter.

L2 weight regularization, also known as weight decay, modifies the training objective by adding a penalty term proportional to the squared magnitude of the model's weights. Rather than minimizing the original loss function alone, the optimizer minimizes a composite objective as defined in Eq.15.1. This penalty discourages large weight values and encourages the network to distribute information across many parameters rather than concentrating it in a small subset, effectively shrinking the model's hypothesis space toward simpler solutions and reducing sensitivity to individual training examples.

$$L_{reg}(\theta) = L(\theta) + \lambda \sum_{i=1}^{n} w_i^2 \,, \qquad \textit{(Eq.15.1)}$$

Where:

$L_{reg}(\theta) = total\ regularized\ loss$

$L(\theta) = original\ unregularized\ loss$

$\lambda = regularization\ coeffecient\ controlling\ the\ penalty\ stregth$

$w_i = weight\ of\ the\ i^{th}\ model\ parameter$

$\sum_{i=1}^{n} \boldsymbol{w_i^2} = sum\ of\ the\ squared\ model\ weights\ (L2\ penalty)$

FOR AUTHOR USE ONLY

Dropout is a stochastic regularization technique in which a random proportion of neurons are deactivated at each training step, preventing co-adaptation between units and forcing the network to learn redundant representations. As illustrated in Figure 15.2, dropout alters the effective network topology during training by randomly removing neurons and their associated connections, while the full network is restored at inference time. This asymmetry between training and inference has a direct consequence for interpreting training curves: because the model operates at reduced capacity during training but at full capacity during evaluation, validation accuracy may legitimately exceed

training accuracy in well-regularized networks — a pattern observed in the balanced model discussed in the Results and Discussion section below.

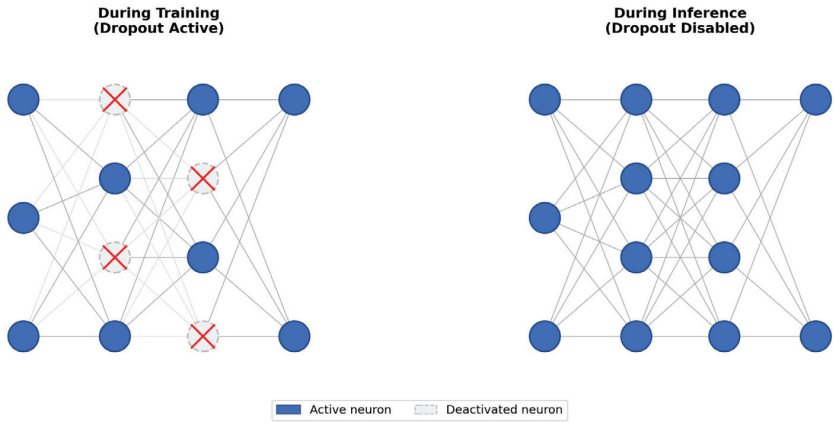


Figure 15.2. Dropout regularization: training vs inference.

FOR AUTHOR USE ONLY

Batch Normalization normalizes the activations of each layer across the current mini-batch, reducing internal covariate shift — the tendency for the distribution of layer inputs to shift during training as earlier layer weights are updated. By stabilizing the activation distributions, Batch Normalization reduces the dependence of each layer on the precise scale of inputs from preceding layers, contributing to more stable and consistent gradient flow during backpropagation. When applied in conjunction with dropout, Batch Normalization has been shown to accelerate convergence and reduce sensitivity to the choice of learning rate.

Early Stopping monitors a designated validation metric throughout training and halts the optimization process when no improvement is observed for a specified number of consecutive epochs, referred to as the patience parameter. As illustrated in Figure 15.3, once validation loss begins to increase — indicating that continued training is refining the model's fit to training-specific noise rather than generalizable structure — the procedure waits for a fixed number of epochs before terminating, then restores the model weights from the epoch of best validation performance. This mechanism directly addresses the risk of training beyond the point of optimal generalization, effectively using the validation curve as an automatic stopping criterion without requiring the practitioner to specify a fixed number of epochs in advance.

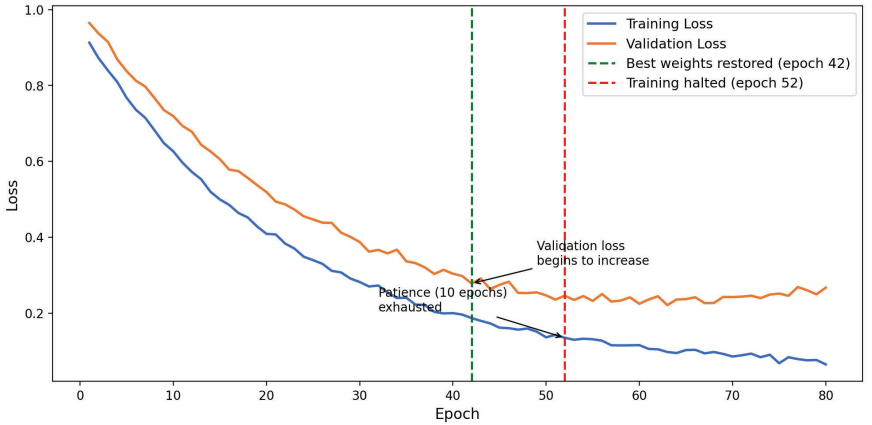


Figure 15.3. Early stopping: validation loss monitoring and weight restoration.

All three regularization strategies are employed jointly in the balanced architecture to demonstrate their combined effect on training stability and generalization performance relative to the unregularized underfit and overfit baselines.

CO-AUTHOR USE ONLY

15.5. Model Selection Strategy

The first experiment examines the problem of selecting an appropriate machine learning model for a structured clinical classification task. The Pima Indians Diabetes dataset, originally introduced by Smith et al. (1988) in the context of early diabetes onset prediction research, serves as the dataset for this experiment. The rationale for selecting this dataset was discussed in the Introduction. It is reused here to evaluate classical machine learning models under identical experimental conditions.

The dataset comprises 768 samples drawn from a population of Pima Indian women, each described by eight clinical input features — including plasma glucose concentration, body mass index, insulin levels, and age — alongside a binary outcome variable indicating the presence or absence of diabetes mellitus. Four classical supervised classifiers of varying complexity are evaluated: Logistic Regression, Decision Tree, Random Forest, and Support Vector Machine. Rather than selecting the model that achieves the highest default accuracy, this section applies GridSearchCV-driven hyperparameter tuning optimized for F1-score and evaluates each model against both

predictive performance and computational cost, reflecting the practical constraints that govern real-world deployment decisions.

15.5.1. Methodology and Dataset

The required libraries are imported. Scikit-learn provides the four classifiers — LogisticRegression, DecisionTreeClassifier, RandomForestClassifier, and SVC — alongside preprocessing utilities, model selection tools, and evaluation metrics. Pandas and NumPy handle data loading and numerical operations, Matplotlib and Seaborn are used for visualization, and the time module records wall-clock training time.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import time

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.metrics import (accuracy_score, precision_score,
                             recall_score, f1_score,
                             classification_report, confusion_matrix)
```

FOR AUTHOR USE ONLY

Code Snippet 15.1. Imports and environment setup.

The dataset is loaded from a local CSV file and inspected to confirm its structure prior to any preprocessing. Initial inspection reports the dataset dimensions, column names, class distribution, and a check for explicitly encoded missing values.

```
df = pd.read_csv('diabetes.csv')

print("Dataset shape:", df.shape)
print("\nColumn names:", df.columns.tolist())
print("\nTarget distribution:\n", df['Outcome'].value_counts())
print("\nMissing values:\n", df.isnull().sum())
df.head()
```

Code Snippet 15.2. Data loading and initial inspection.

The dataset contains 768 samples across nine columns — eight clinical input features and one binary target variable. The feature set includes Pregnancies, Glucose, BloodPressure, SkinThickness, Insulin, BMI, DiabetesPedigreeFunction, and Age. The target column Outcome reflects a class distribution of 500 negative cases and 268 positive cases, confirming a class imbalance of approximately 65.1% to 34.9% that motivates the use of F1-score as the primary evaluation metric rather than accuracy. No explicitly encoded missing values are detected at this stage; however, the dataset contains a subtler data quality issue that requires preprocessing before model training.

```
Dataset shape: (768, 9)

Column names:
  ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness']
  ['Insulin', 'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome']

Target distribution:
 Outcome
0    500
1    268
Name: count, dtype: int64

Missing values:
  Pregnancies                 0
  Glucose                     0
  BloodPressure               0
  SkinThickness               0
  Insulin                     0
  BMI                         0
  DiabetesPedigreeFunction    0
  Age                         0
  Outcome                     0
```

FOR AUTHOR USE ONLY

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Output 15.1. Data overview.

```python
# Columns where 0 is biologically impossible
zero_invalid_cols = ['Glucose', 'BloodPressure', 'SkinThickness',
                     'Insulin', 'BMI']

print("Zero counts per column:")
print((df[zero_invalid_cols] == 0).sum())

# Replace 0 with NaN, then impute with median
for col in zero_invalid_cols:
    df[col] = df[col].replace(0, np.nan)
    df[col] = df[col].fillna(df[col].median())

print("\nAfter cleaning - any remaining NaNs:", df.isnull().sum().sum())
```

Code Snippet 15.3. Zero value detection and median imputation.

Five columns — Glucose, BloodPressure, SkinThickness, Insulin, and BMI — contain zero values that are biologically implausible as genuine clinical measurements. A fasting plasma glucose of zero, for instance, is physiologically impossible and reflects a missing observation rather than a true recorded value. The zero counts per column confirm the extent of this issue: Glucose contains 5 affected entries, BloodPressure 35, SkinThickness 227, Insulin 374, and BMI 11. These entries are replaced with NaN and subsequently imputed using each column's median value. Median imputation is preferred over mean imputation here because clinical measurements such as Insulin exhibit right-skewed distributions where extreme values would disproportionately shift the mean, introducing systematic bias into imputed entries. Following imputation, a confirmation check verifies that no remaining NaN values exist anywhere in the dataset, ensuring a complete dataset for model training.

FOR AUTHOR USE ONLY

```
Zero counts per column:
Glucose            5
BloodPressure     35
SkinThickness    227
Insulin          374
BMI               11
dtype: int64

After cleaning - any remaining NaNs: 0
```

Output 15.2. Data cleaning results.

```
X = df.drop(columns=['Outcome'])
y = df['Outcome']

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    random_state=42,
    stratify=y
)

# Scaling - required for Logistic Regression and SVM
# not necessary for Tree-based models but no issues with scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

print("Train shape:", X_train.shape)
print("Test shape:", X_test.shape)
```

Code Snippet 15.4. Feature-target separation, train-test split and scaling.

FOR AUTHOR USE ONLY

The feature matrix X and target vector y are separated, and the dataset is partitioned into training and test sets using an 80/20 stratified split with random_state=42. Stratification preserves the 65.1/34.9 class ratio in both partitions. The resulting training set contains 614 samples, and the test set 154 samples across eight features. StandardScaler is fitted exclusively on the training set and applied to both splits, ensuring that no information from the test set influences the preprocessing pipeline. While scaling is strictly required only for Logistic Regression and SVM - whose optimization landscapes are sensitive to feature magnitude — it is applied uniformly across all models to ensure that performance differences reflect model characteristics rather than preprocessing inconsistencies.

```
Train shape: (614, 8)
Test shape: (154, 8)
```

Output 15.3. Data shape after train-test splitting.

```
models = {
    'Logistic Regression': LogisticRegression(max_iter=1000, random_state=42),
    'Decision Tree': DecisionTreeClassifier(max_depth=5, random_state=42),
    'Random Forest': RandomForestClassifier(n_estimators=100, random_state=42),
    'SVM': SVC(kernel='rbf', random_state=42)
}

results = []

for name, model in models.items():
    start_time = time.time()
    model.fit(X_train_scaled, y_train)
    train_time = time.time() - start_time

    y_pred = model.predict(X_test_scaled)

    results.append({
        'Model': name,
        'Accuracy': accuracy_score(y_test, y_pred),
        'Precision': precision_score(y_test, y_pred),
        'Recall': recall_score(y_test, y_pred),
        'F1 Score': f1_score(y_test, y_pred),
        'Training Time (s)': train_time
    })

results_df = pd.DataFrame(results)
print(results_df.to_string(index=False))
```

FOR AUTHOR USE ONLY

Code Snippet 15.5. Baseline model training and evaluation.

All four classifiers are instantiated and trained on the scaled training set using initial default configurations. Accuracy, precision, recall, F1-score, and wall-clock training time are recorded for each model to establish a pre-tuning performance baseline. The baseline results are presented in Output 15.4.

```
              Model  Accuracy  Precision   Recall  F1 Score  Training Time (s)
Logistic Regression  0.707792   0.600000 0.500000  0.545455          0.028665
      Decision Tree  0.759740   0.639344 0.722222  0.678261          0.012263
      Random Forest  0.779221   0.717391 0.611111  0.660000          0.791365
                SVM  0.740260   0.652174 0.555556  0.600000          0.038291
```

Output 15.4. Baseline model performance across all four classifiers.

The baseline results reveal meaningful differences in performance profile across classifiers. Random Forest achieves the highest baseline accuracy at 0.779, while Decision Tree achieves the highest recall at 0.722 and the highest F1-score at 0.678 — the metric of primary interest given the clinical cost asymmetry of false negatives in diabetes screening. Logistic Regression, despite its computational efficiency, returns the lowest F1-score at 0.545, reflecting limited capacity to capture non-linear relationships in the feature

space. These initial results motivate a structured hyperparameter search to assess whether the current performance rankings are a product of model architecture or suboptimal default configurations.

```
tuned_results_final = []
fitted_models = {}  # store actual fitted models here

for name, config in param_grids.items():
    start_time = time.time()

    grid = GridSearchCV(
        config['model'],
        config['params'],
        cv=5,
        scoring='f1',
        n_jobs=-1
    )
    grid.fit(X_train_scaled, y_train)

    train_time = time.time() - start_time

    best_model = grid.best_estimator_
    fitted_models[name] = best_model  # save it for later use

    y_pred = best_model.predict(X_test_scaled)

    tuned_results_final.append({
        'Model': name,
        'Accuracy': accuracy_score(y_test, y_pred),
        'Precision': precision_score(y_test, y_pred),
        'Recall': recall_score(y_test, y_pred),
        'F1 Score': f1_score(y_test, y_pred),
        'Training Time (s)': train_time
    })

tuned_df = pd.DataFrame(tuned_results_final)
print(tuned_df.to_string(index=False))
```

FOR AUTHOR USE ONLY

Code Snippet 15.6. Defining hyperparameters and GridSearchCV tuning.

The hyperparameter search spaces are defined for all four classifiers. For Logistic Regression, the regularization strength C is varied over {0.01, 0.1, 1, 10, 100} with L2 penalty. For Decision Tree, max_depth is varied over {3, 5, 7, None}, min_samples_split over {2, 5, 10}, and min_samples_leaf over {1, 2, 4}. For Random Forest, n_estimators is varied over {50, 100, 200} and max_depth over {None, 5, 10}. For SVM, C is varied over {0.1, 1, 10} with rbf and linear kernels. GridSearchCV is applied to each classifier with five-fold cross-validation and F1 as the scoring criterion, ensuring that all hyperparameter

selection decisions are driven by the metric most relevant to the clinical classification task. Each best-performing estimator is stored in a fitted_models dictionary upon search completion, enabling downstream evaluation including confusion matrix generation without rerunning the search. The tuned results are presented in Output 15.5.

Model	Accuracy	Precision	Recall	F1 Score	Training Time (s)
Logistic Regression	0.707792	0.600000	0.500000	0.545455	3.072887
Decision Tree	0.785714	0.698113	0.685185	0.691589	2.584029
Random Forest	0.753247	0.673913	0.574074	0.620000	149.872518
SVM	0.740260	0.652174	0.555556	0.600000	0.653490

Output 15.5. Tuned model performance across all four classifiers.

Following hyperparameter tuning, the Decision Tree achieved the best overall performance with an accuracy of 0.786, precision of 0.698, recall of 0.685, and F1-score of 0.692 — the highest F1 and recall of any tuned classifier. Notably, Random Forest, despite its substantially longer tuning time of approximately 150 seconds, achieves a lower tuned F1-score of 0.620 than the Decision Tree's 2.6-second search, indicating that greater computational investment in hyperparameter search does not necessarily produce proportional improvements in predictive performance. Logistic Regression and SVM show no improvement in F1 following tuning relative to their baseline configurations.

FOR AUTHOR USE ONLY

```
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

tuned_df.set_index('Model')[['Accuracy', 'Precision', 'Recall', 'F1 Score']].plot(
    kind='bar', ax=axes[0], colormap='viridis'
)
axes[0].set_title('Tuned Model Performance Comparison')
axes[0].set_ylabel('Score')
axes[0].legend(loc='lower right')
axes[0].tick_params(axis='x', rotation=30)

axes[1].bar(tuned_df['Model'], tuned_df['Training Time (s)'], color='coral')
axes[1].set_title('Training Time Comparison (Log Scale)')
axes[1].set_ylabel('Time (seconds)')
axes[1].set_yscale('log')  # log scale since RF time is 100x larger than others
axes[1].tick_params(axis='x', rotation=30)

plt.tight_layout()
plt.savefig('model_comparison_tuned.png', dpi=150)
plt.show()
```

Code Snippet 15.7. Tuned models performance and training time visualization.

The tuned performance metrics are visualized as grouped bar charts alongside a log-scale bar chart of tuning time per model (Figure 15.4). A logarithmic scale

is applied to the time axis to accommodate the order-of-magnitude difference between Random Forest's tuning duration and those of the remaining classifiers. The corresponding tuning times are presented in Figure 15.5.

```
best_dt_final = fitted_models['Decision Tree']  # load from storage
y_pred_final = best_dt_final.predict(X_test_scaled)

cm = confusion_matrix(y_test, y_pred_final)
plt.figure(figsize=(6,5))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['No Diabetes', 'Diabetes'],
            yticklabels=['No Diabetes', 'Diabetes'])
plt.title('Confusion Matrix - Tuned Decision Tree')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.tight_layout()
plt.savefig('confusion_matrix_dt.png', dpi=150)
plt.show()
```

Code Snippet 15.8. Confusion matrix for the tuned Decision Tree.

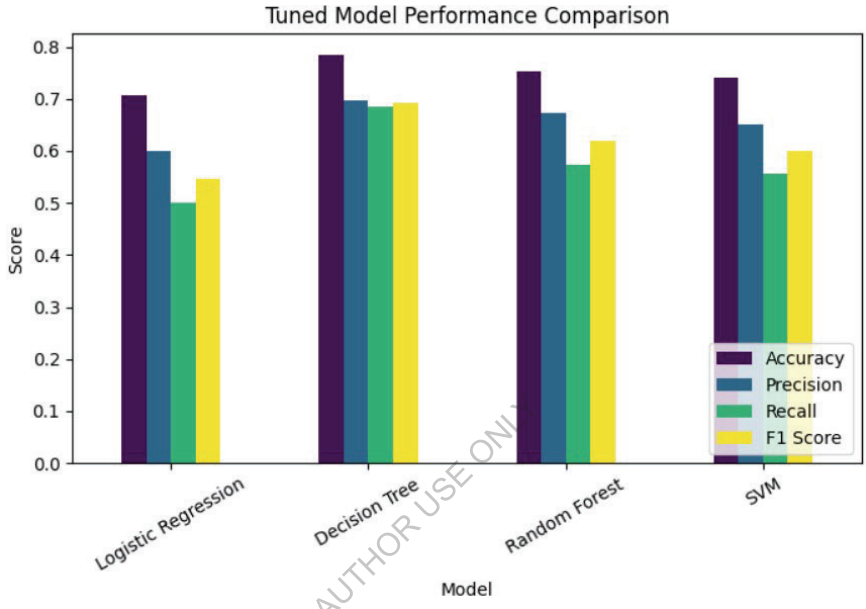
FOR AUTHOR USE ONLY

The confusion matrix for the best-performing model, tuned Decision Tree, is generated using the stored model from the fitted_models dictionary. The matrix provides a breakdown of predictions on the held-out test set, as shown in Figure 15.6 below.

15.5.2 Results and Discussion

Figure 15.4 presents the tuned performance metrics across all four classifiers following GridSearchCV optimization. The tuned Decision Tree achieved the best overall performance, recording an accuracy of 0.786, precision of 0.698, recall of 0.685, and an F1-score of 0.692 — the highest F1 and recall of any model evaluated. Given that recall directly quantifies the proportion of true diabetic cases correctly identified, and that false negatives carry the highest clinical cost in a screening context, the Decision Tree's performance on these two metrics is the primary finding of this comparison. Logistic Regression returns the weakest result with an F1-score of 0.545 and recall of 0.500, suggesting that its linear decision boundary is insufficient to capture the non-linear relationships present in the feature space, and that tuning the regularization strength alone cannot compensate for this structural limitation. SVM achieves an F1-score of 0.600, showing marginal improvement in precision relative to baseline but no gain in recall. Neither Logistic Regression nor SVM shows meaningful improvement in F1 following tuning,

suggesting both models had reached their performance limits under default configurations and that the bottleneck is model capacity rather than hyperparameter configuration.



FOR AUTHOR USE ONLY

Figure 15.4. Tuned model performance comparison.

Random Forest produced an important observation. Despite being the most complex ensemble method in the comparison and requiring the most extensive hyperparameter search, the tuned Random Forest achieves an F1-score of 0.620 — lower than the tuned Decision Tree by a margin of 0.072. This outcome is consistent with a known limitation of ensemble methods on small datasets: with only 614 training samples, the individual trees within the forest are trained on bootstrap samples of approximately 614 rows, which provides insufficient diversity for the ensemble's variance reduction mechanism to yield the gains typically observed on larger datasets. This result is consistent with the argument presented by Domingos (2012) that increased model complexity and tuning effort do not guarantee proportionally improved generalization, particularly when dataset size constrains the learning signal available to the model.

Figure 15.5 complements these findings from a computational cost perspective. Random Forest's tuning process required approximately 150 seconds of wall-clock time — roughly 58 times longer than the Decision Tree's 2.6-second search — yet produced a worse F1-score. SVM completed tuning in under one second, while Logistic Regression required approximately three seconds. In a deployment context where inference latency, retraining frequency, and computational resource constraints are practical considerations, the Decision Tree's combination of strong predictive performance and minimal tuning cost makes it the most suitable model for this dataset.

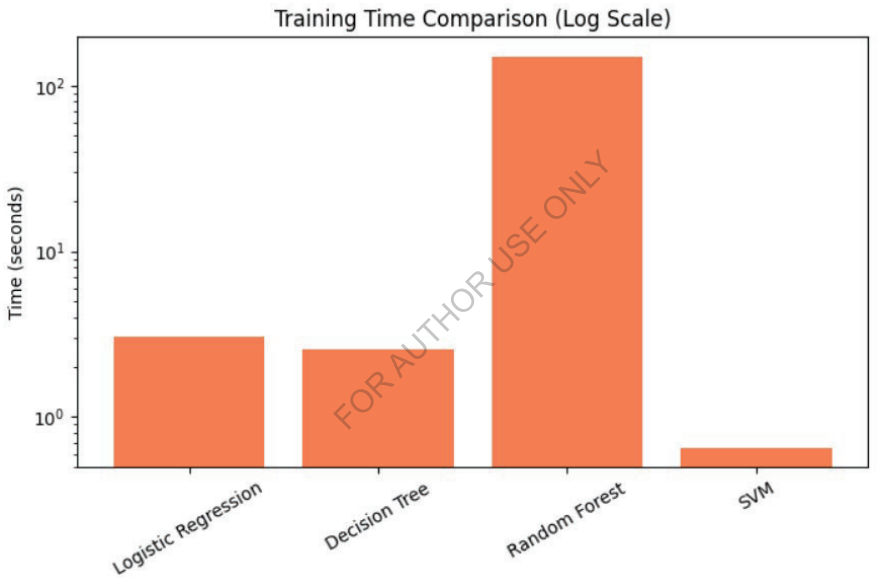


Figure 15.5. Training time comparison across classifiers.

Table 15.1 consolidates the tuned performance metrics and GridSearchCV training time across all four classifiers, providing a unified view of predictive performance and computational cost following hyperparameter optimization. The Decision Tree provides the most favorable balance across all reported metrics, recording the highest F1-score and recall while requiring the second-shortest tuning time of the four models evaluated. Random Forest's disproportionate tuning cost of approximately 150 seconds — nearly 58 times

that of the Decision Tree — alongside its lower F1-score of 0.620 makes it the least computationally efficient model evaluated.

Table 15.1. Tuned model performance and computational cost.

Model	Accuracy	Precision	Recall	F1 Score	Training Time (s)
Logistic Regression	0.708	0.600	0.500	0.545	3.073
Decision Tree	0.786	0.698	0.685	0.692	2.584
Random Forest	0.753	0.674	0.574	0.620	149.873
SVM	0.740	0.652	0.556	0.600	0.653

FOR AUTHOR USE ONLY

Figure 15.6 presents the confusion matrix for the tuned Decision Tree evaluated on the 154-sample held-out test set. The model correctly classifies 84 of 100 negative cases and 37 of 54 positive cases, yielding 17 false negatives and 16 false positives. The false negative count is clinically important: 17 diabetic patients were incorrectly predicted as negative, representing missed diagnoses that would delay treatment without further screening. The false positive count of 16 indicates patients flagged for diabetes who do not have the condition, leading to unnecessary confirmatory testing but does not carry the same severity as a missed diagnosis. The differing clinical impact of these error types is consistent with the rationale for optimizing F1-score — which weights recall alongside precision — rather than accuracy alone, which would mask the model's sensitivity to the minority positive class. The confusion matrix is consistent with the reported recall of 0.685, confirming that the model correctly identifies around two-thirds of true diabetic cases on this test set.

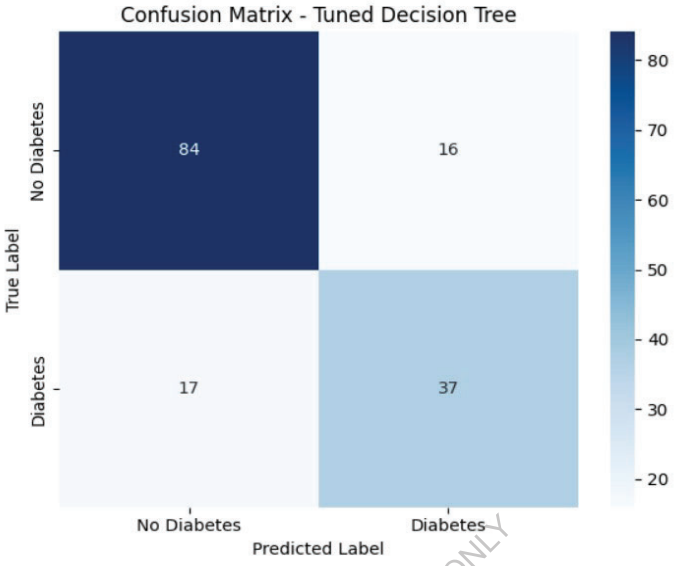


Figure 15.6. Confusion matrix for the tuned Decision Tree.

FOR AUTHOR USE ONLY

Overall, the tuned Decision Tree provided the best balance between predictive performance, interpretability, and computational efficiency, motivating the comparison with neural network architectures in the following experiment, which examines whether neural network architectures — with their substantially larger parameter counts and greater design flexibility — offer meaningful generalization advantages over classical models on this dataset, or whether the additional complexity introduces training instability without commensurate performance gains.

15.6. Architecture Design Philosophy

The second experiment examines how neural network architecture influences generalization performance. The same Pima Indians Diabetes dataset is retained for this section, allowing architecture to remain the only experimental variable - so observed performance differences reflect model design rather than dataset characteristics. Three multilayer perceptron architectures were constructed to explicitly instantiate the three regimes of the bias-variance tradeoff: an underfit model with severely restricted capacity, an overfit model with excessive capacity and no regularization, and a balanced model incorporating L2 regularization, Batch Normalization, Dropout, and Early

Stopping. Training and validation accuracy and loss curves are analyzed across all three architectures to demonstrate how each design choice manifests in observable training dynamics, illustrating the theoretical principles presented earlier.

15.6.1. Methodology and Dataset

TensorFlow/Keras provides the deep learning framework, including the Sequential model API, Dense, Dropout, and BatchNormalization layers, the Adam optimizer, L2 regularization, and the EarlyStopping callback. Scikit-learn provides preprocessing utilities and Matplotlib is used for visualization.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.regularizers import l2
from tensorflow.keras.callbacks import EarlyStopping
```

Code Snippet 15.9. Imports and environment setup.

FOR AUTHOR USE ONLY

Global random seeds are fixed immediately after the import step to ensure reproducible training. The Python random module, NumPy, and TensorFlow random state are all seeded to 42, and TF_DETERMINISTIC_OPS is enabled to suppress non-deterministic GPU operations. Seeding is performed immediately after importing the required libraries and before any data or model operations, as TensorFlow consumes its internal random number stream from the point of the first call - any earlier operation would shift the random state and break reproducibility.

```
import os
import random
import tensorflow as tf

SEED = 42

os.environ['PYTHONHASHSEED'] = str(SEED)
random.seed(SEED)
np.random.seed(SEED)
tf.random.set_seed(SEED)

os.environ['TF_DETERMINISTIC_OPS'] = '1'
```

Code Snippet 15.10. Global seed configuration for reproducibility.

The same preprocessing pipeline described in the previous experiment was applied to ensure that differences in performance were attributable solely to architectural design.

Three multilayer perceptron architectures were developed to represent the three regions of the bias–variance trade-off. All three models used the Adam optimizer (learning rate = 0.001), binary cross-entropy loss, a batch size of 32, a maximum of 100 epochs, and a validation split of 0.2.

```
model_underfit = Sequential([
    Dense(2, activation='relu', input_shape=(X_train_scaled.shape[1],)),
    Dense(1, activation='sigmoid')
])

model_underfit.compile(optimizer=Adam(learning_rate=0.001),
                       loss='binary_crossentropy',
                       metrics=['accuracy'])

history_underfit = model_underfit.fit(
    X_train_scaled, y_train,
    epochs=100,
    batch_size=32,
    validation_split=0.2,
    verbose=0
)

print("Underfit Model - Final Train Acc:",
      round(history_underfit.history['accuracy'][-1], 4))
print("Underfit Model - Final Val Acc:",
      round(history_underfit.history['val_accuracy'][-1], 4))
```

FOR AUTHOR USE ONLY

Code Snippet 15.11. Underfit model architecture, compilation and training.

The underfit architecture consists of a single hidden layer with two neurons using ReLU activation, followed by a sigmoid output unit. No regularization is applied. With only two hidden units processing eight input features, this intentionally limits model capacity below that required to capture the non-linear relationships in the dataset.

```
Underfit Model - Final Train Acc: 0.7739
Underfit Model - Final Val Acc: 0.7724
```

Output 15.6. Underfit model accuracy.

The model trains to a final training accuracy of 0.7739 and validation accuracy of 0.7724 — both converging to a similar low ceiling, consistent with high-bias behavior.

```
model_overfit = Sequential([
    Dense(256, activation='relu', input_shape=(X_train_scaled.shape[1],)),
    Dense(256, activation='relu'),
    Dense(128, activation='relu'),
    Dense(128, activation='relu'),
    Dense(1, activation='sigmoid')
])

model_overfit.compile(optimizer=Adam(learning_rate=0.001),
                      loss='binary_crossentropy',
                      metrics=['accuracy'])

history_overfit = model_overfit.fit(
    X_train_scaled, y_train,
    epochs=100,
    batch_size=32,
    validation_split=0.2,
    verbose=0
)

print("Overfit Model - Final Train Acc:",
      round(history_overfit.history['accuracy'][-1], 4))
print("Overfit Model - Final Val Acc:",
      round(history_overfit.history['val_accuracy'][-1], 4))
```

FOR AUTHOR USE ONLY

Code Snippet 15.12. Overfit model architecture, compilation and training.

The overfit architecture consists of four large hidden layers with 256, 256, 128, and 128 units respectively, all using ReLU activation, followed by a sigmoid output unit. No regularization of any kind is applied. With a parameter count far exceeding what the approximately 491 effective training samples - after the internal 0.2 validation split - can meaningfully constrain, this architecture creates conditions conducive to memorization.

```
Overfit Model - Final Train Acc: 1.0
Overfit Model - Final Val Acc: 0.7642
```

Output 15.7. Overfit model accuracy.

The model trains to a final training accuracy of 1.0 and validation accuracy of 0.7642, a divergence of 23.6 percentage points consistent with overfitting.

```
model_balanced = Sequential([
    Dense(32, activation='relu', input_shape=(X_train_scaled.shape[1]
          kernel_regularizer=l2(0.01)),
    BatchNormalization(),
    Dropout(0.3),

    Dense(16, activation='relu', kernel_regularizer=l2(0.01)),
    BatchNormalization(),
    Dropout(0.2),

    Dense(1, activation='sigmoid')
])

model_balanced.compile(optimizer=Adam(learning_rate=0.001),
                       loss='binary_crossentropy',
                       metrics=['accuracy'])

early_stop = EarlyStopping(monitor='val_loss', patience=10,
                           restore_best_weights=True)

history_balanced = model_balanced.fit(
    X_train_scaled, y_train,
    epochs=100,
    batch_size=32,
    validation_split=0.2,
    callbacks=[early_stop],
    verbose=0
)

print("Balanced Model - Final Train Acc:",
      round(history_balanced.history['accuracy'][-1], 4))
print("Balanced Model - Final Val Acc:",
      round(history_balanced.history['val_accuracy'][-1], 4))
```

FOR AUTHOR USE ONLY

Code Snippet 15.13. Balanced model architecture, compilation and training.

The balanced architecture consists of two hidden layers with 32 and 16 units respectively, each followed by Batch Normalization and Dropout with rates of 0.3 and 0.2. L2 regularization with λ=0.01 is applied to both layers' weight matrices. EarlyStopping monitors validation loss with a patience of 10 epochs and restores the best weights upon stopping. These regularization strategies constrain effective model capacity, stabilize gradient flow, and limit overfitting. Training terminated before 100 epochs and achieves a final training accuracy of 0.8065 and validation accuracy of 0.8211 - the highest validation accuracy among the three architectures.

```
Balanced Model - Final Train Acc: 0.8065
Balanced Model - Final Val Acc: 0.8211
```

Output 15.8. Balanced model accuracy.

Training and validation accuracy curves for all three architectures are plotted side by side with a shared y-axis range of 0.4 to 1.05, enabling direct visual comparison of convergence behavior and the train-validation gap across the three design regimes. The accuracy curves are shown in Figure 15.7.

```
histories = [history_underfit, history_overfit, history_balanced]
titles = ['Underfit Model (Too Simple)', 'Overfit Model (Too Complex)', 'Balanced Model (Well-Designed)']

fig1, axes1 = plt.subplots(1, 3, figsize=(18, 5), dpi=200)
for i, (hist, title) in enumerate(zip(histories, titles)):
    axes1[i].plot(hist.history['accuracy'], label='Train')
    axes1[i].plot(hist.history['val_accuracy'], label='Validation')
    axes1[i].set_title(title)
    axes1[i].set_xlabel('Epoch')
    axes1[i].set_ylabel('Accuracy')
    axes1[i].set_ylim(0.4, 1.05)
    axes1[i].legend()

plt.tight_layout()
plt.savefig('fig_accuracy_comparison.png', bbox_inches='tight')
plt.show()
```

Code Snippet 15.14. Accuracy curves visualization.

Training and validation loss curves for all three architectures are plotted in a separate figure. Loss curves provide diagnostic information that accuracy curves alone cannot capture: since accuracy is a thresholded, discrete metric, it can remain visually flat even as the underlying confidence of incorrect predictions continues to grow.

FOR AUTHOR USE ONLY

```
histories = [history_underfit, history_overfit, history_balanced]
titles = ['Underfit Model (Too Simple)', 'Overfit Model (Too Complex)', 'Balanced Model (Well-Designed)']

fig2, axes2 = plt.subplots(1, 3, figsize=(18, 5), dpi=200)
for i, (hist, title) in enumerate(zip(histories, titles)):
    axes2[i].plot(hist.history['loss'], label='Train')
    axes2[i].plot(hist.history['val_loss'], label='Validation')
    axes2[i].set_title(title)
    axes2[i].set_xlabel('Epoch')
    axes2[i].set_ylabel('Loss')
    axes2[i].legend()

plt.tight_layout()
plt.savefig('fig_loss_comparison.png', bbox_inches='tight')
plt.show()
```

Code Snippet 15.15. Loss curves visualization.

The loss curves are shown in Figure 15.8.

```
y_pred_balanced = (model_balanced.predict(X_test_scaled) > 0.5).astype(int)

print("Balanced Model - Test Set Evaluation")
print(classification_report(y_test, y_pred_balanced,
                            target_names=['No Diabetes', 'Diabetes']))
```

Code Snippet 15.16. Balanced model test set evaluation and classification report.

The balanced model is evaluated on the held-out test set of 154 samples to generate a classification report. Predictions are generated by passing the scaled test features through the trained model and using a decision threshold of 0.5 to the sigmoid output probabilities. Precision, recall, and F1-score are reported for both the negative and positive classes alongside overall accuracy, macro-averaged, and weighted-average metrics.

```
cm_balanced = confusion_matrix(y_test, y_pred_balanced)
plt.figure(figsize=(6, 5))
sns.heatmap(cm_balanced, annot=True, fmt='d', cmap='Blues',
            xticklabels=['No Diabetes', 'Diabetes'],
            yticklabels=['No Diabetes', 'Diabetes'])
plt.title('Confusion Matrix - Balanced Model')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.tight_layout()
plt.savefig('confusion_matrix_balanced.png', dpi=150)
plt.show()
```

Code Snippet 15.17. Confusion matrix – balanced model.

FOR AUTHOR USE ONLY

A confusion matrix for the balanced model is then generated using the stored predictions, providing a class breakdown of correct and incorrect classifications on the held-out test set.

15.6.2. Results and Discussion

Figure 15.7 presents the training and validation accuracy curves for all three architectures across 100 training epochs. The three panels illustrate the bias-variance spectrum introduced earlier, each exhibiting a distinct convergence profile that directly reflects its capacity and regularization configuration. The underfit model displays a smooth, gradual upward trajectory in both training and validation accuracy throughout all 100 epochs, with the two curves rising in near parallel and converging to a final training accuracy of 0.7739 and validation accuracy of 0.7724. The negligible gap of 0.0015 does not indicate successful generalization, instead reflecting the model's inability to fit the training data with any greater precision than it fits unseen data. With only two hidden neurons processing eight input features, the architecture lacks sufficient representational capacity to model the non-linear relationships present in the dataset, resulting in a performance ceiling that additional training cannot overcome. This represents a high-bias regime: both error rates are high and

similar, and the model is constrained by its architecture rather than by its training procedure.

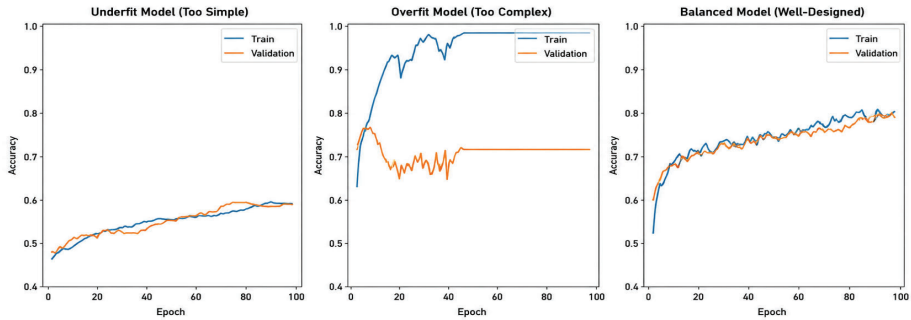


Figure 15.7. Training and validation accuracy curves – underfit, overfit, and balanced models.

The overfit model provides a clear contrast. Training accuracy climbs rapidly toward 1.0, reaching perfect classification by approximately epoch 35 and remaining pinned there for the remainder of training. Validation accuracy rises initially alongside training accuracy but diverges sharply after the first 20 epochs, stabilizing at approximately 0.764 while training accuracy continues to improve.

FOR AUTHOR USE ONLY

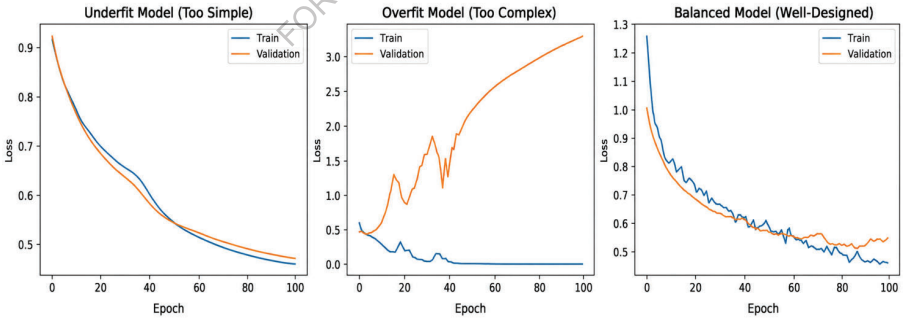


Figure 15.8. Training and validation loss curves - underfit, overfit, and balanced models.

The resulting train-validation gap of 23.6 percentage points at epoch 100 is consistent with high-variance behavior. Notably, the validation accuracy curve appears flat after stabilizing - a pattern that may suggest that the model has

stabilized. The loss curves in Figure 15.8 show that this apparent stability in accuracy is misleading.

Figure 15.8 reveals the training dynamics that accuracy curves alone conceal. For the underfit model, training and validation loss decrease together in near-lockstep throughout all 100 epochs, ending at approximately 0.49 and 0.50 respectively with no divergence. The parallel trajectories confirm that the model is consistently improving but approaching a lower bound imposed by its capacity, not by overfitting. Both curves plateau at a relatively high loss value, consistent with the accuracy ceiling observed in Figure 15.7.

The overfit model's loss curves reveal the extent of memorization that the accuracy panel did not capture. While training loss collapses toward zero by epoch 40 and remains essentially flat thereafter, validation loss diverges dramatically — rising continuously past 3.0 by epoch 100 with no sign of stabilizing. This divergence pattern, where training loss approaches zero while validation loss escalates without bound, consistent with the overfitting behavior described by Geman, Bienenstock, and Doursat (1992): the model becomes progressively more confident in incorrect predictions on unseen data even as its confidence in training predictions reaches high confidence. The accuracy curves in Figure 15.7 do not capture this because accuracy only captures whether the predicted class label is correct, not how extreme the underlying probability estimate has become. A model that predicts 0.51 and a model that predicts 0.99 for the same sample both contribute the same count to accuracy, but their loss contributions are vastly different. The loss curve therefore provides a more sensitive indicator for detecting ongoing overfitting after accuracy has saturated.

DRAFT NOT FOR USE ONLY

The balanced model produces the most stable training profile across both figures. In Figure 15.7, training and validation accuracy rise together from the outset, remaining closely aligned throughout training with the expected noisy convergence associated with Dropout regularization. Training halts before epoch 100 due to the EarlyStopping callback triggering after validation loss fails to improve for ten consecutive epochs, resulting in a shorter training history relative to the other two models. The final training accuracy of 0.8065 and validation accuracy of 0.8211 represent the highest validation accuracy of the three architectures. Validation accuracy slightly exceeds training accuracy because Dropout is active only during training. In Figure 15.8, the balanced model's train and validation loss curves decrease together smoothly, converge

to approximately 0.47 and 0.54 respectively, and show no runaway divergence. EarlyStopping terminates training at the point where validation loss ceases to improve, preventing the continued reduction of training loss from degrading generalization, preventing unnecessary overfitting.

Table 15.2. Training and validation accuracy across neural network architectures.

Architecture	Final Train Accuracy	Final Validation Accuracy	Train-Val Gap
Underfit	0.7739	0.7724	0.0015
Overfit	1.0000	0.7642	0.2358
Balanced	0.8065	0.8211	−0.0146

FOR AUTHOR USE ONLY

Table 15.2 summarizes the final training and validation accuracy for each of the three architectures, providing a numerical summary to complement the curve-based analysis presented in Figures 15.7 and 15.8. The training-validation gap for each model reflects its position on the bias-variance spectrum.

```
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 36ms/step
Balanced Model - Test Set Evaluation
              precision    recall  f1-score   support

 No Diabetes       0.82      0.78      0.80       100
    Diabetes       0.63      0.69      0.65        54

    accuracy                           0.75       154
   macro avg       0.72      0.73      0.73       154
weighted avg       0.75      0.75      0.75       154
```

Output 15.9. Classification report of the balanced model.

The classification report (Output 15.9) confirms a test set accuracy of 0.750, with the negative class achieving a precision of 0.82 and recall of 0.78, yielding an F1-score of 0.80. The positive diabetic class achieves a precision of 0.63,

recall of 0.69, and an F1-score of 0.65. The disparity in F1-score between the two classes - 0.80 for the negative class against 0.65 for the positive class - is consistent with the dataset's class imbalance, where the larger negative class contains more training examples and results in more confident and accurate predictions for that group. The macro-averaged F1-score of 0.73 and weighted average F1-score of 0.75 indicate balanced but modest overall classification performance on the held-out set.

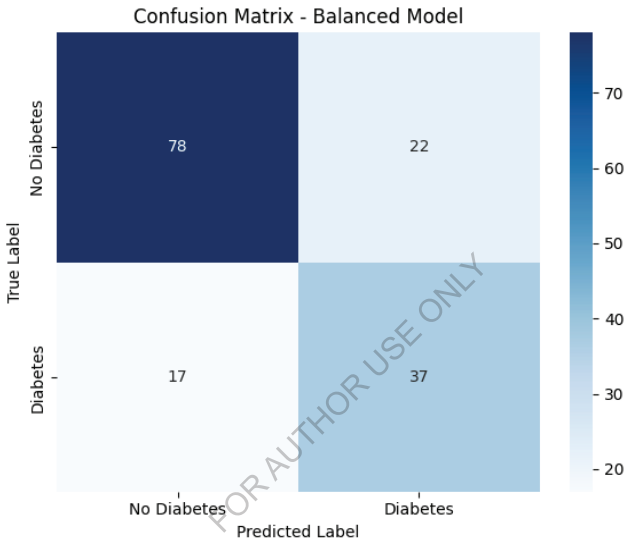


Figure 15.9. Confusion matrix of balanced neural network model.

Figure 15.9 presents the confusion matrix for the balanced model evaluated on the held-out test set of 154 samples. The model correctly classifies 78 of 100 negative cases and 37 of 54 positive cases, yielding 22 false positives and 17 false negatives. The false negative count of 17 is identical to that of the tuned Decision Tree, indicating that both models miss the same number of true diabetic cases on this test set despite their different model architectures. The higher false positive count of 22 relative to the Decision Tree's 16 suggests that the balanced model predicts the positive class more frequently, flagging more negative cases as diabetic.

15.7. Conclusion

This chapter investigated two complementary dimensions of practical machine learning deployment: the selection of an appropriate model from a set of classical classifiers, and the design of a neural network architecture that generalizes effectively without underfitting or overfitting. Both experiments were conducted on the Pima Indians Diabetes dataset, a structured binary classification task comprising 768 clinical samples, using identical preprocessing in both experiments.

In the Model Selection Strategy experiment, four classical classifiers - Logistic Regression, Decision Tree, Random Forest, and SVM - were evaluated under baseline and GridSearchCV-tuned configurations, using F1-score as the optimization metric. As summarized in Table 15.1, the tuned Decision Tree achieved the strongest overall result with an accuracy of 0.786, recall of 0.685, and F1-score of 0.692, while requiring only 2.6 seconds of hyperparameter search time. Random Forest, despite its substantially greater tuning cost of approximately 150 seconds, returned a lower F1-score of 0.620 - showing that increased model complexity did not improve generalization on this dataset of modest size.

FOR AUTHOR USE ONLY

In the Architecture Design Philosophy experiment, three multilayer perceptron architectures were constructed to explicitly instantiate the three regimes of the bias-variance tradeoff. As summarized in Table 15.2, the underfit model converged to a validation accuracy of 0.7724 with a negligible train-validation gap of 0.0015, reflecting a capacity ceiling rather than successful generalization. The overfit model achieved a perfect training accuracy of 1.0 while validation accuracy stabilized at 0.7642, a divergence of 23.6 percentage points that was further evidenced by the validation loss climbing past 3.0 by epoch 100 - indicating substantial overfitting. The regularized balanced model achieved the highest internal validation accuracy of 0.8211 with no runaway divergence, indicating that regularization improved generalization.

Comparing the best-performing models from each experiment yields the most practically significant finding of this chapter. Despite the balanced neural network's superior internal validation accuracy during training, the tuned Decision Tree outperforms it on the held-out test set with an accuracy of 0.786 versus 0.750, precision of 0.698 versus 0.630, and F1-score of 0.692 versus 0.650. Both models produce an identical false negative count of 17 on the 154-

sample test set, indicating similar sensitivity to diabetic cases. The neural network additionally generates six more false positives than the Decision Tree - 22 versus 16 - resulting in more false positive predictions.

On small, structured tabular datasets, Decision Trees naturally align with threshold-based clinical variables and can provide strong predictive performance with low computational cost. Limited training data also restrict the effectiveness of larger neural networks, even when regularization techniques are applied. In contrast, neural networks are better suited to large datasets or problems involving complex, unstructured data where their representational capacity can be fully exploited. Overall, these findings indicate that model complexity should be matched to the structure, scale, and characteristics of the available data.

FOR AUTHOR USE ONLY

References

Arjovsky, M., Chintala, S., & Bottou, L. (2017). Wasserstein GAN. In Proceedings of the 34th International Conference on Machine Learning (Vol. 70, pp. 214–223). PMLR.

Asrani, S. K., Devarbhavi, H., Eaton, J., & Kamath, P. S. (2019). Burden of liver diseases in the world. Journal of Hepatology, 70(1), 151–171. https://doi.org/10.1016/j.jhep.2018.09.014

Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks, 5(2), 157–166. https://doi.org/10.1109/72.279181

Bishop, C. M. (2006). Pattern recognition and machine learning. Springer.

Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: Synthetic minority over-sampling technique. Journal of Artificial Intelligence Research, 16, 321–357. https://doi.org/10.1613/jair.953

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv. https://doi.org/10.48550/arXiv.1406.1078

Chollet, F. (2015). Keras [Computer software]. GitHub. https://github.com/fchollet/keras

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009). ImageNet: A large-scale hierarchical image database. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 248–255). IEEE. https://doi.org/10.1109/CVPR.2009.5206848

De Vito, S., Massera, E., Piga, M., Martinotto, L., & Di Francia, G. (2008). On field calibration of an electronic nose for benzene estimation in an urban pollution monitoring scenario. Sensors and Actuators B: Chemical, 129(2), 750–757. https://doi.org/10.1016/j.snb.2007.09.060

Domingos, P. (2012). A few useful things to know about machine learning. Communications of the ACM, 55(10), 78–87. https://doi.org/10.1145/2347736.2347755

FOR AUTHOR USE ONLY

Esteva, A., Kuprel, B., Novoa, R. A., Ko, J., Swetter, S. M., Blau, H. M., & Thrun, S. (2017). Dermatologist-level classification of skin cancer with deep neural networks. Nature, 542(7639), 115–118. https://doi.org/10.1038/nature21056

FastAPI. (n.d.). Bigger applications — multiple files; Concurrency and async/await. FastAPI Documentation. https://fastapi.tiangolo.com

Frankle, J., & Carlin, M. (2019). The lottery ticket hypothesis: Finding sparse, trainable neural networks. In Proceedings of the 7th International Conference on Learning Representations.

Frid-Adar, M., Diamant, I., Klang, E., Amitai, M., Goldberger, J., & Greenspan, H. (2018). GAN-based synthetic medical image augmentation for increased CNN performance in liver lesion classification. Neurocomputing, 321, 321–331. https://doi.org/10.1016/j.neucom.2018.09.013

Geman, S., Bienenstock, E., & Doursat, R. (1992). Neural networks and the bias/variance dilemma. Neural Computation, 4(1), 1–58. https://doi.org/10.1162/neco.1992.4.1.1

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (pp. 249–256). PMLR.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep learning. MIT Press.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative adversarial nets. In Advances in Neural Information Processing Systems (Vol. 27, pp. 2672–2680). Curran Associates.

Han, S., Pool, J., Tran, J., & Dally, W. J. (2015). Learning both weights and connections for efficient neural networks. In Advances in Neural Information Processing Systems (Vol. 28). Curran Associates.

Hastie, T., Tibshirani, R., & Friedman, J. (2009). The elements of statistical learning: Data mining, inference, and prediction. Springer.

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 (pp. 770–778). IEEE. https://doi.org/10.1109/CVPR.2016.90

FOR AUTHOR USE ONLY

He, K., Girshick, R., & Dollár, P. (2019). Rethinking ImageNet pre-training. In Proceedings of the IEEE/CVF International Conference on Computer Vision (pp. 4918–4927). IEEE. https://doi.org/10.1109/ICCV.2019.00502

Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., & Hochreiter, S. (2017). GANs trained by a two time-scale update rule converge to a local Nash equilibrium. In Advances in Neural Information Processing Systems (Vol. 30). Curran Associates.

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. Neural Computation, 9(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

Howard, J., & Ruder, S. (2018). Universal language model fine-tuning for text classification. In Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (pp. 328–339). ACL. https://doi.org/10.18653/v1/P18-1031

Isola, P., Zhu, J.-Y., Zhou, T., & Efros, A. A. (2017). Image-to-image translation with conditional adversarial networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 1125–1134). IEEE. https://doi.org/10.1109/CVPR.2017.632

Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., & Kalenichenko, D. (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 2704–2713). IEEE. https://doi.org/10.1109/CVPR.2018.00286

Jordan, M. I., & Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. Science, 349(6245), 255–260. https://doi.org/10.1126/science.aaa8415

Karras, T., Laine, S., & Aila, T. (2019). A style-based generator architecture for generative adversarial networks. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (pp. 4401–4410). IEEE. https://doi.org/10.1109/CVPR.2019.00453

Kermany, D. S., Goldbaum, M., Cai, W., Valentim, C. C. S., Liang, H., Baxter, S. L., McKeown, A., Yang, G., Wu, X., Yan, F., Dong, J., … Zhang, K. (2018). Identifying medical diagnoses and treatable diseases by image-based deep learning. Cell, 172(5), 1122–1131. https://doi.org/10.1016/j.cell.2018.02.010

FOR AUTHOR USE ONLY

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In Proceedings of the 3rd International Conference on Learning Representations.

Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. University of Toronto.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. Nature, 521(7553), 436–444. https://doi.org/10.1038/nature14539

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278–2324. https://doi.org/10.1109/5.726791

LeCun, Y., Denker, J. S., & Solla, S. A. (1990). Optimal brain damage. In Advances in Neural Information Processing Systems (Vol. 2, pp. 598–605). Morgan Kaufmann.

Lim, B., & Zohren, S. (2021). Time-series forecasting with deep learning: A survey. Philosophical Transactions of the Royal Society A, 379(2194), Article 20200209. https://doi.org/10.1098/rsta.2020.0209

Lipton, Z. C., Berkowitz, J., & Elkan, C. (2015). A critical review of recurrent neural networks for sequence learning. arXiv. https://doi.org/10.48550/arXiv.1506.00019

Litjens, G., Kooi, T., Bejnordi, B. E., Setio, A. A. A., Ciompi, F., Ghafoorian, M., van der Laak, J. A. W. M., van Ginneken, B., & Sánchez, C. I. (2017). A survey on deep learning in medical image analysis. Medical Image Analysis, 42, 60–88. https://doi.org/10.1016/j.media.2017.07.005

Maas, A. L., Daly, R. E., Pham, P. T., Huang, C., 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 (pp. 142–150). ACL.

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted Boltzmann machines. In Proceedings of the 27th International Conference on Machine Learning (pp. 807–814). Omnipress.

ONNX Runtime. (n.d.). Get started. ONNX Runtime Documentation. https://onnxruntime.ai/docs/get-started

ONNX Runtime. (n.d.). WebGPU execution provider. ONNX Runtime Web Documentation.

FOR AUTHOR USE ONLY

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825–2830.

PyTorch. (n.d.). TorchScript. PyTorch Documentation. https://pytorch.org/docs/stable/jit.html

Radford, A., Metz, L., & Chintala, S. (2016). Unsupervised representation learning with deep convolutional generative adversarial networks. In Proceedings of the 4th International Conference on Learning Representations.

Raghu, M., Zhang, C., Kleinberg, J., & Bengio, S. (2019). Transfusion: Understanding transfer learning for medical imaging. In Advances in Neural Information Processing Systems (Vol. 32). Curran Associates.

Raina, R., Madhavan, A., & Ng, A. Y. (2009). Large-scale deep unsupervised learning using graphics processors. In Proceedings of the 26th Annual International Conference on Machine Learning (pp. 873–880). ACM. https://doi.org/10.1145/1553374.1553486

Rajpurkar, P., Irvin, J., Ball, R. L., Zhu, K., Yang, B., Mehta, H., Duan, T., Ding, D., Bagul, A., Langlotz, C., Shpanskaya, K., Lungren, M. P., & Ng, A. Y. (2017). CheXNet: Radiologist-level pneumonia detection on chest X-rays with deep learning. arXiv.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review, 65(6), 386–408. https://doi.org/10.1037/h0042519

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. Nature, 323(6088), 533–536. https://doi.org/10.1038/323533a0

Smith, J. W., Everhart, J. E., Dickson, W. C., Knowler, W. C., & Johannes, R. S. (1988). Using the ADAP learning algorithm to forecast the onset of diabetes mellitus. Proceedings of the Symposium on Computer Applications in Medical Care, 261–265.

Sokolova, M., & Lapalme, G. (2009). A systematic analysis of performance measures for classification tasks. Information Processing & Management, 45(4), 427–437. https://doi.org/10.1016/j.ipm.2009.03.002

FOR AUTHOR USE ONLY

Srivastava, S. (n.d.). Liver patient dataset: Exploring liver health through clinical biomarkers [Data set]. Kaggle.

Tajbakhsh, N., Shin, J. Y., Gurudu, S. R., Hurst, R. T., Kendall, C. B., Gotway, M. B., & Liang, J. (2016). Convolutional neural networks for medical image analysis: Full training or fine tuning? IEEE Transactions on Medical Imaging, 35(5), 1299–1312. https://doi.org/10.1109/TMI.2016.2535302

TensorFlow. (n.d.). Post-training quantization. TensorFlow Lite Guide.

TensorFlow. (n.d.). The Functional API. TensorFlow Guide. https://www.tensorflow.org/guide/keras/functional_api

TensorFlow. (n.d.). Transfer learning and fine-tuning. TensorFlow Tutorials.

TensorFlow. (n.d.). Using the SavedModel format. TensorFlow Guide.

Yosinski, J., Clune, J., Bengio, Y., & Lipson, H. (2014). How transferable are features in deep neural networks? In Advances in Neural Information Processing Systems (Vol. 27). Curran Associates.

FOR AUTHOR USE ONLY

Acknowledgement of AI-Assisted Editing:

Generative AI (OpenAI ChatGPT) was used solely to assist with language editing, stylistic refinement, consistency checking, and the preparation and enhancement of selected figures and tables. The author independently reviewed, verified, and approved all content and assumes full responsibility for the final manuscript.



FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY



Buy your books fast and straightforward online - at one of world's fastest growing online book stores! Environmentally sound due to Print-on-Demand technologies.

Buy your books online at
www.morebooks.shop

Kaufen Sie Ihre Bücher schnell und unkompliziert online – auf einer der am schnellsten wachsenden Buchhandelsplattformen weltweit! Dank Print-On-Demand umwelt- und ressourcenschonend produzi ert.

Bücher schneller online kaufen
www.morebooks.shop

info@omniscriptum.com
www.omniscriptum.com

OMNIScriptum

FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY

FOR AUTHOR USE ONLY