Deep Learning Architectures for Scientific Machine Learning

A Comprehensive PyTorch Guide

Professor Vectorex Gradiens

Wednesday 3rd December, 2025

Version 1.0

Disclaimer

Important Notice: Apart from this paragraph, every single word in this document was written with the assistance of Large Language Models. This document is intended as an aid for learning PyTorch, in particular for implementing deep learning architectures presented in the course "AI in the Sciences and Engineering" held at ETH Zürich.

The content is extremely likely to contain errors, so use it with care, knowing that this isn't reviewed or approved by domain experts. Always verify critical implementations and consult primary sources for production use.

Giovanni Guidarini, December 2025

Acknowledgments: This textbook builds upon the collective wisdom of the PyTorch community, scientific machine learning researchers, and educators worldwide. Special thanks to the developers of PyTorch, the authors of foundational papers in deep learning, and students whose questions shaped this material.

Preface

My dear student,

Welcome. It is a distinct pleasure to guide you on this journey. You are already embedded in the world of scientific computing, a field of precision, rigor, and computational elegance. You will find that deep learning is not so different. At its heart, it is a powerful form of high-dimensional function approximation, built on the familiar foundations of linear algebra and calculus. The "magic" is simply that we have found a way to make these function approximators (*neural networks*) trainable on a massive scale.

Your lack of PyTorch experience is not a hindrance; it is an opportunity. You arrive with no bad habits. We will build your knowledge from the ground up, with the same care one takes in formulating a proof or designing a robust simulation. PyTorch is our "language" for this. It is expressive, powerful, and, once you are fluent, a genuine joy to use.

This textbook is comprehensive, spanning from the fundamental `torch.Tensor` to state-of-the-art architectures like Transformers and Neural Operators. Each chapter builds systematically, combining rigorous theory with practical implementation. You will not merely learn to use PyTorch—you will understand *why* it works the way it does, and you will develop the intuition to design your own architectures for scientific problems.

How to Use This Book:

- **Theory sections** provide mathematical foundations and intuition. Do not skip these—they are the bedrock upon which everything rests.
- **Implementation sections** show you the "PyTorch way"—idioms, best practices, and common pitfalls. Type out the code yourself; reading is not enough.
- **Exercises** range from elementary to research-level. Attempt all of them. Struggle is where learning happens.
- **Colored boxes** highlight key insights:
  - Professor's Notes for expert intuition
  - PyTorch Idioms for best practices
  - Warnings for common mistakes
  - Expert Tricks for advanced techniques
  - Scientific Applications showing real-world use

Treat this not just as a technical exercise, but as an art. The code we write, like the LaTeX that renders this page, can and should be clean, precise, and beautiful.

Let us begin.

Professor Vectorex Gradiens
ETH Zürich
December 2025

Contents

Preface **3**

I Foundations of PyTorch **7**

1 The `torch.Tensor` **7**
1.1 Introduction: Why Tensors? . . . 7
1.2 Theory: Tensor Fundamentals . . . 7
1.3 Implementation: Creating and Manipulating Tensors . . . 8
1.3.1 Creating Tensors . . . 8
1.3.2 Tensor Properties . . . 9
1.3.3 Indexing and Slicing . . . 9
1.3.4 Shape Manipulation . . . 9
1.3.5 Broadcasting . . . 11
1.3.6 Mathematical Operations . . . 11
1.3.7 Moving Tensors Between Devices . . . 13
1.4 Exercises . . . 13

2 Autograd: Automatic Differentiation **16**
2.1 Introduction: Why Autograd Matters . . . 16
2.2 Theory: Computational Graphs and Backpropagation . . . 16
2.2.1 The Computational Graph . . . 16
2.2.2 Backpropagation: The Chain Rule . . . 16
2.2.3 Dynamic Computation Graphs . . . 17
2.3 Implementation: Using Autograd in Practice . . . 17
2.3.1 Basic Gradient Computation . . . 17
2.3.2 Leaf vs Non-Leaf Tensors . . . 17
2.3.3 Gradient Accumulation and Zeroing . . . 19
2.3.4 Vector-Jacobian Products . . . 19
2.3.5 Disabling Gradient Tracking . . . 20
2.3.6 In-Place Operations . . . 21
2.3.7 Retaining Graphs . . . 21
2.3.8 Higher-Order Derivatives . . . 21
2.4 Implementation: Common Patterns and Debugging . . . 22
2.4.1 Checking Gradients . . . 22
2.4.2 Gradient Flow Debugging . . . 23
2.4.3 Custom Autograd Functions . . . 24
2.5 Exercises . . . 24

3 Building Models with `nn.Module` **27**
3.1 Introduction: Why `nn.Module`? . . . 27
3.2 Theory: The Module Hierarchy . . . 27
3.3 Implementation: Creating Your First Module . . . 27
3.3.1 A Simple Linear Model . . . 27
3.3.2 Accessing Parameters . . . 28
3.3.3 Using `nn.Sequential` . . . 28
3.3.4 Custom Modules with Multiple Layers . . . 30
3.3.5 Parameters vs Buffers . . . 30
3.3.6 Training vs Evaluation Mode . . . 31
3.3.7 Moving Models to GPU . . . 32
3.3.8 Saving and Loading Models . . . 32
3.3.9 Parameter Initialization . . . 33

3.3.10 Module Hooks for Debugging . . . 35
3.4 Implementation: Practical Patterns . . . 35
3.4.1 Freezing Layers . . . 35
3.4.2 Multiple Inputs/Outputs . . . 36
3.4.3 Custom Layer with Parameters . . . 36
3.5 Exercises . . . 37

4 The Training Loop & Optimizers 40
4.1 Introduction: The Training Process . . . 40
4.2 Theory: Gradient Descent and Optimization . . . 40
4.2.1 The Optimization Problem . . . 40
4.2.2 Gradient Descent . . . 40
4.2.3 Learning Rate . . . 40
4.3 Implementation: The Standard Training Loop . . . 41
4.3.1 Complete Training Example . . . 41
4.3.2 Loss Functions . . . 43
4.3.3 Optimizers . . . 44
4.3.4 Learning Rate Schedules . . . 45
4.3.5 Gradient Clipping . . . 45
4.3.6 Validation Loop . . . 47
4.3.7 Early Stopping . . . 48
4.4 Implementation: Debugging Training . . . 49
4.4.1 When Training Goes Wrong . . . 49
4.4.2 Monitoring Training . . . 50
4.5 Exercises . . . 51

5 Data Loading & Preprocessing 54
5.1 Introduction: Why Proper Data Loading Matters . . . 54
5.2 Theory: Dataset and DataLoader Architecture . . . 54
5.3 Implementation: Basic Usage . . . 54
5.3.1 Using Built-in Datasets . . . 54
5.3.2 Creating Custom Datasets . . . 56
5.3.3 Dataset from Files . . . 56
5.3.4 Dataset for Time Series . . . 57
5.3.5 Dataset for Point Clouds . . . 59
5.3.6 Data Transforms . . . 59
5.3.7 Normalization Strategies . . . 61
5.3.8 Handling Variable-Length Sequences . . . 62
5.3.9 Train/Validation/Test Split . . . 64
5.3.10 Efficient Data Loading . . . 64
5.4 Exercises . . . 66

II Deep Learning Architectures 68

6 Fully Connected Networks (MLPs) 68
6.1 Introduction: The Foundation of Deep Learning . . . 68
6.2 Theory: Understanding MLPs . . . 68
6.2.1 Architecture . . . 68
6.2.2 Depth vs Width . . . 69
6.2.3 Activation Functions . . . 70
6.2.4 Why Networks Fail to Train . . . 72
6.3 Implementation: Building MLPs in PyTorch . . . 73
6.3.1 Simple MLP with Sequential . . . 73
6.3.2 Custom MLP Module . . . 73

6.3.3 Weight Initialization . . . 75
6.3.4 Batch Normalization . . . 77
6.3.5 Layer Normalization . . . 78
6.3.6 Dropout . . . 79
6.3.7 Complete MLP with All Techniques . . . 79
6.4 Implementation: Debugging MLPs . . . 81
6.4.1 Checking for Dead ReLUs . . . 81
6.4.2 Visualizing Activation Distributions . . . 81
6.4.3 Gradient Flow Visualization . . . 82
6.5 Exercises . . . 84
6.6 Key Takeaways . . . 87

7 Convolutional Neural Networks (CNNs) 88
7.1 Introduction: Why Convolutions? . . . 88
7.2 Theory: How Convolutions Work . . . 88
7.2.1 The Convolution Operation . . . 88
7.2.2 Parameter Sharing . . . 89
7.2.3 Receptive Fields . . . 90
7.2.4 Output Size Calculation . . . 90
7.2.5 Pooling . . . 91
7.3 Implementation: Building CNNs in PyTorch . . . 92
7.3.1 Conv2d: The Core Layer . . . 92
7.3.2 Shape Calculation Practice . . . 92
7.3.3 Building a Simple CNN . . . 94
7.3.4 Modern CNN with Batch Normalization . . . 94
7.3.5 1D Convolutions for Time Series . . . 97
7.3.6 3D Convolutions for Volumetric Data . . . 97
7.3.7 Transposed Convolutions (Upsampling) . . . 99
7.3.8 Dilated Convolutions . . . 100
7.3.9 Common CNN Architectures . . . 101
7.4 Implementation: Debugging CNNs . . . 101
7.4.1 Visualizing Feature Maps . . . 101
7.4.2 Common CNN Debugging Issues . . . 102
7.5 Exercises . . . 104
7.6 Key Takeaways . . . 107

8 Residual Networks & Skip Connections 108
8.1 Introduction: The Deep Network Problem . . . 108
8.2 Theory: Why Skip Connections Work . . . 108
8.2.1 The Problem: Vanishing Gradients . . . 108
8.2.2 The Solution: Residual Connections . . . 108
8.2.3 Gradient Flow Through Skip Connections . . . 109
8.2.4 Residual Block Variants . . . 110
8.2.5 When Dimensions Don't Match . . . 111
8.3 Implementation: Building Residual Blocks . . . 111
8.3.1 Basic Residual Block . . . 111
8.3.2 Pre-Activation Residual Block . . . 113
8.3.3 Bottleneck Block . . . 113
8.3.4 Building a Complete ResNet . . . 116
8.3.5 Dense Connections (DenseNet) . . . 118
8.3.6 Skip Connections in Practice . . . 119
8.4 Implementation: Debugging ResNets . . . 120
8.4.1 Verifying Skip Connections . . . 120
8.4.2 Comparing With and Without Skip Connections . . . 120
8.5 Exercises . . . 121

8.6 Key Takeaways . . . 124

9 Batch Normalization & Layer Normalization 126
9.1 Introduction: Why Normalization Matters . . . 126
9.2 Theory: Batch Normalization . . . 126
9.2.1 The Problem: Internal Covariate Shift . . . 126
9.2.2 How Batch Normalization Works . . . 126
9.2.3 Why Batch Normalization Helps . . . 128
9.2.4 Limitations of Batch Normalization . . . 128
9.3 Theory: Layer Normalization . . . 128
9.4 Implementation: Using Normalization in PyTorch . . . 130
9.4.1 Batch Normalization Usage . . . 130
9.4.2 Layer Normalization Usage . . . 131
9.4.3 Other Normalization Variants . . . 132
9.4.4 Implementing Batch Norm from Scratch . . . 132
9.4.5 Common Mistakes and Debugging . . . 134
9.5 Exercises . . . 136
9.6 Key Takeaways . . . 139

10 Recurrent Neural Networks (RNNs) 141
10.1 Introduction: Processing Sequential Data . . . 141
10.2 Theory: How RNNs Work . . . 141
10.2.1 The Basic RNN . . . 141
10.2.2 Parameter Sharing . . . 143
10.2.3 Sequence Modeling Patterns . . . 143
10.2.4 The Vanishing Gradient Problem . . . 143
10.3 Theory: Long Short-Term Memory (LSTM) . . . 144
10.3.1 LSTM Architecture . . . 144
10.4 Theory: Gated Recurrent Unit (GRU) . . . 146
10.4.1 GRU Architecture . . . 146
10.5 Implementation: RNNs in PyTorch . . . 146
10.5.1 Vanilla RNN . . . 146
10.5.2 LSTM . . . 148
10.5.3 GRU . . . 148
10.5.4 Building a Sequence Classifier . . . 148
10.5.5 Sequence-to-Sequence (Many-to-Many) . . . 150
10.5.6 Bidirectional RNNs . . . 150
10.5.7 Handling Variable-Length Sequences . . . 152
10.5.8 Gradient Clipping (Essential for RNNs) . . . 153
10.5.9 Training Tips for RNNs . . . 153
10.5.10 Common RNN Architectures . . . 155
10.5.11 Time Series Forecasting Example . . . 157
10.6 Exercises . . . 158
10.7 Key Takeaways . . . 162

11 Attention & Transformers 164
11.1 Introduction: Beyond RNNs . . . 164
11.2 Theory: The Attention Mechanism . . . 164
11.2.1 Intuition: What is Attention? . . . 164
11.2.2 Mathematical Formulation . . . 164
11.2.3 Self-Attention vs Cross-Attention . . . 166
11.2.4 Multi-Head Attention . . . 166
11.2.5 Positional Encoding . . . 166
11.3 Theory: The Transformer Architecture . . . 168
11.3.1 Complete Architecture . . . 168

11.3.2 Why Transformers Work So Well . . . 168
11.4 Implementation: Building Attention from Scratch . . . 170
11.4.1 Scaled Dot-Product Attention . . . 170
11.4.2 Multi-Head Attention . . . 171
11.4.3 Positional Encoding . . . 174
11.4.4 Feed-Forward Network . . . 175
11.4.5 Complete Transformer Encoder Block . . . 175
11.4.6 Using PyTorch's Built-in Attention . . . 177
11.4.7 Masking for Causal Attention . . . 177
11.4.8 Complete Transformer for Classification . . . 178
11.4.9 Transformer for Sequence-to-Sequence . . . 180
11.5 Exercises . . . 183
11.6 Key Takeaways . . . 188

III Practical Skills **190**

12 Debugging & Best Practices **190**
12.1 Introduction: Why Debugging Matters . . . 190
12.2 Common Bugs and How to Fix Them . . . 190
12.2.1 Bug 1: Loss is NaN . . . 190
12.2.2 Bug 2: Loss Not Decreasing . . . 192
12.2.3 Bug 3: Training Loss Decreases, Validation Doesn't . . . 194
12.2.4 Bug 4: Model Works on Small Data, Fails on Full Dataset . . . 195
12.3 Debugging Strategies . . . 196
12.3.1 Start Simple, Add Complexity . . . 196
12.3.2 Check Shapes at Every Step . . . 196
12.3.3 Visualize Gradients . . . 197
12.3.4 Use Hooks for Debugging . . . 197
12.4 Best Practices for Development . . . 198
12.4.1 Code Organization . . . 198
12.4.2 Configuration Management . . . 199
12.4.3 Experiment Tracking . . . 199
12.4.4 Reproducibility . . . 201
12.4.5 Checkpointing . . . 201
12.5 Debugging Checklist . . . 202
12.6 Key Takeaways . . . 203

13 Training Best Practices **204**
13.1 Introduction: From Working to Working Well . . . 204
13.2 Hyperparameter Tuning . . . 204
13.2.1 Which Hyperparameters Matter Most? . . . 204
13.2.2 Learning Rate Strategies . . . 204
13.2.3 Batch Size Selection . . . 207
13.2.4 Model Architecture Selection . . . 207
13.3 Regularization Strategies . . . 208
13.3.1 Dropout . . . 208
13.3.2 Weight Decay (L2 Regularization) . . . 208
13.3.3 Data Augmentation . . . 208
13.4 Training Monitoring . . . 209
13.4.1 What to Track . . . 209
13.4.2 Recognizing Training Issues from Curves . . . 210
13.5 Model Selection and Evaluation . . . 211
13.5.1 Proper Train/Val/Test Split . . . 211
13.5.2 Cross-Validation for Small Datasets . . . 211

13.6 Practical Tips . . . 213
13.6.1 Start with Good Defaults . . . 213
13.6.2 Hyperparameter Search Strategies . . . 213
13.7 Key Takeaways . . . 214

14 Performance & Optimization **216**
14.1 Introduction: Making Training Faster . . . 216
14.2 Mixed Precision Training . . . 216
14.2.1 Automatic Mixed Precision (AMP) . . . 216
14.2.2 Gradient Checkpointing . . . 217
14.3 GPU Optimization . . . 218
14.3.1 Data Transfer Optimization . . . 218
14.3.2 Batch Size and GPU Utilization . . . 218
14.3.3 Multi-GPU Training . . . 218
14.4 DataLoader Optimization . . . 220
14.4.1 Optimal Number of Workers . . . 220
14.4.2 Prefetching . . . 220
14.5 Memory Management . . . 220
14.5.1 Monitoring Memory Usage . . . 220
14.5.2 Reducing Memory Usage . . . 221
14.6 Profiling and Bottleneck Analysis . . . 223
14.6.1 PyTorch Profiler . . . 223
14.6.2 Simple Timing . . . 223
14.6.3 Identify Bottlenecks . . . 223
14.7 Production Deployment . . . 226
14.7.1 Model Export . . . 226
14.7.2 Inference Optimization . . . 226
14.8 Key Takeaways . . . 227

IV Integrative Challenges **229**

15 Grand Challenges **229**
15.1 Introduction: Putting It All Together . . . 229
15.2 Challenge 1: Time Series Forecasting System . . . 229
15.3 Challenge 2: Hybrid Vision-Language Model . . . 231
15.4 Challenge 3: Denoising Scientific Data . . . 234
15.5 Challenge 4: Custom Transformer for Sequence Tasks . . . 238
15.6 Challenge 5: End-to-End ML Pipeline . . . 239
15.7 Evaluation and Next Steps . . . 240
15.8 Final Thoughts . . . 241

Part I

Foundations of PyTorch

1 The `torch.Tensor`

1.1 Introduction: Why Tensors?

Everything in PyTorch—from your input data (images, time series, simulation meshes) to the parameters of your neural network (weights and biases)—is represented as a **tensor**.

If you've used NumPy, you already know the core idea: a tensor is essentially a multi-dimensional array (like `ndarray`). So why use PyTorch instead of NumPy?

Two critical reasons:

1. **GPU Acceleration:** PyTorch tensors can be moved to GPUs with a single command, enabling massive parallelization. For deep learning and large-scale scientific computing, this isn't optional—it's essential.
2. **Automatic Differentiation:** PyTorch can automatically compute gradients of any output with respect to any input. This `autograd` system is the engine that makes deep learning possible. We'll explore this in detail in Section 2.

1.2 Theory: Tensor Fundamentals

Tensor

A **tensor** is a multi-dimensional array of numerical values. It generalizes:

- **Scalar** (0D tensor): a single number
- **Vector** (1D tensor): an array of numbers
- **Matrix** (2D tensor): a 2D grid of numbers
- **Higher-dimensional tensors**: 3D, 4D, ... arrays

Key Properties:

- **Shape:** The dimensions of the tensor, e.g., `(3, 4, 5)` means 3×4×5
- **Dtype:** The data type (e.g., `float32`, `int64`, `bool`)
- **Device:** Where the tensor lives (`cpu` or `cuda:0`, `cuda:1`, etc.)
- **Requires_grad:** Whether to track operations for automatic differentiation

Common Tensor Shapes in Deep Learning:

Shape	Typical Use
`(N,)`	Vector of N values
`(N, D)`	Batch of N samples, D features each
`(N, C, H, W)`	Batch of N images: C channels, H×W pixels
`(N, L, D)`	Batch of N sequences, length L, D features
`(N, C, D, H, W)`	Batch of N 3D volumes

Table 1: Common tensor shapes. N=batch size, C=channels, H/W=height/width, L=sequence length, D=feature dimension

1.3 Implementation: Creating and Manipulating Tensors

1.3.1 Creating Tensors

```python
import torch

# From Python lists
x = torch.tensor([1, 2, 3])
print(x)  # tensor([1, 2, 3])

# Specify dtype
x = torch.tensor([1.0, 2.0, 3.0], dtype=torch.float32)
print(x.dtype)  # torch.float32

# Common initialization functions
zeros = torch.zeros(2, 3)            # 2x3 matrix of zeros
ones = torch.ones(2, 3)              # 2x3 matrix of ones
random = torch.randn(2, 3)           # Random normal
    distribution
uniform = torch.rand(2, 3)           # Uniform distribution
    [0, 1)
arange = torch.arange(0, 10, 2)      # [0, 2, 4, 6, 8]
linspace = torch.linspace(0, 1, 5)   # [0.0, 0.25, 0.5, 0.75,
    1.0]

# Like existing tensor (same shape/dtype)
x = torch.ones(3, 4)
y = torch.zeros_like(x)  # Same shape as x
z = torch.randn_like(x)  # Same shape, random values
```

[Default Dtype] PyTorch's default floating point dtype is `torch.float32`. For most deep learning, this is fine. You can change it globally with:

```python
torch.set_default_dtype(torch.float64)  # More precision
```

However, `float32` is usually the sweet spot: enough precision, less memory, faster on GPUs.

1.3.2 Tensor Properties

```
x = torch.randn(2, 3, 4)

print(x.shape)          # torch.Size([2, 3, 4])
print(x.size())         # Same as .shape
print(x.ndim)           # 3 (number of dimensions)
print(x.numel())        # 24 (total number of elements)
print(x.dtype)          # torch.float32
print(x.device)         # cpu

# Check if tensor requires gradients
print(x.requires_grad)  # False by default
```

1.3.3 Indexing and Slicing

PyTorch indexing works like NumPy:

```
x = torch.arange(12).reshape(3, 4)
# tensor([[ 0,  1,  2,  3],
#         [ 4,  5,  6,  7],
#         [ 8,  9, 10, 11]])

# Basic indexing
print(x[0])         # First row: tensor([0, 1, 2, 3])
print(x[:, 0])      # First column: tensor([0, 4, 8])
print(x[1, 2])      # Element at (1, 2): tensor(6)

# Slicing
print(x[:2, :2])  # Top-left 2x2:
# tensor([[0, 1],
#         [4, 5]])

# Advanced indexing
indices = torch.tensor([0, 2])
print(x[indices])  # Rows 0 and 2

# Boolean masking
mask = x > 5
print(x[mask])      # tensor([ 6,  7,  8,  9, 10, 11])
```

[Views vs Copies] Slicing creates a **view**, not a copy! Modifying a view changes the original:

```
x = torch.ones(3, 3)
y = x[0]        # View of first row
y[0] = 999      # Modifies x!
print(x)        # First element is now 999

# To create a copy:
y = x[0].clone()
```

1.3.4 Shape Manipulation

```
x = torch.arange(12)

# Reshape (must have same total elements)
y = x.reshape(3, 4)
z = x.view(2, 6)        # Similar to reshape
```

```
w = x.reshape(3, -1)  # -1 means "infer this dimension" -> (3, 4)

# Transpose
x = torch.randn(2, 3)
y = x.T               # Shape: (3, 2)
z = x.transpose(0, 1) # Same as .T

# Permute (generalized transpose)
x = torch.randn(2, 3, 4)
y = x.permute(2, 0, 1)  # Shape: (4, 2, 3)

# Flatten
x = torch.randn(2, 3, 4)
y = x.flatten()           # Shape: (24,)
z = x.flatten(1)          # Flatten from dim 1: (2, 12)

# Squeeze and unsqueeze (remove/add dimensions of size 1)
x = torch.randn(2, 1, 3, 1)
y = x.squeeze()           # Shape: (2, 3)
z = x.squeeze(1)          # Remove dim 1: (2, 3, 1)

x = torch.randn(2, 3)
y = x.unsqueeze(0)        # Shape: (1, 2, 3)
z = x.unsqueeze(-1)       # Shape: (2, 3, 1)
```

[Understanding Dimensions]

- Dimension indices start at 0
- Negative indices count from the end: -1 is the last dimension
- `unsqueeze(1)` adds a dimension at position 1
- `squeeze()` removes ALL dimensions of size 1

1.3.5 Broadcasting

Broadcasting is one of the most powerful—and confusing—features. It allows operations between tensors of different shapes.

Broadcasting Rules:

1. If tensors have different number of dimensions, prepend 1s to the smaller one
2. Dimensions are compatible if they're equal or one of them is 1
3. The result shape is the maximum along each dimension

```
# Example 1: Vector + Scalar
x = torch.tensor([1, 2, 3])  # Shape: (3,)
y = 10                       # Shape: () - scalar
z = x + y                    # Shape: (3,) - broadcasts
    scalar to (3,)
# Result: tensor([11, 12, 13])

# Example 2: Matrix + Vector (row-wise)
x = torch.ones(3, 4)         # Shape: (3, 4)
y = torch.tensor([1, 2, 3, 4])  # Shape: (4,)
z = x + y                    # y broadcasts to (1, 4) then
    (3, 4)
# Each row of x has y added to it

# Example 3: Matrix + Column Vector
x = torch.ones(3, 4)         # Shape: (3, 4)
y = torch.tensor([[1], [2], [3]])  # Shape: (3, 1)
z = x + y                    # y broadcasts to (3, 4)
# Each column of x has corresponding y value added

# Example 4: Batch operations
x = torch.randn(10, 3, 4)    # 10 matrices of size 3x4
y = torch.randn(3, 4)        # Single matrix
z = x + y                    # y broadcasts to (10, 3, 4)
# Same matrix y added to all 10 matrices in x
```

[Common Broadcasting Mistake]

```
# Trying to add vectors of different sizes
x = torch.ones(3)      # Shape: (3,)
y = torch.ones(4)      # Shape: (4,)
z = x + y              # ERROR! Shapes don't match

# Need to explicitly reshape:
x = x.unsqueeze(1)     # Shape: (3, 1)
y = y.unsqueeze(0)     # Shape: (1, 4)
z = x + y              # Shape: (3, 4) - outer sum
```

1.3.6 Mathematical Operations

```
x = torch.tensor([1.0, 2.0, 3.0])
y = torch.tensor([4.0, 5.0, 6.0])

# Element-wise operations
z = x + y           # Addition
z = x - y           # Subtraction
z = x * y           # Multiplication
z = x / y           # Division
z = x ** 2          # Power
z = torch.sqrt(x)   # Square root
```

```
z = torch.exp(x)    # Exponential
z = torch.log(x)    # Natural log

# In-place operations (modify the tensor)
x.add_(5)           # Add 5 to x in-place
x.mul_(2)           # Multiply x by 2 in-place

# Matrix operations
A = torch.randn(3, 4)
B = torch.randn(4, 5)
C = torch.mm(A, B)              # Matrix multiply: (3, 5)
C = torch.matmul(A, B)          # Same, but works for batches
    too
C = A @ B                       # Python 3.5+ operator

# Batch matrix multiply
A = torch.randn(10, 3, 4)       # 10 matrices of size 3x4
B = torch.randn(10, 4, 5)       # 10 matrices of size 4x5
C = torch.bmm(A, B)             # Shape: (10, 3, 5)

# Reduction operations
x = torch.randn(3, 4)
mean = x.mean()                 # Scalar: mean of all elements
std = x.std()                   # Standard deviation
sum_all = x.sum()               # Sum of all elements
sum_dim0 = x.sum(dim=0)         # Sum along dimension 0: (4,)
sum_dim1 = x.sum(dim=1)         # Sum along dimension 1: (3,)
max_val, max_idx = x.max(dim=1)  # Max value and index per
    row
```

[Dimension Conventions] When reducing along a dimension:

- `dim=0`: Operate across rows (result has fewer rows)
- `dim=1`: Operate across columns (result has fewer columns)
- `dim=-1`: Last dimension (most common for sequences)
- `keepdim=True`: Keep the dimension (size 1) in result

1.3.7 Moving Tensors Between Devices

```python
# Check if CUDA is available
if torch.cuda.is_available():
    device = torch.device("cuda")
    print(f"Using GPU: {torch.cuda.get_device_name(0)}")
else:
    device = torch.device("cpu")
    print("Using CPU")

# Create tensor on CPU
x = torch.randn(1000, 1000)

# Move to GPU
x_gpu = x.to(device)
# Or:
x_gpu = x.cuda()  # Shorthand for .to('cuda')

# Move back to CPU
x_cpu = x_gpu.to('cpu')
# Or:
x_cpu = x_gpu.cpu()

# Operations must be on same device
y_gpu = torch.randn(1000, 1000, device=device)  # Create directly on GPU
z = x_gpu + y_gpu  # OK: both on GPU

# This would ERROR:
# z = x_cpu + y_gpu  # ERROR! Different devices
```

[Device Mismatch] One of the most common errors:

```
RuntimeError: Expected all tensors to be on the same device
```

Always ensure tensors are on the same device before operations! Use `.to(device)` consistently.

1.4 Exercises

1.1: Tensor Creation - ★★

Goal: Get comfortable creating tensors in different ways.

1. Create a tensor of shape (5, 3) filled with random values from a standard normal distribution
2. Create a tensor of shape (4, 4) filled with values from 0 to 15 (use `arange` and `reshape`)
3. Create a tensor of shape (10,) with evenly spaced values from 0 to 1
4. Create a (3, 3) identity matrix (hint: `torch.eye`)

Starter code:

```python
import torch

# Your code here
x1 = ...
x2 = ...
x3 = ...
x4 = ...

```

```
print(x1.shape, x2.shape, x3.shape, x4.shape)
```

1.2: Shape Manipulation - ★★

Goal: Master reshaping and dimension manipulation.
Given `x = torch.arange(24)`:

1. Reshape x to (2, 3, 4)
2. Transpose the last two dimensions to get (2, 4, 3)
3. Flatten the result to 1D
4. Create a view with an extra dimension at position 0 to get (1, 24)

Hint: Use `reshape`, `transpose`, `flatten`, `unsqueeze`.

1.3: Broadcasting Mastery - ★★★

Goal: Understand broadcasting and avoid common mistakes.

1. Create matrix A of shape (5, 1) with values [1, 2, 3, 4, 5]
2. Create vector b of shape (3,) with values [10, 20, 30]
3. Compute C = A + b using broadcasting. What is the shape of C?
4. Compute the outer product of two vectors u (size 4) and v (size 3) without loops (result should be 4×3)
5. Given batch of images (16, 3, 32, 32) and per-channel mean (3,), subtract the mean from each image

Debugging tip: If broadcasting fails, print the shapes! Use `print(A.shape, b.shape)`.

1.4: Advanced Indexing - ★★★

Goal: Learn advanced indexing techniques.
Given `x = torch.randn(100, 5)`:

1. Select all rows where the first column is positive
2. Get the maximum value in each row and its index
3. Create a new tensor with only the even-indexed rows (0, 2, 4, ...)
4. Replace all negative values with 0 (use boolean indexing)

Hint: Use boolean masks: `mask = x[:, 0] > 0`, then `x[mask]`.

1.5: Implementing Distance Matrix - ★★★★

Goal: Use broadcasting for efficient computation.
Given a set of points `X` of shape (N, D) where N=100 points in D=2 dimensions:

1. Compute the pairwise Euclidean distance matrix (N×N) **without loops**
2. The (i, j) entry should be $\|x_i - x_j\|_2$
3. Verify the diagonal is all zeros
4. Find the nearest neighbor for each point (excluding itself)

Hint: Expand dimensions: `X_i = X.unsqueeze(1)` (shape N, 1, D) and `X_j = X.unsqueeze(0)` (shape 1, N, D). Then compute `(X_i - X_j)**2`.

1.6: GPU Operations - ★★★

Goal: Practice moving tensors between devices.

1. Check if CUDA is available on your system
2. Create a large tensor (1000×1000) on CPU
3. Move it to GPU (if available) and time a matrix multiplication
4. Compare the time with the same operation on CPU
5. Handle the case where GPU is not available gracefully

Starter code:

```
import torch
import time

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Your code here
```

2 Autograd: Automatic Differentiation

2.1 Introduction: Why Autograd Matters

Deep learning is fundamentally an optimization problem. We have a model with parameters (weights), a loss function that measures how wrong our predictions are, and we want to adjust the parameters to minimize the loss.

The key insight: we use **gradient descent**. We compute the gradient of the loss with respect to each parameter, then update the parameters in the direction that decreases the loss:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \nabla_\theta \mathcal{L}$$

For a neural network with millions of parameters, computing these gradients by hand is impossible. PyTorch's `autograd` system does this automatically through the chain rule.

2.2 Theory: Computational Graphs and Backpropagation

2.2.1 The Computational Graph

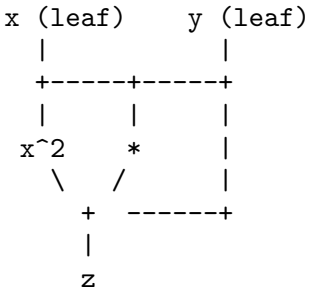
Every operation you perform on a tensor creates a **computational graph**. This graph tracks how each tensor was computed from other tensors.

Example: Simple computation

```
import torch

x = torch.tensor(2.0, requires_grad=True)
y = torch.tensor(3.0, requires_grad=True)
z = x * y + x**2  # z = xy + x^2
```

This creates a graph:



Key Concepts:

- **Leaf tensors:** Input tensors created by the user with `requires_grad=True`
- **Non-leaf tensors:** Result of operations on other tensors
- **grad_fn:** Each non-leaf tensor remembers the operation that created it

2.2.2 Backpropagation: The Chain Rule

When you call `z.backward()`, PyTorch:

1. Traverses the graph backwards from `z`
2. Applies the chain rule at each operation

3. Accumulates gradients in the `.grad` attribute of leaf tensors

For our example:

$$\frac{\partial z}{\partial x} = \frac{\partial}{\partial x}(xy + x^2) = y + 2x$$
$$\frac{\partial z}{\partial y} = \frac{\partial}{\partial y}(xy + x^2) = x$$

With $x = 2, y = 3$: $\frac{\partial z}{\partial x} = 7$, $\frac{\partial z}{\partial y} = 2$

[Why "Automatic"?] You never write the derivative formulas! PyTorch knows the derivative of every operation (add, multiply, sin, exp, matrix multiply, etc.) and chains them automatically. This is the magic of autograd.

2.2.3 Dynamic Computation Graphs

PyTorch uses **dynamic computation graphs** (define-by-run):

- The graph is built *during* the forward pass
- You can use Python control flow (if statements, loops)
- The graph can be different each time you run the code
- After `.backward()`, the graph is *destroyed* (by default)

This is different from TensorFlow 1.x (static graphs) and makes PyTorch more flexible for research.

2.3 Implementation: Using Autograd in Practice

2.3.1 Basic Gradient Computation

```python
import torch

# Create tensors with gradient tracking
x = torch.tensor(2.0, requires_grad=True)
y = torch.tensor(3.0, requires_grad=True)

# Forward pass: compute z
z = x**2 + 2*x*y + y**2

print(z)  # tensor(25., grad_fn=<AddBackward0>)
print(z.grad_fn)  # Shows the operation that created z

# Backward pass: compute gradients
z.backward()

# Gradients are now in .grad
print(x.grad)  # dz/dx = 2x + 2y = 10
print(y.grad)  # dz/dy = 2x + 2y = 10
```

2.3.2 Leaf vs Non-Leaf Tensors

```python
x = torch.tensor(2.0, requires_grad=True)
y = x * 2

print(x.is_leaf)  # True - created by user
print(y.is_leaf)  # False - result of operation

# Only leaf tensors store gradients by default
```

```
z = y.sum()
z.backward()

print(x.grad)  # Available! x is a leaf
print(y.grad)  # None! Non-leaf gradients not retained
```

To retain gradients for non-leaf tensors:

```
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = x * 2
y.retain_grad()  # Tell PyTorch to keep this gradient

z = y.sum()
z.backward()

print(x.grad)  # tensor([2., 2., 2.])
print(y.grad)  # tensor([1., 1., 1.]) - now available!
```

2.3.3 Gradient Accumulation and Zeroing

[Gradients Accumulate!] Calling `.backward()` multiple times **adds** to existing gradients. You must zero them manually between iterations.

```
x = torch.tensor(2.0, requires_grad=True)

# First backward pass
y = x**2
y.backward()
print(x.grad)  # tensor(4.)

# Second backward pass without zeroing
y = x**2
y.backward()
print(x.grad)  # tensor(8.) - DOUBLED!

# Correct approach: zero gradients first
x.grad.zero_()
y = x**2
y.backward()
print(x.grad)  # tensor(4.) - correct again
```

Why does this matter? In training loops, we compute gradients for each batch and update weights. Between batches, we must zero gradients:

```
# Training loop pattern (simplified)
for batch in dataloader:
    # Zero gradients from previous batch
    optimizer.zero_grad()  # Or: model.zero_grad()

    # Forward pass
    output = model(batch)
    loss = criterion(output, target)

    # Backward pass
    loss.backward()

    # Update weights
    optimizer.step()
```

2.3.4 Vector-Jacobian Products

`.backward()` can only be called on scalar outputs (or you must provide a gradient argument):

```
# Scalar output - works
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = x.sum()  # Scalar
y.backward()
print(x.grad)  # Works!

# Vector output - need gradient argument
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = x * 2  # Vector: [2, 4, 6]
# y.backward()  # ERROR! Output is not scalar

# Provide gradient (weights for each output)
gradient = torch.tensor([1.0, 1.0, 1.0])
y.backward(gradient)
print(x.grad)  # tensor([2., 2., 2.])
```

What's happening? When output is a vector $\mathbf{y}$, we compute:

$$\text{x.grad} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}}^T \cdot \text{gradient}$$

This is a **vector-Jacobian product**. For scalar loss, gradient is implicitly 1.

2.3.5 Disabling Gradient Tracking

Sometimes you don't need gradients (inference, debugging, certain computations):

```
x = torch.tensor([1.0, 2.0], requires_grad=True)

# Method 1: torch.no_grad() context
with torch.no_grad():
    y = x * 2
    z = y.sum()
print(z.requires_grad)  # False
# z.backward()  # ERROR - can't backprop

# Method 2: detach() - creates a view without gradients
y = x * 2
y_detached = y.detach()
print(y.requires_grad)          # True
print(y_detached.requires_grad) # False

# Method 3: requires_grad_() - in-place toggle
x = torch.tensor([1.0, 2.0], requires_grad=True)
x.requires_grad_(False)
print(x.requires_grad)  # False
```

[When to Disable Gradients]

- **Inference:** Use `torch.no_grad()` or `torch.inference_mode()` for evaluation
- **Freezing layers:** Set `param.requires_grad = False` for parameters you don't want to train
- **Saving memory:** Gradient tracking uses extra memory
- **Performance:** Operations are faster without gradient tracking

2.3.6 In-Place Operations

[In-Place Operations Break Gradients] In-place operations (those ending with _) can cause errors in backpropagation.

```
# BAD: In-place operation on tensor used in computation
x = torch.tensor([1.0, 2.0], requires_grad=True)
y = x * 2
x.add_(1)  # Modifies x in-place
# y.backward(torch.ones_like(y))  # ERROR! x was modified

# GOOD: Don't modify tensors that are part of the graph
x = torch.tensor([1.0, 2.0], requires_grad=True)
y = x * 2
z = y.sum()
z.backward()  # Works!
```

However, in-place operations are fine for parameters during optimization (outside the graph):

```
# This is OK - happens after .backward()
with torch.no_grad():
    x.add_(-learning_rate * x.grad)
```

2.3.7 Retaining Graphs

By default, the computation graph is freed after `.backward()`:

```
x = torch.tensor(2.0, requires_grad=True)
y = x**2
y.backward()
# y.backward()  # ERROR! Graph was freed

# To keep the graph:
x = torch.tensor(2.0, requires_grad=True)
y = x**2
y.backward(retain_graph=True)
y.backward()  # OK now! But remember to zero gradients if
    needed
```

When is this useful? Multiple backward passes (rare), or when you need to call `.backward()` multiple times in a single forward pass.

2.3.8 Higher-Order Derivatives

PyTorch can compute gradients of gradients:

```
# First derivative
x = torch.tensor(2.0, requires_grad=True)
y = x**3  # y = x^3

dy_dx = torch.autograd.grad(y, x, create_graph=True)[0]
print(dy_dx)  # 3*x^2 = 12

# Second derivative
d2y_dx2 = torch.autograd.grad(dy_dx, x)[0]
print(d2y_dx2)  # 6*x = 12
```

Note: We use `torch.autograd.grad()` instead of `.backward()` here because it gives us the gradient as a tensor (useful for higher-order derivatives).

[torch.autograd.grad vs .backward()]

- `.backward()`: Accumulates gradients in `.grad` attribute. Standard for training.
- `torch.autograd.grad()`: Returns gradients as tensors. Useful for:
  - Higher-order derivatives
  - Multiple gradient computations
  - When you need gradient as a value, not accumulated

2.4 Implementation: Common Patterns and Debugging

2.4.1 Checking Gradients

When implementing custom operations or debugging, verify gradients numerically:

```python
import torch

def numerical_gradient(f, x, eps=1e-5):
    """Compute gradient numerically using finite differences."""
    grad = torch.zeros_like(x)
    for i in range(x.numel()):
        x_plus = x.clone()
        x_plus.flatten()[i] += eps
        x_minus = x.clone()
        x_minus.flatten()[i] -= eps
        grad.flatten()[i] = (f(x_plus) - f(x_minus)) / (2 * eps)
    return grad

# Test it
x = torch.tensor([2.0, 3.0], requires_grad=True)
f = lambda t: (t**2).sum()

# Analytical gradient
y = f(x)
y.backward()
analytical_grad = x.grad

# Numerical gradient
numerical_grad = numerical_gradient(f, x)

print("Analytical:", analytical_grad)  # [4., 6.]
print("Numerical:", numerical_grad)    # [4., 6.] (approximately)
print("Close?", torch.allclose(analytical_grad, numerical_grad))
```

PyTorch also provides `torch.autograd.gradcheck`:

```python
from torch.autograd import gradcheck

x = torch.randn(3, 4, requires_grad=True, dtype=torch.float64)
func = lambda t: (t**2).sum()

# Check if gradients are correct
test = gradcheck(func, x, eps=1e-6)
print(f"Gradient check passed: {test}")
```

2.4.2 Gradient Flow Debugging

When training doesn't work, check if gradients are flowing:

```
import torch.nn as nn

# Simple model
model = nn.Sequential(
    nn.Linear(10, 50),
    nn.ReLU(),
    nn.Linear(50, 1)
)

# Forward pass
x = torch.randn(32, 10)
y = model(x)
loss = y.mean()

# Backward pass
loss.backward()

# Check gradients
for name, param in model.named_parameters():
    if param.grad is not None:
        print(f"{name}: grad mean = {param.grad.abs().mean()
    :.6f}")
    else:
        print(f"{name}: NO GRADIENT!")
```

Common issues:

- **All zeros:** Dead ReLUs, wrong loss function
- **None:** Part of model disconnected from loss
- **Very large:** Exploding gradients (lower learning rate, add gradient clipping)
- **Very small:** Vanishing gradients (different architecture, normalization)

2.4.3 Custom Autograd Functions

For operations PyTorch doesn't support, or to override gradients:

```python
class MySquare(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        """
        Forward pass: compute output and save what's needed for backward.
        ctx is a context object for saving information.
        """
        ctx.save_for_backward(input)
        return input ** 2

    @staticmethod
    def backward(ctx, grad_output):
        """
        Backward pass: compute gradient of loss w.r.t. input.
        grad_output is the gradient of loss w.r.t. output.
        Return gradient w.r.t. each input (input only here).
        """
        input, = ctx.saved_tensors
        grad_input = grad_output * 2 * input
        return grad_input

# Use it
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = MySquare.apply(x)  # Note: .apply() not ()
z = y.sum()
z.backward()

print(x.grad)  # [2., 4., 6.] = 2*x
```

We'll use this for the exercises!

2.5 Exercises

2.1: Basic Gradient Computation - ★★

Goal: Get comfortable with `.backward()` and `.grad`.

1. Create tensors $a = 3$, $b = 4$ with `requires_grad=True`
2. Compute $c = a^2 + b^2$
3. Call `c.backward()` and verify:
   - `a.grad` $= 2a = 6$
   - `b.grad` $= 2b = 8$
4. Now compute $d = 2a + 3b$ and call `d.backward()`
5. What are `a.grad` and `b.grad` now? Why?

Hint: Remember that gradients accumulate!

2.2: Matrix Gradients - ★★

Goal: Understand gradients for matrix operations.

1. Create matrix W of shape (3, 4) with random values, `requires_grad=True`
2. Create input x of shape (4,) with random values (no gradients needed)
3. Compute $y = Wx$ (matrix-vector product)

4. Compute $L = ||y||^2$ (sum of squares)
5. Call `L.backward()` and check the shape of `W.grad`
6. Verify it has the same shape as W

Starter code:

```
import torch
torch.manual_seed(42)

W = torch.randn(3, 4, requires_grad=True)
x = torch.randn(4)

# Your code here
```

2.3: Gradient Accumulation - ★★★

Goal: Master gradient zeroing in training loops.
Implement a mini training loop (without a real model):

1. Create a parameter `w` initialized to 5.0
2. For 10 iterations:
   - Compute loss $= (w - 2)^2$ (we want w to reach 2)
   - Compute gradient
   - Update: $w = w - 0.1 \cdot \text{grad}$
   - **Zero the gradient**
3. Print the final value of `w` (should be close to 2)

Hint: Use `with torch.no_grad():` for the update step.

2.4: Gradient Detaching - ★★★

Goal: Understand when and why to detach gradients.

1. Create $x = [1, 2, 3]$ with gradients
2. Compute $y = x^2$
3. Detach y to create y_{detach}
4. Compute $z_1 = y.sum()$ and $z_2 = y_{\text{detach}}.sum()$
5. Try to call `z1.backward()` - what happens?
6. Try to call `z2.backward()` - what happens?
7. Why would detaching be useful in training?

2.5: Higher-Order Derivatives - ★★★★

Goal: Compute second derivatives (useful for some scientific ML methods).

1. Create $x = 2.0$ with gradients
2. Compute $y = \sin(x)$
3. Use `torch.autograd.grad()` to compute $\frac{dy}{dx}$ with `create_graph=True`
4. Compute the second derivative $\frac{d^2y}{dx^2}$
5. Verify analytically: $y = \sin(x) \Rightarrow y' = \cos(x) \Rightarrow y'' = -\sin(x)$
6. At $x = 2$: $y'' = -\sin(2) \approx -0.909$

Starter code:

```
import torch
```

```python

x = torch.tensor(2.0, requires_grad=True)
y = torch.sin(x)

# First derivative
dy_dx = torch.autograd.grad(y, x, create_graph=True)[0]

# Your code for second derivative
```

2.6: Custom Autograd Function - ★★★★

Goal: Implement a custom operation with manual gradient.
Implement a custom **sigmoid** function:

1. Forward: $\sigma(x) = \frac{1}{1+e^{-x}}$
2. Backward: $\frac{d\sigma}{dx} = \sigma(x)(1 - \sigma(x))$
3. Inherit from `torch.autograd.Function`
4. Implement `forward()` and `backward()` static methods
5. Test it on $x = [0, 1, 2]$
6. Verify gradient using `torch.autograd.gradcheck`

Hint: In `backward()`, you'll need the output of sigmoid. You can either:

- Save the output in `forward` using `ctx.save_for_backward`
- Or recompute it from the saved input

Starter code:

```python
class MySigmoid(torch.autograd.Function):
    @staticmethod
    def forward(ctx, input):
        # Your code here
        pass

    @staticmethod
    def backward(ctx, grad_output):
        # Your code here
        pass

# Test
x = torch.tensor([0.0, 1.0, 2.0], requires_grad=True)
y = MySigmoid.apply(x)
```

3 Building Models with `nn.Module`

3.1 Introduction: Why `nn.Module`?

In Section 2, we manually created tensors with `requires_grad=True` and computed gradients. While this works for simple cases, real neural networks have hundreds of layers and millions of parameters. We need a way to organize this complexity.

`nn.Module` is PyTorch's base class for all neural network components. It provides:

- **Automatic parameter management:** Tracks all learnable parameters
- **GPU/CPU transfer:** Move entire models between devices with one command
- **Training/eval modes:** Control behavior of layers like dropout and batch norm
- **State management:** Save and load model weights
- **Composability:** Build complex models from simple modules

Every neural network layer (`nn.Linear`, `nn.Conv2d`, etc.) and every model you build inherits from `nn.Module`.

3.2 Theory: The Module Hierarchy

A `nn.Module` is a container that can hold:

- **Parameters:** Learnable tensors (weights, biases)
- **Buffers:** Non-learnable tensors (running statistics, constants)
- **Sub-modules:** Other `nn.Module` objects

Example hierarchy:

```
MyModel (nn.Module)
 layer1 (nn.Linear)
    weight (Parameter)
    bias (Parameter)
 layer2 (nn.Linear)
    weight (Parameter)
    bias (Parameter)
 activation (nn.ReLU)
```

When you call `model.parameters()`, PyTorch automatically finds all parameters in the entire tree.

[Key Principle: The Forward Method] Every `nn.Module` must implement a `forward()` method that defines the computation. You never call `forward()` directly—instead, you call the module as a function: `output = model(input)`. This triggers PyTorch's hooks and tracking mechanisms.

3.3 Implementation: Creating Your First Module

3.3.1 A Simple Linear Model

Let's build a simple linear regression model from scratch:

```
1 import torch
2 import torch.nn as nn
3
4 class LinearRegression(nn.Module):
5     def __init__(self, input_dim, output_dim):
6         """
7         Initialize the module.
```

```
        Always call super().__init__() first!
        """
        super().__init__()  # Initialize parent class

        # Define learnable parameters
        self.linear = nn.Linear(input_dim, output_dim)

    def forward(self, x):
        """
        Define the forward pass.
        x: input tensor of shape (batch_size, input_dim)
        returns: output tensor of shape (batch_size,
    output_dim)
        """
        return self.linear(x)

# Create the model
model = LinearRegression(input_dim=10, output_dim=1)

# Use the model
x = torch.randn(32, 10)  # Batch of 32 samples
y = model(x)             # Call the model (invokes forward())
print(y.shape)           # torch.Size([32, 1])
```

[Always Call super().__init__()] Forgetting `super().__init__()` is a common mistake. Without it, PyTorch can't track parameters properly. Always put it as the first line of `__init__()`.

3.3.2 Accessing Parameters

```
model = LinearRegression(10, 1)

# Get all parameters
for name, param in model.named_parameters():
    print(f"{name}: {param.shape}")
# Output:
# linear.weight: torch.Size([1, 10])
# linear.bias: torch.Size([1])

# Count parameters
total_params = sum(p.numel() for p in model.parameters())
print(f"Total parameters: {total_params}")  # 11 (10 weights
    + 1 bias)

# Access specific parameters
print(model.linear.weight)  # The weight matrix
print(model.linear.bias)    # The bias vector
```

3.3.3 Using `nn.Sequential`

For simple feed-forward architectures, `nn.Sequential` is convenient:

```
# Method 1: Pass modules as arguments
model = nn.Sequential(
    nn.Linear(10, 50),
    nn.ReLU(),
    nn.Linear(50, 20),
    nn.ReLU(),
    nn.Linear(20, 1)
)
```

```
9
10 # Method 2: Use OrderedDict for named layers
11 from collections import OrderedDict
12
13 model = nn.Sequential(OrderedDict([
14     ('fc1', nn.Linear(10, 50)),
15     ('relu1', nn.ReLU()),
16     ('fc2', nn.Linear(50, 20)),
17     ('relu2', nn.ReLU()),
18     ('output', nn.Linear(20, 1))
19 ]))
20
21 # Use it
22 x = torch.randn(32, 10)
23 y = model(x)
24
25 # Access layers
26 print(model[0])  # First layer (fc1)
27 print(model.fc1)  # Same, if using OrderedDict
```

When to use Sequential:

- Simple feed-forward architecture
- No branching or skip connections
- No custom logic in forward pass

When to use custom Module:

- Complex architectures (ResNets, U-Nets)
- Multiple inputs/outputs
- Custom forward logic (if statements, loops)
- Need to access intermediate values

3.3.4 Custom Modules with Multiple Layers

```python
class MLP(nn.Module):
    """Multi-Layer Perceptron with configurable hidden layers."""

    def __init__(self, input_dim, hidden_dims, output_dim):
        """
        Args:
            input_dim: Input feature dimension
            hidden_dims: List of hidden layer dimensions
            output_dim: Output dimension
        """
        super().__init__()

        # Build layers dynamically
        layers = []
        prev_dim = input_dim

        for hidden_dim in hidden_dims:
            layers.append(nn.Linear(prev_dim, hidden_dim))
            layers.append(nn.ReLU())
            prev_dim = hidden_dim

        # Output layer (no activation)
        layers.append(nn.Linear(prev_dim, output_dim))

        # Store as Sequential
        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)

# Create a 3-layer network
model = MLP(input_dim=10, hidden_dims=[50, 30], output_dim=1)

# The architecture is: 10 -> 50 -> 30 -> 1
# With ReLU after first two layers
```

3.3.5 Parameters vs Buffers

Parameters: Learnable tensors (updated by optimizer)

Buffers: Non-learnable tensors (not updated by optimizer, but saved with model)

```python
class MyModule(nn.Module):
    def __init__(self):
        super().__init__()

        # Learnable parameter
        self.weight = nn.Parameter(torch.randn(10, 10))

        # Non-learnable buffer (e.g., running statistics)
        self.register_buffer('running_mean', torch.zeros(10))

    def forward(self, x):
        # Use both in computation
        return x @ self.weight + self.running_mean

model = MyModule()

# Parameters (will be updated by optimizer)
```

```
print(len(list(model.parameters())))  # 1 (weight only)

# Buffers (won't be updated by optimizer)
print(len(list(model.buffers())))     # 1 (running_mean)

# Both are saved in state_dict
print(model.state_dict().keys())
# dict_keys(['weight', 'running_mean'])
```

When to use buffers:

- Running statistics in BatchNorm
- Fixed embeddings
- Constants that should be saved with the model
- Anything that needs to move with the model to GPU but shouldn't be trained

3.3.6 Training vs Evaluation Mode

Some layers behave differently during training and evaluation:

```
model = nn.Sequential(
    nn.Linear(10, 50),
    nn.ReLU(),
    nn.Dropout(0.5),  # Randomly drops 50% of values during
    training
    nn.Linear(50, 1)
)

# Training mode (default)
model.train()
x = torch.randn(5, 10)
print(model(x))  # Dropout is active

# Evaluation mode
model.eval()
print(model(x))  # Dropout is disabled

# Check current mode
print(model.training)  # False when in eval mode
```

[Always Set Mode Correctly] Forgetting to call `model.eval()` during inference can lead to incorrect results (dropout still active, batch norm using batch statistics instead of running statistics). Always:

```
model.train()  # Before training
model.eval()   # Before evaluation/inference
```

3.3.7 Moving Models to GPU

```
device = torch.device('cuda' if torch.cuda.is_available()
    else 'cpu')

# Create model
model = MLP(10, [50, 30], 1)

# Move to GPU
model = model.to(device)

# All parameters and buffers are now on GPU
print(next(model.parameters()).device)  # cuda:0

# Input must also be on GPU
x = torch.randn(32, 10, device=device)
y = model(x)  # Works! Both model and input on same device

# Moving back to CPU
model = model.to('cpu')
```

[Device Management Pattern] Common pattern for device-agnostic code:

```
device = torch.device('cuda' if torch.cuda.is_available()
    else 'cpu')
model = model.to(device)

for batch in dataloader:
    x, y = batch
    x, y = x.to(device), y.to(device)
    output = model(x)
    # ... rest of training loop
```

3.3.8 Saving and Loading Models

```
# Save model weights
model = MLP(10, [50, 30], 1)
torch.save(model.state_dict(), 'model_weights.pth')

# Load weights into new model
new_model = MLP(10, [50, 30], 1)  # Must have same
    architecture!
new_model.load_state_dict(torch.load('model_weights.pth'))

# Save entire model (architecture + weights)
torch.save(model, 'entire_model.pth')
loaded_model = torch.load('entire_model.pth')

# For inference, set to eval mode
loaded_model.eval()
```

Best practice: Save `state_dict()` rather than entire model. This is more flexible and portable.

```
# Better: Save with additional info
checkpoint = {
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'epoch': epoch,
    'loss': loss,
}
```

```
torch.save(checkpoint, 'checkpoint.pth')

# Load
checkpoint = torch.load('checkpoint.pth')
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
```

3.3.9 Parameter Initialization

PyTorch layers have default initialization, but you often want custom initialization:

```
import torch.nn.init as init

class InitializedMLP(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)
        self.relu = nn.ReLU()

        # Custom initialization
        self._initialize_weights()

    def _initialize_weights(self):
        # Xavier initialization for fc1
        init.xavier_uniform_(self.fc1.weight)
        init.zeros_(self.fc1.bias)

        # He initialization for fc2 (better for ReLU)
        init.kaiming_normal_(self.fc2.weight, nonlinearity='relu')
        init.zeros_(self.fc2.bias)

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Common initialization strategies:

Method	Use Case	Function
Xavier/Glorot	Sigmoid/Tanh	`init.xavier_uniform_`
He/Kaiming	ReLU	`init.kaiming_normal_`
Zeros	Biases	`init.zeros_`
Constant	Set to specific value	`init.constant_`
Normal	Custom std dev	`init.normal_`
Orthogonal	RNNs	`init.orthogonal_`

Table 2: Common weight initialization methods

[Why Initialization Matters] Poor initialization can cause:

- **Vanishing gradients:** Weights too small, gradients die out
- **Exploding gradients:** Weights too large, gradients explode
- **Slow convergence:** Network takes forever to learn
- **Dead neurons:** ReLUs output zero and never recover

Use Xavier for sigmoid/tanh activations, He for ReLU activations.

3.3.10 Module Hooks for Debugging

Hooks let you inspect intermediate values during forward/backward pass:

```
model = nn.Sequential(
    nn.Linear(10, 50),
    nn.ReLU(),
    nn.Linear(50, 1)
)

# Forward hook: called after forward pass
def forward_hook(module, input, output):
    print(f"Forward: {output.shape}")

# Register hook on second layer
handle = model[2].register_forward_hook(forward_hook)

# Forward pass
x = torch.randn(5, 10)
y = model(x)  # Prints: Forward: torch.Size([5, 1])

# Remove hook when done
handle.remove()

# Backward hook: called during backward pass
def backward_hook(module, grad_input, grad_output):
    print(f"Grad output shape: {grad_output[0].shape}")

handle = model[2].register_backward_hook(backward_hook)
y = model(x)
y.sum().backward()  # Prints grad shape

handle.remove()
```

Use cases for hooks:

- Debugging shape mismatches
- Visualizing activations
- Gradient flow analysis
- Feature extraction from intermediate layers

3.4 Implementation: Practical Patterns

3.4.1 Freezing Layers

Sometimes you want to freeze part of the model (transfer learning):

```
# Freeze all parameters
for param in model.parameters():
    param.requires_grad = False

# Freeze specific layer
for param in model.layer1.parameters():
    param.requires_grad = False

# Only train the last layer
model = nn.Sequential(
    nn.Linear(10, 50),
    nn.ReLU(),
    nn.Linear(50, 1)
)
```

```

# Freeze first two layers
for param in model[:2].parameters():
    param.requires_grad = False

# Now only model[2].parameters() will be updated
optimizer = torch.optim.Adam(
    filter(lambda p: p.requires_grad, model.parameters()),
    lr=0.001
)
```

3.4.2 Multiple Inputs/Outputs

```
class MultiInputModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.branch1 = nn.Linear(10, 50)
        self.branch2 = nn.Linear(5, 50)
        self.combine = nn.Linear(100, 1)

    def forward(self, x1, x2):
        """
        x1: shape (batch, 10)
        x2: shape (batch, 5)
        """
        out1 = self.branch1(x1)
        out2 = self.branch2(x2)
        combined = torch.cat([out1, out2], dim=1)  # (batch, 100)
        return self.combine(combined)

model = MultiInputModel()
x1 = torch.randn(32, 10)
x2 = torch.randn(32, 5)
y = model(x1, x2)
```

3.4.3 Custom Layer with Parameters

```
class ScaledLinear(nn.Module):
    """Linear layer with learnable scaling factor."""

    def __init__(self, in_features, out_features):
        super().__init__()
        self.linear = nn.Linear(in_features, out_features)
        # Custom learnable parameter
        self.scale = nn.Parameter(torch.ones(1))

    def forward(self, x):
        return self.scale * self.linear(x)

layer = ScaledLinear(10, 5)

# Parameters include linear weights/bias AND scale
for name, param in layer.named_parameters():
    print(f"{name}: {param.shape}")
# linear.weight: torch.Size([5, 10])
# linear.bias: torch.Size([5])
# scale: torch.Size([1])
```

3.5 Exercises

3.1: Simple Module - ★★

Goal: Create your first custom module.
Implement a `TwoLayerNet` module:

1. Two linear layers: input_dim → hidden_dim → output_dim
2. ReLU activation between them
3. No activation after the output layer
4. Test with input_dim=20, hidden_dim=50, output_dim=10
5. Verify the output shape for a batch of 16 samples

Starter code:

```python
import torch
import torch.nn as nn

class TwoLayerNet(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim
    ):
        super().__init__()
        # Your code here

    def forward(self, x):
        # Your code here
        pass

# Test
model = TwoLayerNet(20, 50, 10)
x = torch.randn(16, 20)
y = model(x)
print(y.shape)  # Should be torch.Size([16, 10])
```

3.2: Parameter Counting - ★★

Goal: Understand model size.

1. Create a model: 100 → 500 → 200 → 50 → 10 (all linear, ReLU between)
2. Count total parameters programmatically
3. Calculate manually and verify:
   - Layer 1: $100 \times 500 + 500 = 50{,}500$
   - Layer 2: $500 \times 200 + 200 = 100{,}200$
   - ... (continue)
4. Print the number of parameters in each layer

Hint: Use `param.numel()` to count elements in each parameter.

3.3: Sequential vs Custom - ★★★

Goal: Compare two approaches.
Build the same network two ways:

1. Using `nn.Sequential`
2. Using a custom `nn.Module` with explicit layers
3. Both should have: 10 → 50 → 30 → 1 with ReLU activations
4. Load the same random seed before creating each

5. Verify they produce identical outputs for the same input

Challenge: Can you implement a method to copy weights from one to the other?

3.4: Custom Initialization - ★★★

Goal: Implement custom weight initialization.

1. Create a 3-layer MLP
2. Initialize all weights with He initialization
3. Initialize all biases to zero
4. Create another identical model with default initialization
5. Train both on a simple task (e.g., fit $y = x^2$ for x in [-1, 1])
6. Compare convergence speed (plot losses)

Hint: Use `nn.init.kaiming_normal_()` for He initialization.

3.5: Save and Load - ★★★

Goal: Practice model persistence.

1. Create a model and train it for a few steps (any simple task)
2. Save the state dict
3. Create a new model instance
4. Load the weights
5. Verify both models produce identical outputs
6. Try saving/loading with `torch.save(model, ...)` and compare

3.6: Multi-Input Network - ★★★★

Goal: Build a more complex architecture.

Implement a model that takes two inputs and combines them:

1. Input 1: shape (batch, 20)
2. Input 2: shape (batch, 10)
3. Process each through separate 2-layer networks
4. Concatenate the results
5. Pass through a final layer to get output shape (batch, 1)
6. Test with random inputs

Extra challenge: Add skip connections (concatenate the original inputs with processed features before the final layer).

3.7: Frozen Layers - ★★★★

Goal: Practice transfer learning patterns.

1. Create a 4-layer network
2. "Pretrain" it on a simple task
3. Freeze the first two layers (`requires_grad=False`)
4. "Fine-tune" on a different task (only last two layers train)
5. Verify the first two layers' weights don't change
6. Compare with training all layers from scratch

Task idea: First task: classify points in quadrants, second task: classify by

distance from origin.

4 The Training Loop & Optimizers

4.1 Introduction: The Training Process

We now have all the pieces:

- **Tensors:** For data representation
- **Autograd:** For computing gradients
- **Modules:** For building models

Now we need to **train** the model—adjust its parameters to minimize a loss function. This involves:

1. Forward pass: Compute predictions
2. Loss computation: Measure how wrong we are
3. Backward pass: Compute gradients
4. Parameter update: Move in the direction that reduces loss
5. Repeat!

4.2 Theory: Gradient Descent and Optimization

4.2.1 The Optimization Problem

Training is an optimization problem. Given:

- Model f_θ with parameters θ
- Dataset $\{(x_i, y_i)\}_{i=1}^N$
- Loss function $\mathcal{L}$

We want to find:

$$\theta^* = \arg\min_\theta \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f_\theta(x_i), y_i)$$

4.2.2 Gradient Descent

The basic update rule:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}$$

where η is the **learning rate**.

Variants:

- **Batch Gradient Descent:** Use entire dataset (slow, memory intensive)
- **Stochastic Gradient Descent (SGD):** Use one sample (fast, noisy)
- **Mini-Batch SGD:** Use a batch of samples (sweet spot)

In practice, we always use mini-batch SGD (typically 32-256 samples per batch).

4.2.3 Learning Rate

The learning rate η is the most important hyperparameter:

- **Too large:** Training diverges (loss increases)
- **Too small:** Training is very slow
- **Just right:** Steady convergence

Typical ranges: 10^{-4} to 10^{-2} for Adam, 10^{-3} to 10^{-1} for SGD.

[Momentum and Adaptive Learning Rates] Modern optimizers go beyond basic gradient descent:

- **Momentum:** Accumulate gradients over time (helps escape local minima)
- **Adaptive rates:** Different learning rate for each parameter (Adam, RMSprop)

These techniques make training more stable and faster.

4.3 Implementation: The Standard Training Loop

4.3.1 Complete Training Example

Here's the canonical PyTorch training loop:

```
import torch
import torch.nn as nn
import torch.optim as optim

# 1. CREATE MODEL
model = nn.Sequential(
    nn.Linear(10, 50),
    nn.ReLU(),
    nn.Linear(50, 1)
)

# 2. DEFINE LOSS AND OPTIMIZER
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# 3. CREATE DUMMY DATA (replace with real DataLoader)
X_train = torch.randn(1000, 10)
y_train = torch.randn(1000, 1)

# 4. TRAINING LOOP
num_epochs = 100
batch_size = 32

for epoch in range(num_epochs):
    # Shuffle data (simplified; use DataLoader in practice)
    indices = torch.randperm(len(X_train))

    epoch_loss = 0.0
    for i in range(0, len(X_train), batch_size):
        # Get batch
        batch_indices = indices[i:i+batch_size]
        X_batch = X_train[batch_indices]
        y_batch = y_train[batch_indices]

        # FORWARD PASS
        predictions = model(X_batch)
        loss = criterion(predictions, y_batch)

        # BACKWARD PASS
        optimizer.zero_grad()  # Clear old gradients
        loss.backward()        # Compute new gradients

        # UPDATE PARAMETERS
        optimizer.step()

        epoch_loss += loss.item()
```

```
47
48      # Print progress
49      if (epoch + 1) % 10 == 0:
50          avg_loss = epoch_loss / (len(X_train) / batch_size)
51          print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {
    avg_loss:.4f}')
```

[The Five Steps of Training] Every training loop has these steps:

1. `optimizer.zero_grad()`: Clear old gradients
2. `output = model(input)`: Forward pass
3. `loss = criterion(output, target)`: Compute loss
4. `loss.backward()`: Compute gradients
5. `optimizer.step()`: Update parameters

Always in this order!

4.3.2 Loss Functions

Different tasks require different loss functions:

Regression (continuous output):

```
# Mean Squared Error (L2 loss)
criterion = nn.MSELoss()

# Mean Absolute Error (L1 loss, more robust to outliers)
criterion = nn.L1Loss()

# Smooth L1 (Huber loss, combines L1 and L2)
criterion = nn.SmoothL1Loss()
```

Binary Classification (0 or 1):

```
# Binary Cross Entropy (output must be in [0,1], use sigmoid)
criterion = nn.BCELoss()

# BCE with built-in sigmoid (more numerically stable)
criterion = nn.BCEWithLogitsLoss()  # Preferred!

# Example
model = nn.Sequential(nn.Linear(10, 1))  # No sigmoid in
    model
criterion = nn.BCEWithLogitsLoss()
logits = model(x)  # Raw outputs
loss = criterion(logits, targets)  # Applies sigmoid
    internally
```

Multi-Class Classification (one of K classes):

```
# Cross Entropy Loss (combines LogSoftmax + NLLLoss)
criterion = nn.CrossEntropyLoss()

# Example
model = nn.Sequential(nn.Linear(10, 5))  # 5 classes, no
    softmax!
criterion = nn.CrossEntropyLoss()

logits = model(x)  # Shape: (batch, 5), raw scores
targets = torch.tensor([0, 2, 1, 4, 3])  # Class indices

loss = criterion(logits, targets)  # Applies softmax
    internally
```

[Don't Apply Softmax Before CrossEntropyLoss] `nn.CrossEntropyLoss` expects **raw logits**, not probabilities. It applies softmax internally for numerical stability. Applying softmax yourself will give wrong results!

WRONG:

```
output = torch.softmax(model(x), dim=1)
loss = nn.CrossEntropyLoss()(output, targets)  # INCORRECT!
```

CORRECT:

```
output = model(x)  # Raw logits
loss = nn.CrossEntropyLoss()(output, targets)  # Correct!
```

Multi-Label Classification (multiple classes can be active):

```python
# Use BCE with Logits for each class independently
criterion = nn.BCEWithLogitsLoss()

model = nn.Sequential(nn.Linear(10, 5))  # 5 binary outputs
logits = model(x)  # Shape: (batch, 5)
targets = torch.tensor([[1, 0, 1, 0, 0],    # Multiple 1s allowed
                        [0, 1, 1, 1, 0]])  # Shape: (batch, 5)

loss = criterion(logits, targets.float())
```

4.3.3 Optimizers

PyTorch provides many optimizers. Here are the most common:

Stochastic Gradient Descent (SGD):

```python
# Basic SGD
optimizer = optim.SGD(model.parameters(), lr=0.01)

# SGD with momentum (recommended)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

# SGD with momentum and weight decay (L2 regularization)
optimizer = optim.SGD(model.parameters(), lr=0.01,
                      momentum=0.9, weight_decay=1e-4)
```

Adam (Adaptive Moment Estimation):

```python
# Adam: adaptive learning rates per parameter
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Adam with weight decay
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-4)
```

AdamW (Adam with decoupled weight decay):

```python
# AdamW: better weight decay than Adam
optimizer = optim.AdamW(model.parameters(), lr=0.001, weight_decay=0.01)
```

[Which Optimizer to Use?] **For most tasks:** Start with `Adam` or `AdamW` with lr=0.001

Use Adam/AdamW when:

- You want fast initial progress
- You're prototyping
- You have many hyperparameters (adaptive rates help)

Use SGD with momentum when:

- You need the absolute best final performance
- You're training CNNs for vision tasks
- You can afford to tune learning rate carefully

General rule: Adam for fast development, SGD for final tuning.

4.3.4 Learning Rate Schedules

Instead of fixed learning rate, reduce it during training:

```
import torch.optim.lr_scheduler as lr_scheduler

optimizer = optim.Adam(model.parameters(), lr=0.001)

# Step decay: reduce LR every step_size epochs
scheduler = lr_scheduler.StepLR(optimizer, step_size=30,
    gamma=0.1)

# Exponential decay
scheduler = lr_scheduler.ExponentialLR(optimizer, gamma=0.95)

# Reduce on plateau: reduce when loss stops improving
scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, mode='
    min',
                                           factor=0.5,
    patience=10)

# Cosine annealing: smooth decay to zero
scheduler = lr_scheduler.CosineAnnealingLR(optimizer, T_max
    =100)

# Usage in training loop
for epoch in range(num_epochs):
    train_one_epoch()

    # Update learning rate
    scheduler.step()  # For most schedulers

    # For ReduceLROnPlateau, pass validation loss
    # scheduler.step(val_loss)

    # Check current LR
    current_lr = optimizer.param_groups[0]['lr']
    print(f"Current LR: {current_lr}")
```

When to use schedules:

- Long training runs (>100 epochs)
- When loss plateaus
- Fine-tuning pretrained models

When not to use:

- Short training (<50 epochs)
- Already using Adam with good convergence
- Adds complexity without clear benefit

4.3.5 Gradient Clipping

Prevent exploding gradients by limiting their magnitude:

```
# Training loop with gradient clipping
for epoch in range(num_epochs):
    for batch in dataloader:
        optimizer.zero_grad()

        output = model(batch)
```

```
        loss = criterion(output, targets)
        loss.backward()

        # Clip gradients by norm
        torch.nn.utils.clip_grad_norm_(model.parameters(),
    max_norm=1.0)

        optimizer.step()
```

Two methods:

```
# Clip by norm (preferred): scale down if norm > max_norm
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm
    =1.0)

# Clip by value: clamp each gradient to [-clip_value,
    clip_value]
torch.nn.utils.clip_grad_value_(model.parameters(),
    clip_value=0.5)
```

[When to Use Gradient Clipping] **Always use for:**

- RNNs and LSTMs (prone to exploding gradients)
- Transformers
- Any sequence model

Often useful for:

- Deep networks (>50 layers)
- Training instability

Typical values: max_norm=1.0 to 5.0

4.3.6 Validation Loop

Always evaluate on held-out data:

```python
def train_epoch(model, dataloader, criterion, optimizer, device):
    model.train()  # Set to training mode
    total_loss = 0.0

    for batch_idx, (data, target) in enumerate(dataloader):
        data, target = data.to(device), target.to(device)

        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    return total_loss / len(dataloader)

def validate(model, dataloader, criterion, device):
    model.eval()  # Set to evaluation mode
    total_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():  # Disable gradient computation
        for data, target in dataloader:
            data, target = data.to(device), target.to(device)

            output = model(data)
            loss = criterion(output, target)
            total_loss += loss.item()

            # For classification: compute accuracy
            _, predicted = torch.max(output, 1)
            total += target.size(0)
            correct += (predicted == target).sum().item()

    avg_loss = total_loss / len(dataloader)
    accuracy = 100 * correct / total
    return avg_loss, accuracy

# Main training loop
for epoch in range(num_epochs):
    train_loss = train_epoch(model, train_loader, criterion,
                             optimizer, device)
    val_loss, val_acc = validate(model, val_loader, criterion, device)

    print(f'Epoch {epoch+1}: Train Loss={train_loss:.4f}, '
          f'Val Loss={val_loss:.4f}, Val Acc={val_acc:.2f}%')
```

[Don't Forget model.eval() and torch.no_grad()] When validating:

- `model.eval()`: Disables dropout, uses running stats for batch norm
- `torch.no_grad()`: Saves memory and computation

Forgetting these can lead to:

- Incorrect validation metrics
- Out of memory errors
- Slower evaluation

4.3.7 Early Stopping

Stop training when validation loss stops improving:

```python
class EarlyStopping:
    def __init__(self, patience=10, min_delta=0):
        """
        patience: how many epochs to wait after last improvement
        min_delta: minimum change to qualify as improvement
        """
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = None
        self.early_stop = False

    def __call__(self, val_loss):
        if self.best_loss is None:
            self.best_loss = val_loss
        elif val_loss > self.best_loss - self.min_delta:
            self.counter += 1
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_loss = val_loss
            self.counter = 0

# Usage
early_stopping = EarlyStopping(patience=10)

for epoch in range(num_epochs):
    train_loss = train_epoch(...)
    val_loss, val_acc = validate(...)

    early_stopping(val_loss)
    if early_stopping.early_stop:
        print(f"Early stopping at epoch {epoch+1}")
        break
```

4.4 Implementation: Debugging Training

4.4.1 When Training Goes Wrong

Problem: Loss is NaN

```
# Check for NaN
if torch.isnan(loss):
    print("NaN detected!")
    print(f"Input contains NaN: {torch.isnan(data).any()}")
    print(f"Output contains NaN: {torch.isnan(output).any()}")

    # Check gradients
    for name, param in model.named_parameters():
        if param.grad is not None:
            print(f"{name} grad: {param.grad.abs().max()}")
```

Common causes:

- Learning rate too high
- Numerical instability (use `BCEWithLogitsLoss`, not `BCELoss`)
- Division by zero
- Log of zero or negative number

Problem: Loss doesn't decrease

```
# Sanity checks
print("Model output range:", output.min().item(), output.max().item())
print("Target range:", target.min().item(), target.max().item())

# Check if gradients are flowing
for name, param in model.named_parameters():
    if param.grad is not None:
        print(f"{name}: grad mean = {param.grad.abs().mean():.6f}")
```

Common causes:

- Learning rate too low
- Wrong loss function
- Forgot `optimizer.zero_grad()`
- Model architecture issue (dead ReLUs, wrong dimensions)
- Data not normalized

Problem: Loss explodes

- Learning rate too high (reduce by 10x)
- Use gradient clipping
- Check for large batch effects (try smaller batches)

Problem: Overfitting (train loss « val loss)

```
# Add regularization
model = nn.Sequential(
    nn.Linear(10, 50),
    nn.ReLU(),
```

```
    nn.Dropout(0.5),  # Add dropout
    nn.Linear(50, 1)
)

# Add weight decay
optimizer = optim.Adam(model.parameters(), lr=0.001,
    weight_decay=1e-4)

# Use data augmentation (for images)
# Reduce model size
# Get more data
```

Problem: Underfitting (both losses high)

- Increase model capacity (more layers, more neurons)
- Train longer
- Increase learning rate
- Remove regularization
- Check if task is actually learnable

4.4.2 Monitoring Training

```
# Track metrics
train_losses = []
val_losses = []

for epoch in range(num_epochs):
    train_loss = train_epoch(...)
    val_loss, val_acc = validate(...)

    train_losses.append(train_loss)
    val_losses.append(val_loss)

    # Plot periodically
    if epoch % 10 == 0:
        import matplotlib.pyplot as plt
        plt.plot(train_losses, label='Train')
        plt.plot(val_losses, label='Val')
        plt.xlabel('Epoch')
        plt.ylabel('Loss')
        plt.legend()
        plt.show()
```

4.5 Exercises

4.1: Basic Training Loop - ★★

Goal: Implement a complete training loop from scratch.
Task: Fit a neural network to $y = \sin(x)$ for $x \in [0, 2\pi]$

1. Generate 1000 training samples: x uniform in $[0, 2\pi]$, $y = \sin(x)$ + small noise
2. Create a 3-layer MLP: $1 \to 50 \to 50 \to 1$ with ReLU
3. Use MSE loss and Adam optimizer
4. Train for 100 epochs, batch size 32
5. Print loss every 10 epochs
6. Plot predictions vs true function

Starter code:

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

# Generate data
x = torch.linspace(0, 2*np.pi, 1000).reshape(-1, 1)
y = torch.sin(x) + 0.1 * torch.randn_like(x)

# Your code here
```

4.2: Loss Function Comparison - ★★★

Goal: Understand different loss functions.
Train the same model with different losses:

1. MSE Loss (L2)
2. L1 Loss
3. Smooth L1 Loss

Add outliers to the data (10% of points with large noise) and compare:

- Final training loss
- Robustness to outliers
- Convergence speed

Which loss is most robust to outliers?

4.3: Optimizer Comparison - ★★★

Goal: Compare optimizer behavior.
Train the same model with:

1. SGD (lr=0.01)
2. SGD with momentum (lr=0.01, momentum=0.9)
3. Adam (lr=0.001)
4. AdamW (lr=0.001)

Plot all loss curves on the same graph. Observations:

- Which converges fastest initially?
- Which achieves lowest final loss?
- How does momentum help SGD?

4.4: Learning Rate Experiments - ★★★

Goal: Understand learning rate impact.
Train the same model with different learning rates:

- 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1.0

For each:

1. Track loss over epochs
2. Identify: too small, just right, too large
3. Observe divergence (NaN loss) for very large LR

Extra: Implement a learning rate finder (sweep from small to large, stop when loss diverges).

4.5: Overfitting Detection - ★★★★

Goal: Identify and prevent overfitting.

1. Generate a small dataset (100 samples)
2. Split into train (80) and validation (20)
3. Train a large model (can easily overfit)
4. Track both train and validation loss
5. Identify when overfitting starts (losses diverge)
6. Add regularization:
   - Dropout
   - Weight decay
   - Early stopping
7. Compare final validation performance

4.6: Complete Training Pipeline - ★★★★

Goal: Build a production-ready training pipeline.
Implement a complete pipeline with:

1. Data splitting (train/val/test)
2. Training loop with progress bars
3. Validation after each epoch
4. Model checkpointing (save best model)
5. Early stopping
6. Learning rate scheduling
7. Tensorboard logging (optional)
8. Final test set evaluation

Test on a real dataset (e.g., sklearn's make_classification).
Hint: Use `tqdm` for progress bars:

```
from tqdm import tqdm

for epoch in tqdm(range(num_epochs)):
    # training code
```

4.7: Debugging Challenge - ★★★★

Goal: Practice debugging common training issues.
Given buggy training code (deliberately broken):

```
# This code has 5 bugs - find and fix them!
model = nn.Sequential(nn.Linear(10, 1), nn.Softmax())
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=10.0)

for epoch in range(100):
    output = model(X_train)
    loss = criterion(output, y_train)
    loss.backward()
    optimizer.step()
    print(f"Epoch {epoch}, Loss: {loss.item()}")
```

Find and fix:

1. Wrong activation before CrossEntropyLoss
2. Missing `optimizer.zero_grad()`
3. Learning rate too high
4. No train/eval modes
5. No validation set

5 Data Loading & Preprocessing

5.1 Introduction: Why Proper Data Loading Matters

So far, we've used simple tensors for data. In practice, you'll have:

- Large datasets that don't fit in memory
- Complex preprocessing pipelines
- Data on disk (files, databases)
- Need for efficient batching and shuffling
- Different data types (images, time series, point clouds, meshes)

PyTorch provides `Dataset` and `DataLoader` to handle this elegantly:

- **Dataset:** Defines how to access individual samples
- **DataLoader:** Handles batching, shuffling, parallel loading

5.2 Theory: Dataset and DataLoader Architecture

[The Data Pipeline]

```
Raw Data → Dataset → DataLoader → Model
           (access)  (batch, shuffle, parallel)
```

Dataset answers: "How do I get sample i?"

DataLoader answers: "How do I create batches efficiently?"

Dataset Requirements:

A valid dataset must implement:

1. `__len__()`: Return total number of samples
2. `__getitem__(idx)`: Return sample at index `idx`

DataLoader Benefits:

- Automatic batching
- Shuffling for training
- Parallel data loading (multiple workers)
- Automatic memory pinning for GPU transfer
- Handling variable-length sequences

5.3 Implementation: Basic Usage

5.3.1 Using Built-in Datasets

PyTorch provides common datasets in `torchvision.datasets`:

```
import torch
from torch.utils.data import DataLoader
import torchvision
import torchvision.transforms as transforms

# Load MNIST dataset
transform = transforms.ToTensor()  # Convert PIL Image to
    tensor

train_dataset = torchvision.datasets.MNIST(
```

```
    root='./data',
    train=True,
    download=True,
    transform=transform
)

test_dataset = torchvision.datasets.MNIST(
    root='./data',
    train=False,
    download=True,
    transform=transform
)

# Create DataLoaders
train_loader = DataLoader(
    train_dataset,
    batch_size=64,
    shuffle=True,       # Shuffle training data
    num_workers=2,      # Parallel loading
    pin_memory=True     # Faster GPU transfer
)

test_loader = DataLoader(
    test_dataset,
    batch_size=64,
    shuffle=False,      # Don't shuffle test data
    num_workers=2
)

# Iterate through batches
for batch_idx, (images, labels) in enumerate(train_loader):
    # images: (64, 1, 28, 28) - batch of 64 grayscale 28x28 images
    # labels: (64,) - batch of 64 labels
    print(f"Batch {batch_idx}: {images.shape}, {labels.shape}")
    break
```

[DataLoader Parameters] Key parameters:

- **batch_size:** Number of samples per batch (32-256 typically)
- **shuffle:** Randomize order (True for training, False for evaluation)
- **num_workers:** Parallel processes for loading (0=main process, 2-4 for speed)
- **pin_memory:** Faster CPU→GPU transfer (True if using GPU)
- **drop_last:** Drop incomplete last batch (useful for batch norm)

5.3.2 Creating Custom Datasets

For your own data, create a custom `Dataset`:

```
from torch.utils.data import Dataset

class SimpleDataset(Dataset):
    """Example: Dataset from numpy arrays."""

    def __init__(self, X, y):
        """
        Args:
            X: numpy array or tensor of features
            y: numpy array or tensor of labels
        """
        self.X = torch.FloatTensor(X) if not isinstance(X, torch.Tensor) else X
        self.y = torch.FloatTensor(y) if not isinstance(y, torch.Tensor) else y

    def __len__(self):
        """Return total number of samples."""
        return len(self.X)

    def __getitem__(self, idx):
        """Return sample at index idx."""
        return self.X[idx], self.y[idx]

# Usage
import numpy as np

X = np.random.randn(1000, 10)
y = np.random.randn(1000, 1)

dataset = SimpleDataset(X, y)
loader = DataLoader(dataset, batch_size=32, shuffle=True)

for batch_X, batch_y in loader:
    print(batch_X.shape, batch_y.shape)
    break
```

5.3.3 Dataset from Files

More realistic: loading data from disk:

```
import os
import pandas as pd
from PIL import Image

class ImageDataset(Dataset):
    """Load images from a directory with CSV labels."""

    def __init__(self, csv_file, img_dir, transform=None):
        """
        Args:
            csv_file: Path to CSV with columns [filename, label]
            img_dir: Directory with images
            transform: Optional transform to apply
        """
        self.data = pd.read_csv(csv_file)
        self.img_dir = img_dir
```

```
        self.transform = transform

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        # Get image filename and label
        img_name = self.data.iloc[idx, 0]
        label = self.data.iloc[idx, 1]

        # Load image
        img_path = os.path.join(self.img_dir, img_name)
        image = Image.open(img_path).convert('RGB')

        # Apply transforms
        if self.transform:
            image = self.transform(image)

        return image, label

# Usage with transforms
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

dataset = ImageDataset('labels.csv', 'images/', transform=
    transform)
loader = DataLoader(dataset, batch_size=32, shuffle=True)
```

5.3.4 Dataset for Time Series

For scientific computing, time series are common:

```
class TimeSeriesDataset(Dataset):
    """
    Create windows from a long time series.
    Useful for forecasting, where we predict future from past
    .
    """

    def __init__(self, data, window_size, forecast_horizon):
        """
        Args:
            data: 1D array of time series values
            window_size: Length of input sequence
            forecast_horizon: How many steps ahead to predict
        """
        self.data = torch.FloatTensor(data)
        self.window_size = window_size
        self.forecast_horizon = forecast_horizon

    def __len__(self):
        # Number of valid windows
        return len(self.data) - self.window_size - self.
    forecast_horizon + 1

    def __getitem__(self, idx):
        # Input: window_size points starting at idx
```

```
        x = self.data[idx : idx + self.window_size]

        # Target: forecast_horizon points after the input
        y = self.data[idx + self.window_size :
                      idx + self.window_size + self.
    forecast_horizon]

        return x, y

# Example: predict next 10 steps from past 50
data = np.sin(np.linspace(0, 100, 1000))
dataset = TimeSeriesDataset(data, window_size=50,
    forecast_horizon=10)

print(f"Dataset size: {len(dataset)}")
x, y = dataset[0]
print(f"Input shape: {x.shape}, Target shape: {y.shape}")
# Input shape: torch.Size([50]), Target shape: torch.Size
    ([10])

loader = DataLoader(dataset, batch_size=32, shuffle=False)
# Note: Don't shuffle time series if order matters!
```

5.3.5 Dataset for Point Clouds

For 3D data (molecular structures, point clouds):

```python
class PointCloudDataset(Dataset):
    """Dataset for 3D point clouds."""

    def __init__(self, data_dir, num_points=1024):
        """
        Args:
            data_dir: Directory with .npy files, each containing Nx3 points
            num_points: Sample this many points from each cloud
        """
        self.files = [os.path.join(data_dir, f)
                      for f in os.listdir(data_dir) if f.endswith('.npy')]
        self.num_points = num_points

    def __len__(self):
        return len(self.files)

    def __getitem__(self, idx):
        # Load point cloud (N x 3)
        points = np.load(self.files[idx])

        # Sample fixed number of points
        if len(points) > self.num_points:
            # Random sampling
            indices = np.random.choice(len(points), self.num_points,
                                       replace=False)
            points = points[indices]
        elif len(points) < self.num_points:
            # Pad with zeros
            padding = np.zeros((self.num_points - len(points), 3))
            points = np.vstack([points, padding])

        return torch.FloatTensor(points)

# Usage
# dataset = PointCloudDataset('point_clouds/')
# loader = DataLoader(dataset, batch_size=8, shuffle=True)
# for batch in loader:
#     print(batch.shape)  # (8, 1024, 3)
```

5.3.6 Data Transforms

Transforms process data on-the-fly:

```python
from torchvision import transforms

# Common image transforms
transform = transforms.Compose([
    # Resize to fixed size
    transforms.Resize((224, 224)),

    # Data augmentation (for training)
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomRotation(degrees=15),
```

```
    transforms.ColorJitter(brightness=0.2, contrast=0.2),

    # Convert to tensor
    transforms.ToTensor(),  # Converts to [0, 1] and (C, H, W)

    # Normalize (important!)
    transforms.Normalize(mean=[0.485, 0.456, 0.406],  # ImageNet stats
                         std=[0.229, 0.224, 0.225])
])

# For evaluation/test: no data augmentation
test_transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

# Custom transform as a callable
class AddGaussianNoise:
    """Add Gaussian noise to tensor."""
    def __init__(self, mean=0., std=0.1):
        self.mean = mean
        self.std = std

    def __call__(self, tensor):
        return tensor + torch.randn(tensor.size()) * self.std + self.mean

# Use in Compose
transform = transforms.Compose([
    transforms.ToTensor(),
    AddGaussianNoise(std=0.05)
])
```

[When to Apply Transforms] Transforms can be applied in two places:

In Dataset (___getitem___):

- Pro: Each sample transformed on-the-fly
- Pro: Can use data augmentation
- Con: Adds computation during training

Pre-computed and saved:

- Pro: Faster training (no transform overhead)
- Con: More disk space
- Con: Can't use random augmentation

For training with augmentation: use Dataset transforms. For inference: consider pre-computing.

5.3.7 Normalization Strategies

Normalization is crucial for training stability:

```
# Method 1: Standardization (zero mean, unit variance)
def compute_mean_std(dataset):
    """Compute mean and std for normalization."""
    loader = DataLoader(dataset, batch_size=64, shuffle=False)

    mean = 0.0
    std = 0.0
    total_samples = 0

    for data, _ in loader:
        batch_samples = data.size(0)
        data = data.view(batch_samples, data.size(1), -1)
        mean += data.mean(2).sum(0)
        std += data.std(2).sum(0)
        total_samples += batch_samples

    mean /= total_samples
    std /= total_samples

    return mean, std

# Usage
# mean, std = compute_mean_std(train_dataset)
# transform = transforms.Normalize(mean=mean, std=std)

# Method 2: Min-Max scaling to [0, 1] or [-1, 1]
class MinMaxScaler:
    def __init__(self, data):
        self.min = data.min()
        self.max = data.max()

    def transform(self, data):
        return (data - self.min) / (self.max - self.min)

    def inverse_transform(self, data):
        return data * (self.max - self.min) + self.min

# Method 3: Per-feature normalization (for tabular data)
class StandardScaler:
    def __init__(self, data):
        """
        data: (N, D) tensor
        """
        self.mean = data.mean(dim=0)
        self.std = data.std(dim=0)

    def transform(self, data):
        return (data - self.mean) / (self.std + 1e-8)

    def inverse_transform(self, data):
        return data * self.std + self.mean

# Usage
X_train = torch.randn(1000, 10)
scaler = StandardScaler(X_train)
X_train_normalized = scaler.transform(X_train)
X_test_normalized = scaler.transform(X_test)  # Use training
```

```
    statistics!
```

[Always Use Training Statistics] When normalizing:

1. Compute mean/std from **training data only**
2. Apply the same transformation to validation and test data
3. **Never** compute statistics from test data!

Why? Using test statistics is a form of data leakage and gives overly optimistic results.

5.3.8 Handling Variable-Length Sequences

For sequences of different lengths (text, time series with variable duration):

```
from torch.nn.utils.rnn import pad_sequence,
    pack_padded_sequence

def collate_fn(batch):
    """
    Custom collate function for variable-length sequences.

    batch: List of (sequence, label) tuples
    sequences: List of tensors with different lengths
    """
    sequences, labels = zip(*batch)

    # Get lengths
    lengths = torch.LongTensor([len(seq) for seq in sequences
    ])

    # Pad sequences to same length
    sequences_padded = pad_sequence(sequences, batch_first=
    True,
                                    padding_value=0)

    labels = torch.LongTensor(labels)

    return sequences_padded, labels, lengths

# Example dataset with variable lengths
class VariableLengthDataset(Dataset):
    def __init__(self, num_samples=100):
        self.data = []
        for _ in range(num_samples):
            # Random length between 10 and 50
            length = torch.randint(10, 50, (1,)).item()
            seq = torch.randn(length, 5)  # 5 features
            label = torch.randint(0, 2, (1,)).item()
            self.data.append((seq, label))

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return self.data[idx]

# Use with custom collate_fn
dataset = VariableLengthDataset()
loader = DataLoader(dataset, batch_size=8, collate_fn=
    collate_fn)

```

```
for sequences, labels, lengths in loader:
    print(f"Padded sequences: {sequences.shape}")  # (8,
    max_len, 5)
    print(f"Lengths: {lengths}")  # (8,)
    break
```

5.3.9 Train/Validation/Test Split

Proper data splitting:

```
from torch.utils.data import random_split

# Method 1: random_split
dataset = SimpleDataset(X, y)
train_size = int(0.7 * len(dataset))
val_size = int(0.15 * len(dataset))
test_size = len(dataset) - train_size - val_size

train_dataset, val_dataset, test_dataset = random_split(
    dataset, [train_size, val_size, test_size]
)

# Method 2: Using indices (more control)
from torch.utils.data import Subset
import numpy as np

indices = np.arange(len(dataset))
np.random.shuffle(indices)

train_idx = indices[:train_size]
val_idx = indices[train_size:train_size+val_size]
test_idx = indices[train_size+val_size:]

train_dataset = Subset(dataset, train_idx)
val_dataset = Subset(dataset, val_idx)
test_dataset = Subset(dataset, test_idx)

# Method 3: sklearn for stratified split (classification)
from sklearn.model_selection import train_test_split

X = np.random.randn(1000, 10)
y = np.random.randint(0, 3, 1000)  # 3 classes

X_train, X_temp, y_train, y_temp = train_test_split(
    X, y, test_size=0.3, stratify=y, random_state=42
)
X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.5, stratify=y_temp,
    random_state=42
)

train_dataset = SimpleDataset(X_train, y_train)
val_dataset = SimpleDataset(X_val, y_val)
test_dataset = SimpleDataset(X_test, y_test)
```

5.3.10 Efficient Data Loading

For large datasets that don't fit in RAM:

```
class LazyLoadDataset(Dataset):
    """Load data on-demand, don't store in memory."""

    def __init__(self, file_list):
        """
        file_list: List of file paths
        """
        self.file_list = file_list

```

```
    def __len__(self):
        return len(self.file_list)

    def __getitem__(self, idx):
        # Load from disk only when needed
        data = np.load(self.file_list[idx])
        return torch.FloatTensor(data)

# For even larger datasets: memory-mapped arrays
class MemMapDataset(Dataset):
    """Use memory-mapped arrays for huge datasets."""

    def __init__(self, memmap_file, shape, dtype):
        """
        memmap_file: Path to .npy file
        shape: Shape of the data array
        dtype: Data type
        """
        self.data = np.memmap(memmap_file, dtype=dtype,
                              mode='r', shape=shape)

    def __len__(self):
        return self.data.shape[0]

    def __getitem__(self, idx):
        # Read only the needed slice from disk
        return torch.FloatTensor(self.data[idx])
```

[Optimizing DataLoader Performance] If data loading is slow:

1. **Increase num_workers:** 2-4 usually optimal
2. **Use pin_memory=True:** If using GPU
3. **Prefetch:** DataLoader automatically prefetches next batch
4. **Reduce transforms:** Pre-compute expensive transforms
5. **Use SSD:** Faster disk I/O
6. **Check CPU bottleneck:** If GPU is idle, data loading is the issue

Warning: Too many workers can hurt performance (overhead).

5.4 Exercises

5.1: Simple Dataset - ★★

Goal: Create your first custom dataset.

1. Create a synthetic dataset: $y = 2x_1 + 3x_2 + \epsilon$
2. Generate 1000 samples with $x_1, x_2 \sim \mathcal{N}(0, 1)$ and noise $\epsilon \sim \mathcal{N}(0, 0.1)$
3. Implement a custom `Dataset` class
4. Create a `DataLoader` with batch_size=32
5. Iterate through one epoch and print batch shapes

5.2: Train/Val/Test Split - ★★

Goal: Practice proper data splitting.

1. Generate a classification dataset (1000 samples, 3 classes)
2. Split into train (70%), validation (15%), test (15%)
3. Ensure the split is stratified (balanced classes in each split)
4. Create separate DataLoaders for each split
5. Verify class distributions are similar

Hint: Use `sklearn.model_selection.train_test_split` with `stratify`.

5.3: Time Series Dataset - ★★★

Goal: Handle sequential data.

1. Generate a noisy sine wave with 10,000 points
2. Create a dataset that returns windows of size 50 to predict the next 10 points
3. Implement the `TimeSeriesDataset` class
4. Train a simple MLP to predict future values
5. Visualize predictions vs ground truth

Challenge: Try with a real time series (stock prices, weather data).

5.4: Data Normalization - ★★★

Goal: Understand normalization impact.

1. Create a dataset with features on different scales:
   - Feature 1: range [0, 1]
   - Feature 2: range [0, 1000]
   - Feature 3: range [-100, 100]
2. Train a model **without** normalization
3. Train a model **with** standardization
4. Compare:
   - Training speed (epochs to converge)
   - Final loss
   - Gradient magnitudes

5.5: Data Augmentation - ★★★

Goal: Implement custom transforms.

1. Create a simple image dataset (or use MNIST)
2. Implement custom transforms:
   - Add Gaussian noise
   - Random scaling
   - Random shifts
3. Train models with and without augmentation
4. Compare validation performance
5. Visualize augmented samples

5.6: Variable-Length Sequences - ★★★★

Goal: Handle sequences of different lengths.

1. Create a dataset of sequences with random lengths (10-100)
2. Implement a custom `collate_fn` to pad sequences
3. Create a DataLoader using this collate function
4. Train an RNN that handles the padded sequences
5. Use `pack_padded_sequence` for efficiency (we'll learn this properly in the RNN section)

Hint: The collate function should return sequences, lengths, and labels.

5.7: Complete Data Pipeline - ★★★★

Goal: Build a production-ready data pipeline.

Create a complete pipeline with:

1. Custom dataset loading from CSV files
2. Train/val/test split with stratification
3. Different transforms for train (with augmentation) and test
4. Normalization using training statistics
5. Efficient DataLoaders with multiple workers
6. Save and load normalization parameters
7. Handle class imbalance with weighted sampling (optional)

Test on a real dataset (e.g., UCI datasets, Kaggle).

Part II

Deep Learning Architectures

6 Fully Connected Networks (MLPs)

6.1 Introduction: The Foundation of Deep Learning

Multi-Layer Perceptrons (MLPs), also called fully connected networks or feedforward networks, are the simplest and most fundamental deep learning architecture. Every neuron in one layer connects to every neuron in the next layer—hence "fully connected."

Despite their simplicity, MLPs are:

- **Universal approximators:** Can approximate any continuous function
- **Building blocks:** Components of more complex architectures
- **Highly effective:** For tabular data, function approximation, and many scientific computing tasks

When to use MLPs:

- Tabular data (features don't have spatial/temporal structure)
- Function approximation in scientific computing
- As components in larger architectures
- When you have relatively small input dimensions (<1000s of features)

When NOT to use MLPs:

- Images (use CNNs—exploit spatial structure)
- Sequences (use RNNs/Transformers—exploit temporal structure)
- Very high-dimensional inputs (too many parameters)

6.2 Theory: Understanding MLPs

6.2.1 Architecture

An MLP consists of:

- **Input layer:** Receives features $\mathbf{x} \in \mathbb{R}^{d_{in}}$
- **Hidden layers:** Transform representations
- **Output layer:** Produces predictions $\hat{\mathbf{y}} \in \mathbb{R}^{d_{out}}$

Single hidden layer MLP:

$$\mathbf{h} = \sigma(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$

$$\hat{\mathbf{y}} = \mathbf{W}_2\mathbf{h} + \mathbf{b}_2$$

where:

- $\mathbf{W}_1 \in \mathbb{R}^{h \times d_{in}}$: First layer weights
- $\mathbf{b}_1 \in \mathbb{R}^h$: First layer biases
- σ: Activation function (nonlinearity)
- $\mathbf{h} \in \mathbb{R}^h$: Hidden representation
- $\mathbf{W}_2, \mathbf{b}_2$: Second layer parameters

[Universal Approximation Theorem] A single-hidden-layer MLP with enough neurons can approximate any continuous function on a compact domain to arbitrary accuracy.

Key insight: Width matters! But in practice, depth is often more efficient than width.

Intuition: Think of hidden neurons as "basis functions." With enough of them, you can represent complex functions as combinations of simple ones.

6.2.2 Depth vs Width

Wide networks (few layers, many neurons per layer):

- Pro: Easier to optimize (fewer layers = less gradient propagation)
- Pro: Universal approximation holds
- Con: Need exponentially many neurons for complex functions
- Con: Don't learn hierarchical features

Deep networks (many layers, moderate neurons per layer):

- Pro: More parameter efficient (exponentially fewer parameters)
- Pro: Learn hierarchical representations (low-level $\rightarrow$ high-level features)
- Pro: Better generalization (implicit regularization)
- Con: Harder to optimize (vanishing/exploding gradients)
- Con: Requires careful initialization and normalization

Rule of thumb:

- Start with 2-3 hidden layers
- Width: 1-2$\times$ input dimension for first layer, then decrease
- Increase depth if data is complex and you have lots of samples
- Use modern techniques (batch norm, skip connections) for very deep networks

6.2.3 Activation Functions

Activation functions introduce nonlinearity. Without them, multiple layers would collapse to a single linear transformation.

ReLU (Rectified Linear Unit):

$$\text{ReLU}(x) = \max(0, x)$$

- **Pros:** Fast to compute, sparse activations, no vanishing gradient for $x > 0$
- **Cons:** "Dead ReLUs" (neurons that always output 0)
- **Derivative:** $\frac{d}{dx}\text{ReLU}(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$
- **Use when:** Default choice for most tasks, especially deep networks

Leaky ReLU:

$$\text{LeakyReLU}(x) = \max(\alpha x, x) \quad (\text{typically } \alpha = 0.01)$$

- **Pros:** Fixes dead ReLU problem (always has gradient)
- **Cons:** Extra hyperparameter α
- **Use when:** ReLU causes many dead neurons

ELU (Exponential Linear Unit):

$$\text{ELU}(x) = \begin{cases} x & x > 0 \\ \alpha(e^x - 1) & x \leq 0 \end{cases}$$

- **Pros:** Smooth, mean activations closer to zero
- **Cons:** Slower to compute (exponential)
- **Use when:** You want smoother gradients

Tanh (Hyperbolic Tangent):

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- **Pros:** Zero-centered (outputs in $[-1, 1]$), smooth
- **Cons:** Vanishing gradient for large $|x|$
- **Use when:** Need zero-centered activations (RNNs, some scientific applications)

Sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- **Pros:** Outputs in $(0, 1)$ (interpretable as probabilities)
- **Cons:** Strong vanishing gradient, not zero-centered
- **Use when:** **Only in output layer** for binary classification

GELU (Gaussian Error Linear Unit):

$$\text{GELU}(x) = x \cdot \Phi(x) \quad \text{where } \Phi \text{ is CDF of } \mathcal{N}(0, 1)$$

- **Pros:** Smooth, used in state-of-the-art models (BERT, GPT)
- **Cons:** More expensive than ReLU
- **Use when:** Training Transformers or want best performance

Swish (SiLU):

$$\text{Swish}(x) = x \cdot \sigma(x)$$

- **Pros:** Smooth, unbounded above, self-gated
- **Cons:** More expensive than ReLU
- **Use when:** Similar to GELU, modern architecture

[Choosing Activation Functions] **Default choice:** Use **ReLU** for hidden layers unless you have a reason not to.

Use Leaky ReLU or ELU if: You observe many dead neurons (check activation distributions)

Use Tanh if: You need zero-centered activations (RNNs, specific scientific applications)

Use GELU or Swish if: You're training a Transformer or want state-of-the-art performance and can afford the compute

Never use Sigmoid in hidden layers: Strong vanishing gradient makes training very difficult

6.2.4 Why Networks Fail to Train

1. Vanishing Gradients

As gradients backpropagate through many layers, they can become exponentially small.

Cause: Chain rule multiplies many small derivatives (< 1)

Effect: Early layers learn very slowly or not at all

Solutions:

- Use ReLU instead of sigmoid/tanh (ReLU has gradient 1 for $x > 0$)
- Proper initialization (Xavier, He)
- Batch normalization
- Skip connections (ResNets—we'll cover this later)

2. Exploding Gradients

Gradients become exponentially large, causing NaN or very large parameter updates.

Cause: Chain rule multiplies many large derivatives (> 1)

Effect: Training diverges, loss becomes NaN

Solutions:

- Gradient clipping
- Proper initialization
- Lower learning rate
- Batch normalization

3. Dead ReLUs

ReLU neurons that always output 0 (because input is always negative).

Cause: Large negative bias or poor initialization

Effect: Neurons become permanently inactive, reducing model capacity

Detection:

```
# Check percentage of dead neurons
activations = model(x)
dead_percentage = (activations == 0).float().mean()
print(f"Dead neurons: {dead_percentage*100:.2f}%")
```

Solutions:

- Use Leaky ReLU or ELU
- Proper initialization (He initialization)
- Lower learning rate
- Batch normalization

4. Poor Initialization

Starting with inappropriate parameter values.

Effects:

- Too small: Vanishing activations/gradients
- Too large: Exploding activations/gradients

- All same: All neurons learn same features (symmetry problem)

Solution: Use proper initialization schemes (next section).

6.3 Implementation: Building MLPs in PyTorch

6.3.1 Simple MLP with Sequential

```python
import torch
import torch.nn as nn

# 3-layer MLP: input_dim -> 128 -> 64 -> output_dim
model = nn.Sequential(
    nn.Linear(10, 128),
    nn.ReLU(),
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Linear(64, 1)
)

# Test
x = torch.randn(32, 10)  # Batch of 32 samples
y = model(x)
print(y.shape)  # torch.Size([32, 1])
```

6.3.2 Custom MLP Module

```python
class MLP(nn.Module):
    """Flexible MLP with configurable architecture."""

    def __init__(self, input_dim, hidden_dims, output_dim,
                 activation='relu', dropout=0.0):
        """
        Args:
            input_dim: Input feature dimension
            hidden_dims: List of hidden layer dimensions
            output_dim: Output dimension
            activation: Activation function ('relu', 'tanh', 'gelu', etc.)
            dropout: Dropout probability (0 = no dropout)
        """
        super().__init__()

        # Build layers
        layers = []
        prev_dim = input_dim

        for hidden_dim in hidden_dims:
            layers.append(nn.Linear(prev_dim, hidden_dim))

            # Activation
            if activation == 'relu':
                layers.append(nn.ReLU())
            elif activation == 'tanh':
                layers.append(nn.Tanh())
            elif activation == 'gelu':
                layers.append(nn.GELU())
            elif activation == 'leaky_relu':
                layers.append(nn.LeakyReLU(0.01))

            # Dropout (if specified)
```

```
            if dropout > 0:
                layers.append(nn.Dropout(dropout))

            prev_dim = hidden_dim

        # Output layer (no activation)
        layers.append(nn.Linear(prev_dim, output_dim))

        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)

# Usage
model = MLP(input_dim=10, hidden_dims=[128, 64, 32],
    output_dim=1,
            activation='relu', dropout=0.2)

print(model)
"""
MLP(
  (network): Sequential(
    (0): Linear(in_features=10, out_features=128, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.2, inplace=False)
    (3): Linear(in_features=128, out_features=64, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.2, inplace=False)
    ...
  )
)
"""
```

6.3.3 Weight Initialization

Why initialization matters:

```python
# Bad initialization (too small)
model = nn.Linear(100, 100)
with torch.no_grad():
    model.weight.fill_(0.01)
    model.bias.fill_(0)

x = torch.randn(1, 100)
for i in range(10):
    x = torch.relu(model(x))
    print(f"Layer {i}: mean={x.mean():.4f}, std={x.std():.4f}")
# Output: Values shrink to near zero (vanishing)

# Bad initialization (too large)
model = nn.Linear(100, 100)
with torch.no_grad():
    model.weight.fill_(1.0)
    model.bias.fill_(0)

x = torch.randn(1, 100)
for i in range(10):
    x = torch.relu(model(x))
    print(f"Layer {i}: mean={x.mean():.4f}, std={x.std():.4f}")
# Output: Values explode (exploding)
```

Xavier/Glorot Initialization (for Tanh/Sigmoid):

Maintains variance across layers for linear activations.

$$W \sim \mathcal{U}\left(-\sqrt{\frac{6}{n_{in}+n_{out}}}, \sqrt{\frac{6}{n_{in}+n_{out}}}\right)$$

```python
import torch.nn.init as init

# Xavier uniform
init.xavier_uniform_(model.weight)

# Xavier normal
init.xavier_normal_(model.weight)
```

He/Kaiming Initialization (for ReLU):

Accounts for ReLU killing half the neurons (setting them to 0).

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{in}}}\right)$$

```python
# He/Kaiming normal (preferred for ReLU)
init.kaiming_normal_(model.weight, mode='fan_in',
    nonlinearity='relu')

# He/Kaiming uniform
init.kaiming_uniform_(model.weight, nonlinearity='relu')
```

Applying initialization to your model:

```
def init_weights(m):
    """Initialize weights for Linear layers."""
    if isinstance(m, nn.Linear):
        # He initialization for weights
        init.kaiming_normal_(m.weight, mode='fan_in',
    nonlinearity='relu')
        # Zero initialization for biases
        if m.bias is not None:
            init.constant_(m.bias, 0)

# Apply to model
model = MLP(10, [128, 64], 1)
model.apply(init_weights)

# Verify
for name, param in model.named_parameters():
    if 'weight' in name:
        print(f"{name}: mean={param.mean():.4f}, std={param.
    std():.4f}")
```

[Initialization Guidelines] **For ReLU networks:** Use He/Kaiming initialization

```
init.kaiming_normal_(weight, nonlinearity='relu')
```

For Tanh/Sigmoid networks: Use Xavier/Glorot initialization

```
init.xavier_normal_(weight)
```

For biases: Initialize to zero (or small constant)

```
init.constant_(bias, 0)
```

PyTorch defaults: Most layers use reasonable defaults (Kaiming uniform for Linear). Explicit initialization is often unnecessary but can help for deep networks.

6.3.4 Batch Normalization

Normalize activations within each mini-batch to stabilize training.

How it works:

$$\hat{x} = \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$y = \gamma \hat{x} + \beta$$

where μ_B, σ_B are batch mean/std, and γ, β are learnable parameters.

```
class MLPWithBatchNorm(nn.Module):
    def __init__(self, input_dim, hidden_dims, output_dim):
        super().__init__()

        layers = []
        prev_dim = input_dim

        for hidden_dim in hidden_dims:
            layers.append(nn.Linear(prev_dim, hidden_dim))
            layers.append(nn.BatchNorm1d(hidden_dim))  # Add
    BatchNorm
            layers.append(nn.ReLU())
            prev_dim = hidden_dim

        layers.append(nn.Linear(prev_dim, output_dim))
        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)

# Usage
model = MLPWithBatchNorm(10, [128, 64], 1)
```

Where to place BatchNorm?

Two conventions:

1. Linear → BatchNorm → Activation (more common)
2. Linear → Activation → BatchNorm

Both work; the first is more standard.

Training vs Eval mode:

```
model.train()  # Use batch statistics
output = model(x)

model.eval()   # Use running statistics
with torch.no_grad():
    output = model(x)
```

[BatchNorm Pitfalls]

1. **Small batches:** BatchNorm works poorly with batch_size < 8. Use LayerNorm instead.
2. **Forgetting eval():** Model behaves differently in train/eval mode
3. **Before or after activation:** Be consistent in your architecture

6.3.5 Layer Normalization

Normalize across features instead of batch:

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

where μ, σ are computed per sample across features.

```
class MLPWithLayerNorm(nn.Module):
    def __init__(self, input_dim, hidden_dims, output_dim):
        super().__init__()

        layers = []
        prev_dim = input_dim

        for hidden_dim in hidden_dims:
            layers.append(nn.Linear(prev_dim, hidden_dim))
            layers.append(nn.LayerNorm(hidden_dim))  # LayerNorm
            layers.append(nn.ReLU())
            prev_dim = hidden_dim

        layers.append(nn.Linear(prev_dim, output_dim))
        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)
```

BatchNorm vs LayerNorm:

Aspect	BatchNorm	LayerNorm
Normalizes over	Batch dimension	Feature dimension
Requires batches	Yes (issues with batch=1)	No
Train/eval modes	Different	Same
Use in	CNNs, MLPs	Transformers, RNNs
Typical batch size	≥ 8	Any (even 1)

Table 3: BatchNorm vs LayerNorm comparison

6.3.6 Dropout

Randomly zero out neurons during training to prevent overfitting:

```
class MLPWithDropout(nn.Module):
    def __init__(self, input_dim, hidden_dims, output_dim, dropout=0.5):
        super().__init__()

        layers = []
        prev_dim = input_dim

        for hidden_dim in hidden_dims:
            layers.append(nn.Linear(prev_dim, hidden_dim))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(dropout))  # Dropout after activation
            prev_dim = hidden_dim

        layers.append(nn.Linear(prev_dim, output_dim))
        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)

# Usage
model = MLPWithDropout(10, [128, 64], 1, dropout=0.5)

# Training: dropout active
model.train()
output = model(x)  # Some neurons randomly zeroed

# Eval: dropout disabled
model.eval()
output = model(x)  # All neurons active, outputs scaled
```

Dropout probabilities:

- Typical values: 0.2 - 0.5
- Higher dropout = more regularization (but too high hurts performance)
- Often use different dropout rates per layer (higher in later layers)

Where to place dropout:

- After activation functions
- **Not** before the output layer (usually)
- Can place after input layer (input dropout, typically lower rate like 0.1)

6.3.7 Complete MLP with All Techniques

```
class ModernMLP(nn.Module):
    """MLP with all modern techniques."""

    def __init__(self, input_dim, hidden_dims, output_dim,
                 activation='relu', dropout=0.0, use_batch_norm=True):
        super().__init__()

        self.input_bn = nn.BatchNorm1d(input_dim) if use_batch_norm else None
```

```
9
10          layers = []
11          prev_dim = input_dim
12
13          for i, hidden_dim in enumerate(hidden_dims):
14              # Linear layer
15              layers.append(nn.Linear(prev_dim, hidden_dim))
16
17              # Batch normalization
18              if use_batch_norm:
19                  layers.append(nn.BatchNorm1d(hidden_dim))
20
21              # Activation
22              if activation == 'relu':
23                  layers.append(nn.ReLU())
24              elif activation == 'gelu':
25                  layers.append(nn.GELU())
26              elif activation == 'leaky_relu':
27                  layers.append(nn.LeakyReLU(0.01))
28
29              # Dropout (if specified)
30              if dropout > 0:
31                  layers.append(nn.Dropout(dropout))
32
33              prev_dim = hidden_dim
34
35          # Output layer
36          layers.append(nn.Linear(prev_dim, output_dim))
37
38          self.network = nn.Sequential(*layers)
39
40          # Initialize weights
41          self.apply(self._init_weights)
42
43      def _init_weights(self, m):
44          if isinstance(m, nn.Linear):
45              init.kaiming_normal_(m.weight, nonlinearity='relu
    ')
46              if m.bias is not None:
47                  init.constant_(m.bias, 0)
48
49      def forward(self, x):
50          # Optional input batch norm
51          if self.input_bn is not None:
52              x = self.input_bn(x)
53          return self.network(x)
54
55 # Usage: best practices
56 model = ModernMLP(
57      input_dim=10,
58      hidden_dims=[256, 128, 64],
59      output_dim=1,
60      activation='relu',
61      dropout=0.3,
62      use_batch_norm=True
63 )
```

6.4 Implementation: Debugging MLPs

6.4.1 Checking for Dead ReLUs

```python
def check_dead_neurons(model, dataloader, device):
    """Check percentage of dead ReLU neurons."""
    model.eval()

    # Hooks to capture activations
    activations = {}

    def hook_fn(name):
        def hook(module, input, output):
            if name not in activations:
                activations[name] = []
            activations[name].append((output == 0).float())
        return hook

    # Register hooks on ReLU layers
    hooks = []
    for name, module in model.named_modules():
        if isinstance(module, nn.ReLU):
            hooks.append(module.register_forward_hook(hook_fn(name)))

    # Forward pass on some data
    with torch.no_grad():
        for data, _ in dataloader:
            data = data.to(device)
            model(data)
            break  # Just one batch

    # Remove hooks
    for hook in hooks:
        hook.remove()

    # Compute dead neuron percentage
    for name, acts in activations.items():
        dead_pct = torch.cat(acts).mean().item() * 100
        print(f"{name}: {dead_pct:.2f}% neurons always zero")

# Usage
# check_dead_neurons(model, train_loader, device)
```

6.4.2 Visualizing Activation Distributions

```python
import matplotlib.pyplot as plt

def plot_activation_distributions(model, x):
    """Plot distribution of activations in each layer."""
    model.eval()

    activations = []

    def hook_fn(module, input, output):
        activations.append(output.detach().cpu().flatten())

    # Register hooks
    hooks = []
    for name, module in model.named_modules():
        if isinstance(module, (nn.Linear, nn.ReLU)):
```

```
            hooks.append(module.register_forward_hook(hook_fn))

    # Forward pass
    with torch.no_grad():
        model(x)

    # Remove hooks
    for hook in hooks:
        hook.remove()

    # Plot
    fig, axes = plt.subplots(len(activations), 1,
                             figsize=(10, 3*len(activations)))
    for i, act in enumerate(activations):
        axes[i].hist(act.numpy(), bins=50)
        axes[i].set_title(f'Layer {i} activations')
        axes[i].set_xlabel('Value')
        axes[i].set_ylabel('Count')
    plt.tight_layout()
    plt.show()

# Usage
# x = torch.randn(100, 10)
# plot_activation_distributions(model, x)
```

6.4.3 Gradient Flow Visualization

```
def plot_gradient_flow(model):
    """
    Plot gradient flow through network.
    Call after loss.backward() but before optimizer.step()
    """
    ave_grads = []
    max_grads = []
    layers = []

    for name, param in model.named_parameters():
        if param.requires_grad and param.grad is not None:
            layers.append(name)
            ave_grads.append(param.grad.abs().mean().item())
            max_grads.append(param.grad.abs().max().item())

    plt.figure(figsize=(12, 6))
    plt.bar(range(len(ave_grads)), ave_grads, alpha=0.5, label='mean')
    plt.bar(range(len(max_grads)), max_grads, alpha=0.5, label='max')
    plt.hlines(0, 0, len(ave_grads), linewidth=2, color='k')
    plt.xticks(range(len(layers)), layers, rotation='vertical')
    plt.xlim(-1, len(ave_grads))
    plt.xlabel('Layers')
    plt.ylabel('Gradient magnitude')
    plt.legend()
    plt.title('Gradient Flow')
    plt.grid(True)
    plt.tight_layout()
    plt.show()

# Usage in training loop
```

```
# output = model(x)
# loss = criterion(output, y)
# loss.backward()
# plot_gradient_flow(model)  # Before optimizer.step()
# optimizer.step()
```

6.5 Exercises

6.1: Building Your First MLP - ★★

Goal: Create and train a basic MLP.

1. Build an MLP: $20 \to 64 \to 32 \to 1$
2. Use ReLU activations
3. Train on synthetic data: $y = \sum_{i=1}^{20} x_i + \epsilon$
4. Use 1000 samples, MSE loss, Adam optimizer
5. Train for 100 epochs
6. Plot training loss curve

Success criterion: Final loss < 0.1

Starter code:

```
import torch
import torch.nn as nn
import torch.optim as optim

# Generate data
X = torch.randn(1000, 20)
y = X.sum(dim=1, keepdim=True) + 0.1 * torch.randn(1000,
    1)

# Your MLP here
model = ...

# Training loop here
```

6.2: Activation Function Comparison - ★★★

Goal: Understand how different activations affect training.

Train identical MLPs (3 layers, 64 hidden units) with different activations:

1. ReLU
2. Leaky ReLU
3. Tanh
4. GELU
5. Sigmoid (observe why this is bad for hidden layers!)

For each:

- Train on the same task (e.g., nonlinear function approximation)
- Track training loss over epochs
- Measure final test performance
- Plot all loss curves on the same graph

Questions to answer:

- Which converges fastest?
- Which achieves best final performance?
- Why does sigmoid perform poorly?

6.3: Initialization Impact - ★★★

Goal: See how initialization affects training.

Create the same MLP with different initializations:

1. All zeros (model won't learn—why?)
2. All ones (all neurons learn the same features)
3. Random normal with $\sigma = 0.01$ (too small)
4. Random normal with $\sigma = 1.0$ (too large)
5. Xavier initialization
6. He initialization (best for ReLU)

For each:

- Train for 50 epochs
- Track loss
- Check for NaN values

Visualize: Plot all loss curves. Which initializations fail? Which work best?

6.4: Depth vs Width Experiment - ★★★

Goal: Compare deep vs wide networks.
Create networks with approximately the same number of parameters:

1. **Shallow & wide:** 2 layers, 256 neurons each
2. **Deep & narrow:** 8 layers, 64 neurons each
3. **Medium:** 4 layers, 128 neurons each

Train all three on a moderately complex task (e.g., 2D function approximation).
Compare:

- Training speed (epochs to converge)
- Final test performance
- Generalization (train vs test loss gap)

Challenge: Add batch normalization to the deep network. Does it help?

6.5: Regularization Techniques - ★★★★

Goal: Prevent overfitting with regularization.

1. Create a small dataset (200 samples)
2. Build a large MLP (can easily overfit): $10 \to 128 \to 128 \to 64 \to 1$
3. Train without regularization (observe overfitting)
4. Add dropout (try 0.1, 0.3, 0.5)
5. Add batch normalization
6. Add weight decay
7. Combine techniques

For each variant:

- Track train and validation loss
- Measure final test performance
- Plot train vs val loss curves

Questions:

- Which technique helps most?
- What's the best combination?
- Can you achieve zero overfitting gap?

6.6: Complex Function Approximation - ★★★★

Goal: Use MLP for scientific computing task.
Approximate a complex 2D function:

$$f(x, y) = \sin(5x)\cos(5y) + 0.5\sin(10x) + 0.3\cos(10y)$$

1. Generate training data: sample (x, y) uniformly in $[-1, 1]^2$
2. Create MLP: 2 inputs → hidden layers → 1 output
3. Train with MSE loss
4. Visualize:
   - True function as heatmap
   - MLP prediction as heatmap
   - Absolute error heatmap
5. Try different architectures (depth, width)
6. Try different activations

Success criterion: Mean absolute error < 0.05 on test set
Starter code for visualization:

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

# Create grid
x = np.linspace(-1, 1, 100)
y = np.linspace(-1, 1, 100)
X, Y = np.meshgrid(x, y)

# True function
Z_true = np.sin(5*X)*np.cos(5*Y) + 0.5*np.sin(10*X) +
    0.3*np.cos(10*Y)

# Model prediction
grid_points = torch.FloatTensor(np.stack([X.ravel(), Y.
    ravel()], axis=1))
with torch.no_grad():
    Z_pred = model(grid_points).numpy().reshape(X.shape)

# Plot
fig, axes = plt.subplots(1, 3, figsize=(15, 4))
axes[0].contourf(X, Y, Z_true, levels=20)
axes[0].set_title('True Function')
axes[1].contourf(X, Y, Z_pred, levels=20)
axes[1].set_title('MLP Prediction')
axes[2].contourf(X, Y, np.abs(Z_true - Z_pred), levels
    =20)
axes[2].set_title('Absolute Error')
plt.show()
```

6.6 Key Takeaways

MLPs are versatile:

- Universal function approximators
- Foundation for understanding all neural networks
- Effective for tabular data and function approximation

Design choices matter:

- **Activation:** ReLU is default, but consider alternatives for specific tasks
- **Depth vs Width:** Deeper networks more parameter-efficient but harder to train
- **Initialization:** He for ReLU, Xavier for Tanh, crucial for deep networks
- **Normalization:** BatchNorm stabilizes training, especially for deep networks
- **Regularization:** Dropout and weight decay prevent overfitting

Common failure modes:

- Dead ReLUs → Use Leaky ReLU or proper initialization
- Vanishing gradients → Use ReLU, batch norm, or skip connections
- Exploding gradients → Use gradient clipping, lower LR, batch norm
- Overfitting → Add dropout, weight decay, or get more data

Best practices for MLPs:

1. Start with 2-3 hidden layers
2. Use ReLU activation
3. Initialize with He initialization
4. Add batch normalization for deep networks
5. Use dropout (0.2-0.5) to prevent overfitting
6. Monitor both train and validation loss
7. Visualize activations and gradients when debugging

7 Convolutional Neural Networks (CNNs)

7.1 Introduction: Why Convolutions?

MLPs work well for tabular data, but they have serious limitations for spatial data like images:

Problems with MLPs for images:

- **Too many parameters:** A 224×224 RGB image has 150,528 inputs. First hidden layer with 1000 neurons = 150M parameters!
- **No spatial structure:** Treats each pixel independently, ignoring that nearby pixels are related
- **Not translation invariant:** A cat in the top-left corner is different from a cat in the center
- **Can't generalize:** Learning a feature at one position doesn't help recognize it elsewhere

Convolutions solve these problems through:

1. **Parameter sharing:** Same kernel applied everywhere (drastically fewer parameters)
2. **Local connectivity:** Each neuron only looks at a small region (receptive field)
3. **Translation invariance:** Features learned at one position work everywhere
4. **Hierarchical learning:** Early layers detect edges, later layers detect complex patterns

When to use CNNs:

- Images (most common use case)
- Time series (1D convolutions)
- Volumetric data (3D medical scans, physical simulations)
- Spatial/temporal data with local correlations

When NOT to use CNNs:

- Tabular data (no spatial structure)
- Very small spatial dimensions (convolutions overkill)
- When you need long-range dependencies only (use attention/Transformers)

7.2 Theory: How Convolutions Work

7.2.1 The Convolution Operation

A 2D convolution slides a **kernel** (or **filter**) over the input, computing dot products:

$$\text{Output}[i,j] = \sum_{m=0}^{k-1}\sum_{n=0}^{k-1} \text{Input}[i+m, j+n] \cdot \text{Kernel}[m,n]$$

Example: 3×3 kernel on 5×5 input

```
Input (5x5):              Kernel (3x3):
1  2  3  4  5             1  0 -1
2  3  4  5  6      *      2  0 -2
3  4  5  6  7             1  0 -1
```

```
4  5  6  7  8
5  6  7  8  9

Output (3x3):
-4  -4  -4
-4  -4  -4
-4  -4  -4
```

This particular kernel is a vertical edge detector (Sobel filter).

[Key Concepts]

- **Kernel/Filter:** Small matrix of learnable weights (e.g., 3×3, 5×5)
- **Stride:** How many pixels to move the kernel each step (stride=1 → move 1 pixel)
- **Padding:** Add zeros around input to control output size
- **Channels:** Input depth (RGB image has 3 channels)
- **Feature maps:** Output of applying convolution (number = number of kernels)

7.2.2 Parameter Sharing

A single 3×3 kernel has only 9 parameters, but is applied to every position in the image.

MLP for 28×28 image:

- Input: 784 pixels
- Hidden layer: 128 neurons
- Parameters: $784 \times 128 = 100{,}352$

CNN for same image:

- Input: 28×28×1
- Conv layer: 32 filters, 3×3 kernel
- Parameters: $3 \times 3 \times 1 \times 32 + 32 = 320$ (including biases)

300× fewer parameters!

7.2.3 Receptive Fields

The **receptive field** of a neuron is the region of the input it "sees."

Example: Two 3×3 conv layers

```
Layer 1: Each neuron sees 3×3 region
Layer 2: Each neuron sees 3×3 of Layer 1
         → Sees 5×5 region of original input!
```

Receptive field grows with depth:

- 1 layer (3×3 kernel): $3 \times 3 = 9$ pixels
- 2 layers: $5 \times 5 = 25$ pixels
- 3 layers: $7 \times 7 = 49$ pixels
- n layers: $(2n + 1) \times (2n + 1)$ pixels (for 3×3 kernels, stride=1)

Formula for receptive field:

$$r_n = r_{n-1} + (k - 1) \cdot \prod_{i=1}^{n-1} s_i$$

where k is kernel size, s_i is stride at layer i.

[Why Receptive Fields Matter] For a network to classify an entire image, the final layer's receptive field must cover the whole image. This determines your network depth:

Small images (32×32): 5-10 layers sufficient

Large images (224×224): 15-20+ layers needed

Use pooling or larger strides to grow receptive field faster.

7.2.4 Output Size Calculation

Formula for output size:

$$\text{Output size} = \left\lfloor \frac{\text{Input size} + 2 \times \text{Padding} - \text{Kernel size}}{\text{Stride}} \right\rfloor + 1$$

Examples:

Input	Kernel	Stride	Padding	Output	Note
28	3	1	0	26	Shrinks by 2
28	3	1	1	28	Same size
28	3	2	0	13	Half size
28	5	1	2	28	Same size
32	3	1	0	30	Shrinks by 2

Table 4: Output size examples

Common padding strategies:

- **Valid (no padding):** Output shrinks
- **Same:** Output size = input size (for stride=1)
- **Full:** Maximum padding (output grows)

7.2.5 Pooling

Pooling **downsamples** the spatial dimensions, making the network:

- More computationally efficient
- More translation invariant
- With larger receptive fields

Max Pooling: Take maximum value in each window

```
Input (4x4):          Max Pool 2x2, stride=2:
1  2  3  4
2  4  6  8      →     4   8
3  6  9  12           12  16
4  8  12 16
```

Average Pooling: Take average value in each window

```
Input (4x4):          Avg Pool 2x2, stride=2:
1  2  3  4
2  4  6  8      →     2.25  5.25
3  6  9  12           5.25  11.25
4  8  12 16
```

When to use each:

- **Max pooling:** Most common. Preserves strong features. Good for classification.
- **Average pooling:** Smoother. Good for final global pooling. Sometimes better for regression.
- **No pooling:** Use strided convolutions instead (more learnable, modern approach)

7.3 Implementation: Building CNNs in PyTorch

7.3.1 Conv2d: The Core Layer

```python
import torch
import torch.nn as nn

# Basic Conv2d
conv = nn.Conv2d(
    in_channels=3,       # Input channels (e.g., RGB = 3)
    out_channels=64,     # Number of filters (output channels)
    kernel_size=3,       # 3x3 kernel
    stride=1,            # Move 1 pixel at a time
    padding=1,           # Pad with 1 pixel of zeros (same
    size)
    bias=True            # Include bias term
)

# Input: (batch_size, channels, height, width)
x = torch.randn(8, 3, 32, 32)  # 8 RGB images, 32x32
output = conv(x)
print(output.shape)  # torch.Size([8, 64, 32, 32])
# 64 feature maps, same spatial size due to padding=1

# Number of parameters
params = conv.weight.numel() + conv.bias.numel()
print(f"Parameters: {params}")  # 3*3*3*64 + 64 = 1792
```

Key parameters explained:

- **in_channels:** Depth of input (3 for RGB, 1 for grayscale)
- **out_channels:** Number of filters = depth of output
- **kernel_size:** Can be int ($3 \rightarrow 3\times3$) or tuple ($(3, 5) \rightarrow 3\times5$)
- **stride:** Step size. stride=2 halves spatial dimensions
- **padding:**
  - int: uniform padding
  - tuple: (pad_height, pad_width)
  - 'same': auto-pad to keep size (stride=1 only)
  - 'valid': no padding
- **dilation:** Spacing between kernel elements (1 = standard, >1 = dilated)
- **groups:** Divide channels into groups (1 = standard, in_channels = depthwise)

7.3.2 Shape Calculation Practice

```python
# Practice calculating output shapes
x = torch.randn(1, 3, 64, 64)  # Single 64x64 RGB image

# Example 1: Standard convolution
conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
out1 = conv1(x)
print(out1.shape)  # [1, 32, 64, 64] - same size

# Example 2: Strided convolution (downsampling)
conv2 = nn.Conv2d(3, 32, kernel_size=3, stride=2, padding=1)
out2 = conv2(x)
print(out2.shape)  # [1, 32, 32, 32] - halved

# Example 3: No padding (shrinks)
```

```
conv3 = nn.Conv2d(3, 32, kernel_size=5, padding=0)
out3 = conv3(x)
print(out3.shape)  # [1, 32, 60, 60] - shrinks by 4

# Example 4: Large kernel
conv4 = nn.Conv2d(3, 32, kernel_size=7, padding=3)
out4 = conv4(x)
print(out4.shape)  # [1, 32, 64, 64] - same size

# Calculate expected output
def calc_output_size(input_size, kernel_size, stride, padding):
    return (input_size + 2*padding - kernel_size) // stride + 1

print(calc_output_size(64, 3, 1, 1))  # 64
print(calc_output_size(64, 3, 2, 1))  # 32
print(calc_output_size(64, 5, 1, 0))  # 60
```

[Common Shape Mistakes] **Mistake 1:** Forgetting to flatten before Linear layer

```
# WRONG
output = conv_layers(x)  # Shape: (batch, channels, H, W)
output = fc(output)      # ERROR! Wrong shape

# CORRECT
output = conv_layers(x)
output = output.view(output.size(0), -1)  # Flatten
output = fc(output)  # OK
```

Mistake 2: Wrong channel order (PyTorch uses NCHW, not NHWC)

```
# PyTorch expects: (batch, channels, height, width)
x = torch.randn(8, 32, 32, 3)  # WRONG order
x = x.permute(0, 3, 1, 2)      # Fix to (8, 3, 32, 32)
```

7.3.3 Building a Simple CNN

```python
class SimpleCNN(nn.Module):
    """Simple CNN for MNIST (28x28 grayscale images)."""

    def __init__(self, num_classes=10):
        super().__init__()

        # Convolutional layers
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        # Output: 32 x 28 x 28

        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        # Output: 64 x 28 x 28

        self.pool = nn.MaxPool2d(2, 2)  # 2x2 pooling
        # After pool: 64 x 14 x 14

        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        # Output: 128 x 14 x 14
        # After pool: 128 x 7 x 7

        # Fully connected layers
        self.fc1 = nn.Linear(128 * 7 * 7, 256)
        self.fc2 = nn.Linear(256, num_classes)

        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        # Conv block 1
        x = self.relu(self.conv1(x))
        x = self.pool(x)

        # Conv block 2
        x = self.relu(self.conv2(x))
        x = self.pool(x)

        # Conv block 3
        x = self.relu(self.conv3(x))

        # Flatten
        x = x.view(x.size(0), -1)  # or x.flatten(1)

        # Fully connected
        x = self.relu(self.fc1(x))
        x = self.dropout(x)
        x = self.fc2(x)

        return x

# Test
model = SimpleCNN()
x = torch.randn(4, 1, 28, 28)  # 4 grayscale 28x28 images
output = model(x)
print(output.shape)  # torch.Size([4, 10])
```

7.3.4 Modern CNN with Batch Normalization

```python
class ModernCNN(nn.Module):
    """CNN with batch normalization and modern practices."""

    def __init__(self, num_classes=10):
        super().__init__()

        self.features = nn.Sequential(
            # Block 1: 1 -> 32
            nn.Conv2d(1, 32, 3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.Conv2d(32, 32, 3, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),  # 28x28 -> 14x14

            # Block 2: 32 -> 64
            nn.Conv2d(32, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(2, 2),  # 14x14 -> 7x7

            # Block 3: 64 -> 128
            nn.Conv2d(64, 128, 3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(128, 128, 3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
        )

        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)),  # Global average pooling
            nn.Flatten(),
            nn.Linear(128, 256),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(256, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x

# Test
model = ModernCNN()
x = torch.randn(4, 1, 28, 28)
output = model(x)
print(output.shape)  # torch.Size([4, 10])
```

[AdaptiveAvgPool2d vs Flatten] **Old approach:** Fixed spatial size before FC layer

```python
x = x.view(batch_size, -1)  # Must know spatial size
fc = nn.Linear(128 * 7 * 7, num_classes)
```

Modern approach: Global pooling adapts to any input size

```
x = nn.AdaptiveAvgPool2d((1, 1))(x)  # Always outputs 1x1
x = x.flatten(1)  # (batch, 128, 1, 1) -> (batch, 128)
fc = nn.Linear(128, num_classes)
```

Benefit: Same network works for different input sizes (e.g., 28×28 or 32×32).

7.3.5 1D Convolutions for Time Series

```python
class TimeSeriesCNN(nn.Module):
    """1D CNN for time series or sequence data."""

    def __init__(self, input_channels=1, num_classes=10):
        super().__init__()

        self.conv1 = nn.Conv1d(
            in_channels=input_channels,
            out_channels=64,
            kernel_size=7,  # Look at 7 time steps
            padding=3
        )

        self.conv2 = nn.Conv1d(64, 128, kernel_size=5, padding=2)
        self.conv3 = nn.Conv1d(128, 256, kernel_size=3, padding=1)

        self.pool = nn.MaxPool1d(2)  # Downsample by 2

        self.global_pool = nn.AdaptiveAvgPool1d(1)
        self.fc = nn.Linear(256, num_classes)

        self.relu = nn.ReLU()

    def forward(self, x):
        # x shape: (batch, channels, sequence_length)

        x = self.relu(self.conv1(x))
        x = self.pool(x)

        x = self.relu(self.conv2(x))
        x = self.pool(x)

        x = self.relu(self.conv3(x))

        # Global pooling
        x = self.global_pool(x)  # (batch, 256, 1)
        x = x.squeeze(-1)        # (batch, 256)

        x = self.fc(x)
        return x

# Test on time series
model = TimeSeriesCNN(input_channels=1)
x = torch.randn(32, 1, 100)  # 32 samples, 1 channel, 100 time steps
output = model(x)
print(output.shape)  # torch.Size([32, 10])
```

When to use 1D convolutions:

- Time series data (sensor readings, stock prices)
- Audio signals (raw waveforms)
- Text (character or word level)
- Any 1D sequence with local correlations

7.3.6 3D Convolutions for Volumetric Data

```python
class VolumetricCNN(nn.Module):
    """3D CNN for volumetric data (medical scans, video, 3D simulations)."""

    def __init__(self, in_channels=1, num_classes=2):
        super().__init__()

        self.conv1 = nn.Conv3d(in_channels, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv3d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv3d(64, 128, kernel_size=3, padding=1)

        self.pool = nn.MaxPool3d(2)  # Downsample all 3 dimensions

        self.global_pool = nn.AdaptiveAvgPool3d((1, 1, 1))
        self.fc = nn.Linear(128, num_classes)

        self.relu = nn.ReLU()

    def forward(self, x):
        # x shape: (batch, channels, depth, height, width)

        x = self.relu(self.conv1(x))
        x = self.pool(x)

        x = self.relu(self.conv2(x))
        x = self.pool(x)

        x = self.relu(self.conv3(x))

        x = self.global_pool(x)
        x = x.view(x.size(0), -1)

        x = self.fc(x)
        return x

# Test on 3D volume
model = VolumetricCNN()
x = torch.randn(4, 1, 32, 32, 32)  # 4 volumes, 32x32x32
output = model(x)
print(output.shape)  # torch.Size([4, 2])
```

When to use 3D convolutions:

- Medical imaging (CT scans, MRI)
- Video (treat time as 3rd spatial dimension)
- 3D physical simulations
- Molecular structure analysis

Warning: 3D convolutions are very memory-intensive! Use smaller batch sizes.

7.3.7 Transposed Convolutions (Upsampling)

Transposed convolutions (sometimes called "deconvolutions") **increase** spatial dimensions. Used in:

- Autoencoders (decoder part)
- Generative models (GANs, VAEs)
- Semantic segmentation (U-Net)
- Super-resolution

```
# Regular convolution (downsampling)
conv = nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1)
x = torch.randn(1, 64, 32, 32)
out = conv(x)
print(out.shape)  # torch.Size([1, 128, 16, 16]) - halved

# Transposed convolution (upsampling)
deconv = nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2,
    padding=1,
                            output_padding=1)
out2 = deconv(out)
print(out2.shape)  # torch.Size([1, 64, 32, 32]) - doubled
    back
```

Output size calculation:

$$\text{Output size} = (\text{Input size} - 1) \times \text{Stride} - 2 \times \text{Padding} + \text{Kernel size} + \text{Output padding}$$

Simple U-Net style autoencoder:

```
class SimpleAutoencoder(nn.Module):
    """Convolutional autoencoder for images."""

    def __init__(self):
        super().__init__()

        # Encoder (downsampling)
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 32, 3, stride=2, padding=1),  # 28
    -> 14
            nn.ReLU(),
            nn.Conv2d(32, 64, 3, stride=2, padding=1), # 14
    -> 7
            nn.ReLU(),
            nn.Conv2d(64, 128, 3, stride=2, padding=1), # 7
    -> 4 (rounds up)
            nn.ReLU(),
        )

        # Decoder (upsampling)
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(128, 64, 3, stride=2, padding
    =1,
                               output_padding=1),  # 4 -> 7
            nn.ReLU(),
            nn.ConvTranspose2d(64, 32, 3, stride=2, padding
    =1,
                               output_padding=1),  # 7 -> 14
            nn.ReLU(),
            nn.ConvTranspose2d(32, 1, 3, stride=2, padding=1,
```

```
                                    output_padding=1),  # 14 -> 28
            nn.Sigmoid()  # Output in [0, 1]
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded

# Test
model = SimpleAutoencoder()
x = torch.randn(4, 1, 28, 28)
reconstructed = model(x)
print(reconstructed.shape)  # torch.Size([4, 1, 28, 28])
```

[Checkerboard Artifacts] Transposed convolutions can create checkerboard artifacts (visible patterns in output).

Solution: Use **resize + convolution** instead:

```
# Instead of ConvTranspose2d
nn.ConvTranspose2d(64, 32, kernel_size=4, stride=2, padding
    =1)

# Use this (better quality)
nn.Sequential(
    nn.Upsample(scale_factor=2, mode='bilinear',
    align_corners=False),
    nn.Conv2d(64, 32, kernel_size=3, padding=1)
)
```

7.3.8 Dilated Convolutions

Dilated (atrous) convolutions increase receptive field without increasing parameters or losing resolution.

```
# Standard 3x3 convolution
conv_standard = nn.Conv2d(64, 64, kernel_size=3, padding=1)
# Receptive field: 3x3

# Dilated 3x3 convolution (dilation=2)
conv_dilated = nn.Conv2d(64, 64, kernel_size=3, padding=2,
    dilation=2)
# Receptive field: 5x5 (with gaps)
# Same parameters as standard 3x3!

# Effective kernel size: k + (k-1)*(d-1)
# For k=3, d=2: 3 + (3-1)*(2-1) = 5
```

When to use dilated convolutions:

- Semantic segmentation (maintain resolution while growing receptive field)
- Dense prediction tasks
- When you need large receptive field but want to keep resolution

7.3.9 Common CNN Architectures

VGG-style (stacking blocks):

```python
def vgg_block(in_channels, out_channels, num_convs):
    """VGG-style block: multiple convs + pool."""
    layers = []
    for _ in range(num_convs):
        layers.append(nn.Conv2d(in_channels, out_channels,
                                kernel_size=3, padding=1))
        layers.append(nn.ReLU())
        in_channels = out_channels
    layers.append(nn.MaxPool2d(2, 2))
    return nn.Sequential(*layers)

class VGGStyleNet(nn.Module):
    def __init__(self, num_classes=10):
        super().__init__()

        self.features = nn.Sequential(
            vgg_block(3, 64, 2),    # 2 convs, 64 filters
            vgg_block(64, 128, 2),  # 2 convs, 128 filters
            vgg_block(128, 256, 3), # 3 convs, 256 filters
            vgg_block(256, 512, 3), # 3 convs, 512 filters
        )

        self.classifier = nn.Sequential(
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(512, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = self.classifier(x)
        return x
```

Key design patterns:

1. **Double channels after pooling:** $64 \to 128 \to 256 \to 512$
2. **Multiple convs per block:** Learn richer features before downsampling
3. **Small kernels (3×3):** More efficient than large kernels
4. **Batch norm after conv:** Stabilizes training
5. **Global pooling:** More flexible than fixed FC input size

7.4 Implementation: Debugging CNNs

7.4.1 Visualizing Feature Maps

```python
def visualize_feature_maps(model, x, layer_num=0):
    """Visualize activations from a specific layer."""
    activation = {}

    def get_activation(name):
        def hook(model, input, output):
            activation[name] = output.detach()
        return hook

```

```
    # Register hook
    layer_name = f'features.{layer_num}'
    handle = dict(model.named_modules())[layer_name].
    register_forward_hook(
        get_activation(layer_name))

    # Forward pass
    model.eval()
    with torch.no_grad():
        _ = model(x)

    handle.remove()

    # Visualize
    act = activation[layer_name].squeeze()  # Remove batch
    dim

    import matplotlib.pyplot as plt
    fig, axes = plt.subplots(4, 8, figsize=(16, 8))
    for i, ax in enumerate(axes.flat):
        if i < act.shape[0]:
            ax.imshow(act[i].cpu(), cmap='viridis')
            ax.axis('off')
    plt.tight_layout()
    plt.show()

# Usage
# model = ModernCNN()
# x = torch.randn(1, 1, 28, 28)
# visualize_feature_maps(model, x, layer_num=0)
```

7.4.2 Common CNN Debugging Issues

Problem: Output shape wrong

```
# Debug shape through network
def debug_shapes(model, input_shape):
    """Print shapes through the network."""
    x = torch.randn(*input_shape)
    print(f"Input: {x.shape}")

    for name, module in model.named_children():
        x = module(x)
        print(f"{name}: {x.shape}")

# Usage
model = SimpleCNN()
debug_shapes(model, (1, 1, 28, 28))
"""
Input: torch.Size([1, 1, 28, 28])
conv1: torch.Size([1, 32, 28, 28])
pool: torch.Size([1, 32, 14, 14])
...
"""
```

Problem: Memory issues with 3D convolutions

```
# Use gradient checkpointing for large models
from torch.utils.checkpoint import checkpoint

class MemoryEfficientCNN(nn.Module):
```

```
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv3d(1, 32, 3)
        self.conv2 = nn.Conv3d(32, 64, 3)
        # ... more layers

    def forward(self, x):
        # Checkpoint expensive layers
        x = checkpoint(self.conv1, x)
        x = checkpoint(self.conv2, x)
        return x
```

Tips for reducing memory:

- Use smaller batch sizes
- Use mixed precision training (FP16)
- Reduce number of filters
- Use gradient checkpointing
- Process data in patches (for very large images)

7.5 Exercises

7.1: Basic CNN for MNIST - ★★

Goal: Build and train your first CNN.

1. Build a CNN for MNIST: 2 conv layers + 2 FC layers
2. Use 32 and 64 filters, 3×3 kernels, ReLU, max pooling
3. Train for 5 epochs
4. Achieve >95% test accuracy
5. Visualize some predictions

Starter code:

```
from torchvision import datasets, transforms

# Load MNIST
transform = transforms.ToTensor()
train_dataset = datasets.MNIST('./data', train=True,
                               download=True, transform=
    transform)
test_dataset = datasets.MNIST('./data', train=False,
    transform=transform)

# Your CNN here
class MyCNN(nn.Module):
    def __init__(self):
        super().__init__()
        # Your architecture

    def forward(self, x):
        # Your forward pass
        pass
```

7.2: Shape Calculation Practice - ★★

Goal: Master output size calculations.

For a 32×32 input image, calculate output sizes for:

1. Conv(3×3, stride=1, padding=1)
2. Conv(5×5, stride=2, padding=2)
3. Conv(3×3, stride=1, padding=0) followed by MaxPool(2×2)
4. Three consecutive Conv(3×3, stride=1, padding=1) layers
5. Conv(3×3, stride=1, padding=1, dilation=2)

Then implement these layers and verify your calculations with actual tensors.

7.3: 1D CNN for Time Series - ★★★

Goal: Apply convolutions to sequence data.

1. Generate synthetic time series: $y = \sin(x) + 0.5\sin(3x) + \text{noise}$
2. Create dataset: windows of 100 points predict next 10 points
3. Build a 1D CNN: 3 conv layers with increasing filters
4. Train and compare with a simple MLP baseline
5. Visualize predictions on test data

Hint: Use `nn.Conv1d` with kernel_size=7 or 9.

7.4: VGG-Style Network - ★★★

Goal: Build a deeper network with proper architecture.

1. Implement a VGG-style network:
   - Block 1: 2×Conv(64) + Pool
   - Block 2: 2×Conv(128) + Pool
   - Block 3: 3×Conv(256) + Pool
   - Classifier: Global pooling + FC layers
2. Add batch normalization after each conv
3. Train on CIFAR-10
4. Achieve >70% test accuracy

Challenge: Add dropout and compare with/without it.

7.5: Simple U-Net - ★★★★

Goal: Implement encoder-decoder architecture.

Build a simple U-Net for image denoising:

1. Encoder: 3 conv blocks with downsampling
2. Decoder: 3 transposed conv blocks with upsampling
3. Add skip connections (concatenate encoder features with decoder)
4. Train on MNIST with added Gaussian noise
5. Visualize: noisy input → denoised output → clean target

Architecture:

```
Encoder:        Decoder:
1 -> 32 -------> concat -> 32 -> 1
     |                       |
32 -> 64 -------> concat -> 64 -> 32
     |                       |
64 -> 128 -----> 128 -> 64
```

Starter code:

```python
class SimpleUNet(nn.Module):
    def __init__(self):
        super().__init__()
        # Encoder
        self.enc1 = self.conv_block(1, 32)
        self.enc2 = self.conv_block(32, 64)
        self.enc3 = self.conv_block(64, 128)

        # Decoder
        self.dec3 = self.upconv_block(128, 64)
        self.dec2 = self.upconv_block(128, 32)  # 128 = 64 + 64 from skip
        self.dec1 = nn.Conv2d(64, 1, 1)  # 64 = 32 + 32 from skip

        self.pool = nn.MaxPool2d(2)

    def conv_block(self, in_ch, out_ch):
        return nn.Sequential(
            nn.Conv2d(in_ch, out_ch, 3, padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU()
        )

```

```
    def upconv_block(self, in_ch, out_ch):
        return nn.Sequential(
            nn.ConvTranspose2d(in_ch, out_ch, 2, stride=2),
            nn.BatchNorm2d(out_ch),
            nn.ReLU()
        )

    def forward(self, x):
        # Encoder with skip connections
        e1 = self.enc1(x)
        e2 = self.enc2(self.pool(e1))
        e3 = self.enc3(self.pool(e2))

        # Decoder with skip connections
        d3 = self.dec3(e3)
        d2 = self.dec2(torch.cat([d3, e2], dim=1))  # Skip connection
        d1 = self.dec1(torch.cat([d2, e1], dim=1))  # Skip connection

        return torch.sigmoid(d1)
```

7.6: 3D CNN for Volumetric Data - ★★★★

Goal: Work with 3D convolutions.

1. Generate synthetic 3D data: 3D Gaussian blobs as "tumors"
2. Build a 3D CNN classifier: 3 conv layers + global pooling + FC
3. Handle memory constraints (small batch size, fewer filters)
4. Achieve good classification accuracy
5. Visualize 3D volumes and predictions (use slicing)

Data generation:

```
def generate_3d_blob(size=32):
    """Generate 3D volume with central blob."""
    volume = np.zeros((size, size, size))
    center = size // 2

    # Create Gaussian blob
    for i in range(size):
        for j in range(size):
            for k in range(size):
                dist = np.sqrt((i-center)**2 + (j-center)**2 +
                               (k-center)**2)
                volume[i, j, k] = np.exp(-dist**2 / 100)

    return torch.FloatTensor(volume).unsqueeze(0)  # Add channel dim
```

Warning: Use batch_size=2 or 4 for 3D convolutions!

7.6 Key Takeaways

Why CNNs work:

- **Parameter sharing:** Drastically fewer parameters than MLPs
- **Translation invariance:** Features work regardless of position
- **Hierarchical learning:** Low-level → high-level features
- **Local connectivity:** Exploits spatial structure

Architecture design principles:

- Start with 32-64 filters, double after each pooling
- Use small kernels (3×3) rather than large ones
- Add batch normalization for deep networks
- Use strided convolutions or pooling for downsampling
- Global pooling at the end for flexibility

Convolution variants:

- **1D:** Time series, audio, text sequences
- **2D:** Images (most common)
- **3D:** Videos, medical scans, volumetric data
- **Transposed:** Upsampling in decoder/generator
- **Dilated:** Large receptive field without losing resolution

Common patterns:

- **VGG-style:** Stack conv blocks, double channels after pooling
- **Encoder-decoder:** Downsample then upsample (autoencoders, U-Net)
- **Skip connections:** Connect encoder to decoder (U-Net, ResNet)

Debugging tips:

- Always check shapes at each layer
- Visualize feature maps to understand what network learns
- Start small (few layers, few filters) then scale up
- Use `AdaptiveAvgPool2d` for flexibility
- Watch out for memory with 3D convolutions

8 Residual Networks & Skip Connections

8.1 Introduction: The Deep Network Problem

In the early 2010s, researchers discovered something puzzling: making networks deeper didn't always make them better. Sometimes, a 56-layer network performed **worse** than a 20-layer network—even on training data!

This wasn't overfitting (test loss was also worse). It was a fundamental optimization problem.

The Degradation Problem:

- Very deep networks are harder to optimize
- Gradients vanish or explode
- Training error increases with depth (counterintuitively)
- Not caused by overfitting—happens on training set too

Residual Networks (ResNets) solved this in 2015, enabling networks with 100+ layers that actually train better than shallow ones. The key insight: **skip connections** (also called **residual connections**).

Impact:

- ResNet-152 won ImageNet 2015 (3.57% error, human-level performance)
- Now standard in almost all vision architectures
- Fundamental principle used across all deep learning (Transformers, etc.)

8.2 Theory: Why Skip Connections Work

8.2.1 The Problem: Vanishing Gradients

In a deep network without skip connections:

$$\mathbf{x}_L = f_L(f_{L-1}(...f_2(f_1(\mathbf{x}_0))))$$

During backpropagation, gradients multiply through the chain rule:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_0} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_L} \cdot \frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_{L-1}} \cdot ... \cdot \frac{\partial \mathbf{x}_2}{\partial \mathbf{x}_1} \cdot \frac{\partial \mathbf{x}_1}{\partial \mathbf{x}_0}$$

If each term is less than 1, the gradient **vanishes** exponentially as it propagates backward.

Example: If each layer multiplies gradient by 0.9:

- After 10 layers: $0.9^{10} \approx 0.35$
- After 20 layers: $0.9^{20} \approx 0.12$
- After 50 layers: $0.9^{50} \approx 0.005$ (essentially zero!)

Early layers barely learn, making deep networks ineffective.

8.2.2 The Solution: Residual Connections

Instead of learning a direct mapping $\mathbf{y} = \mathcal{F}(\mathbf{x})$, learn a **residual**:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}) + \mathbf{x}$$

The network learns the **difference** (residual) from the identity mapping.

[Key Insight: Identity Mapping] With a skip connection, the network can learn the identity function by simply setting $\mathcal{F}(\mathbf{x}) = 0$ (all weights to zero).

Why this matters:

- Identity is easy to learn (do nothing)
- Network starts with identity, then learns refinements
- Even if $\mathcal{F}$ is poorly initialized, gradients still flow through skip connection
- Optimization becomes: "How should I modify the input?" instead of "What should the output be?"

8.2.3 Gradient Flow Through Skip Connections

During backpropagation with skip connections:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}} = \frac{\partial \mathcal{L}}{\partial \mathbf{y}} \left(\frac{\partial \mathcal{F}}{\partial \mathbf{x}} + 1 \right)$$

The +1 term creates a **gradient highway**:

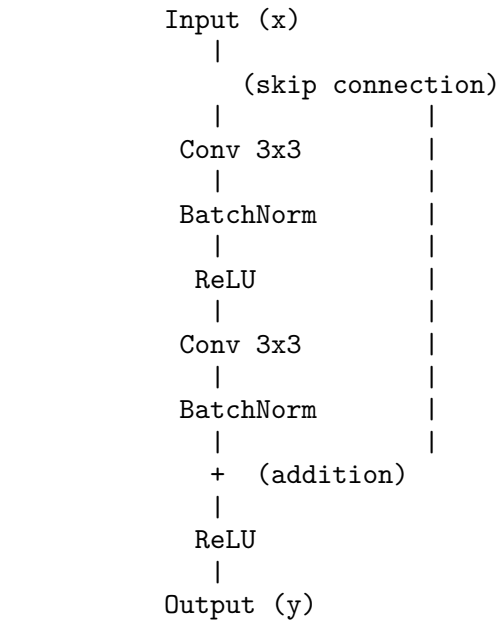
- Gradients always have a direct path backward (through the +1)
- Even if $\frac{\partial \mathcal{F}}{\partial \mathbf{x}} \approx 0$, gradients still flow
- No vanishing gradient problem!

Analogy: Regular network = narrow mountain trail (one path, easy to get stuck)

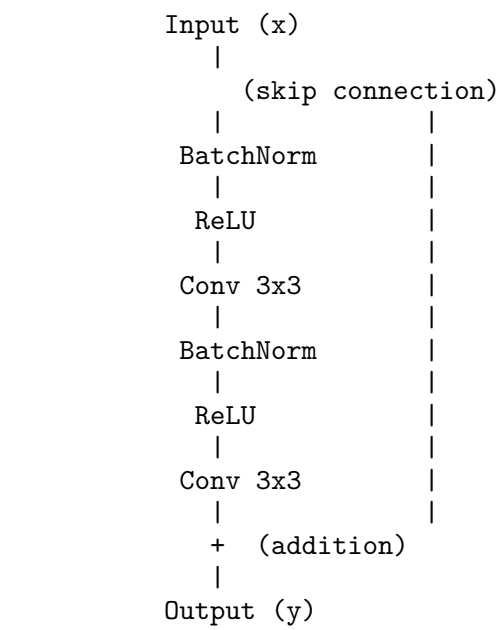
Skip connections = trail + highway (gradient can always get through)

8.2.4 Residual Block Variants

Basic Residual Block (Original):



Pre-Activation Residual Block (Improved):



Difference:

- **Post-activation:** Conv → BN → ReLU (then add)
- **Pre-activation:** BN → ReLU → Conv (then add)

Pre-activation advantages:

- Better gradient flow (identity mapping is truly unimpeded)
- Easier optimization
- Better for very deep networks (>100 layers)

8.2.5 When Dimensions Don't Match

If input and output have different dimensions, use a **projection shortcut**:

$$\mathbf{y} = \mathcal{F}(\mathbf{x}) + \mathbf{W}_s\mathbf{x}$$

where $\mathbf{W}_s$ is a linear projection (1×1 convolution).

Two approaches:

1. **Identity shortcut + zero padding:** Pad channels with zeros (parameter-free)
2. **Projection shortcut:** Use 1×1 conv to match dimensions (adds parameters)

Most implementations use projection shortcuts when dimensions change.

8.3 Implementation: Building Residual Blocks

8.3.1 Basic Residual Block

```python
import torch
import torch.nn as nn

class BasicBlock(nn.Module):
    """Basic residual block with two 3x3 convolutions."""

    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()

        # Main path
        self.conv1 = nn.Conv2d(in_channels, out_channels,
                               kernel_size=3, stride=stride,
    padding=1,
                               bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)

        self.conv2 = nn.Conv2d(out_channels, out_channels,
                               kernel_size=3, stride=1,
    padding=1,
                               bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Skip connection (identity or projection)
        self.skip = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.skip = nn.Sequential(
                nn.Conv2d(in_channels, out_channels,
                          kernel_size=1, stride=stride, bias=
    False),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        identity = x

        # Main path
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
```

```
        out = self.bn2(out)

        # Skip connection
        identity = self.skip(identity)

        # Add residual
        out += identity
        out = self.relu(out)

        return out

# Test
block = BasicBlock(64, 64)
x = torch.randn(4, 64, 32, 32)
y = block(x)
print(y.shape)  # torch.Size([4, 64, 32, 32])

# Test with dimension change
block_downsample = BasicBlock(64, 128, stride=2)
y2 = block_downsample(x)
print(y2.shape)  # torch.Size([4, 128, 16, 16])
```

[Why bias=False?] When using batch normalization immediately after convolution, the bias is redundant (BatchNorm has its own bias term). Setting `bias=False` saves parameters and computation without affecting performance.

8.3.2 Pre-Activation Residual Block

```
class PreActBlock(nn.Module):
    """Pre-activation residual block (improved version)."""

    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()

        # Pre-activation
        self.bn1 = nn.BatchNorm2d(in_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv1 = nn.Conv2d(in_channels, out_channels,
                               kernel_size=3, stride=stride,
    padding=1,
                               bias=False)

        self.bn2 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels,
                               kernel_size=3, stride=1,
    padding=1,
                               bias=False)

        # Skip connection
        self.skip = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.skip = nn.Sequential(
                nn.Conv2d(in_channels, out_channels,
                          kernel_size=1, stride=stride, bias=
    False)
            )

    def forward(self, x):
        # Pre-activation
        out = self.bn1(x)
        out = self.relu(out)

        # Save for skip connection (after activation)
        identity = self.skip(out)

        # First conv
        out = self.conv1(out)

        # Second pre-activation + conv
        out = self.bn2(out)
        out = self.relu(out)
        out = self.conv2(out)

        # Add residual (no activation after addition!)
        out += identity

        return out
```

Key difference: In pre-activation, the final addition has no activation. The next block will apply activation first.

8.3.3 Bottleneck Block

For deeper networks (ResNet-50+), use bottleneck blocks to reduce parameters:

```
class BottleneckBlock(nn.Module):
    """
    Bottleneck block: 1x1 -> 3x3 -> 1x1
```

```
    """Reduces parameters by using 1x1 convolutions to reduce/expand channels.
    """
    expansion = 4  # Output channels = input channels * 4

    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()

        # Bottleneck: reduce channels
        self.conv1 = nn.Conv2d(in_channels, out_channels,
                               kernel_size=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)

        # Main 3x3 conv
        self.conv2 = nn.Conv2d(out_channels, out_channels,
                               kernel_size=3, stride=stride,
                               padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Expand channels
        self.conv3 = nn.Conv2d(out_channels, out_channels * self.expansion,
                               kernel_size=1, bias=False)
        self.bn3 = nn.BatchNorm2d(out_channels * self.expansion)

        self.relu = nn.ReLU(inplace=True)

        # Skip connection
        self.skip = nn.Sequential()
        if stride != 1 or in_channels != out_channels * self.expansion:
            self.skip = nn.Sequential(
                nn.Conv2d(in_channels, out_channels * self.expansion,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels * self.expansion)
            )

    def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        identity = self.skip(identity)
        out += identity
        out = self.relu(out)

        return out

# Compare parameters
basic = BasicBlock(256, 256)
```

```
bottleneck = BottleneckBlock(256, 64)  # Output: 64*4 = 256

basic_params = sum(p.numel() for p in basic.parameters())
bottleneck_params = sum(p.numel() for p in bottleneck.
    parameters())

print(f"Basic block: {basic_params:,} parameters")
print(f"Bottleneck: {bottleneck_params:,} parameters")
# Basic block: ~590,000 parameters
# Bottleneck: ~70,000 parameters (much fewer!)
```

8.3.4 Building a Complete ResNet

```python
class ResNet(nn.Module):
    """ResNet architecture for CIFAR-10."""

    def __init__(self, block, num_blocks, num_classes=10):
        """
        Args:
            block: BasicBlock or BottleneckBlock
            num_blocks: List of number of blocks per layer
            num_classes: Number of output classes
        """
        super().__init__()
        self.in_channels = 64

        # Initial convolution
        self.conv1 = nn.Conv2d(3, 64, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)

        # Residual layers
        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=2)

        # Global pooling and classifier
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * block.expansion, num_classes)

    def _make_layer(self, block, out_channels, num_blocks, stride):
        """Create a layer with multiple residual blocks."""
        strides = [stride] + [1] * (num_blocks - 1)
        layers = []

        for stride in strides:
            layers.append(block(self.in_channels, out_channels, stride))
            self.in_channels = out_channels * block.expansion

        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)

        out = self.avgpool(out)
        out = out.view(out.size(0), -1)
```

```
        out = self.fc(out)

        return out

# ResNet-18
def ResNet18(num_classes=10):
    return ResNet(BasicBlock, [2, 2, 2, 2], num_classes)

# ResNet-34
def ResNet34(num_classes=10):
    return ResNet(BasicBlock, [3, 4, 6, 3], num_classes)

# ResNet-50 (uses bottleneck)
def ResNet50(num_classes=10):
    # Note: BottleneckBlock needs expansion=4
    BottleneckBlock.expansion = 4
    return ResNet(BottleneckBlock, [3, 4, 6, 3], num_classes)

# Test
model = ResNet18()
x = torch.randn(4, 3, 32, 32)
y = model(x)
print(y.shape)  # torch.Size([4, 10])

# Count parameters
total_params = sum(p.numel() for p in model.parameters())
print(f"ResNet-18: {total_params:,} parameters")
```

[ResNet Naming] ResNet-N refers to the total number of weight layers:

ResNet-18: 1 initial conv + 4 layers $\times$ 2 blocks $\times$ 2 convs + 1 FC = 18 layers

ResNet-34: 1 + (3+4+6+3) $\times$ 2 + 1 = 34 layers

ResNet-50: 1 + (3+4+6+3) $\times$ 3 + 1 = 50 layers (bottleneck has 3 convs)

Deeper = more capacity but slower training/inference.

8.3.5 Dense Connections (DenseNet)

DenseNet takes skip connections further: **every layer connects to every other layer**.

```python
class DenseBlock(nn.Module):
    """Dense block where each layer receives all previous layers as input."""

    def __init__(self, in_channels, growth_rate, num_layers):
        """
        Args:
            in_channels: Initial number of channels
            growth_rate: Number of channels added per layer
            num_layers: Number of layers in the dense block
        """
        super().__init__()

        self.layers = nn.ModuleList()
        for i in range(num_layers):
            self.layers.append(
                self._make_dense_layer(in_channels + i * growth_rate,
                                       growth_rate)
            )

    def _make_dense_layer(self, in_channels, growth_rate):
        return nn.Sequential(
            nn.BatchNorm2d(in_channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels, growth_rate, kernel_size=3,
                      padding=1, bias=False)
        )

    def forward(self, x):
        features = [x]
        for layer in self.layers:
            # Concatenate all previous feature maps
            new_features = layer(torch.cat(features, dim=1))
            features.append(new_features)

        # Return concatenation of all features
        return torch.cat(features, dim=1)

# Test
block = DenseBlock(in_channels=64, growth_rate=32, num_layers=4)
x = torch.randn(2, 64, 16, 16)
y = block(x)
print(y.shape)  # torch.Size([2, 192, 16, 16])
# 192 = 64 (input) + 4 * 32 (4 layers, each adds 32 channels)
```

DenseNet characteristics:

- **Feature reuse:** All layers have access to all previous features
- **Fewer parameters:** Smaller growth_rate than ResNet channel counts
- **Memory intensive:** Must store all intermediate features
- **Better gradient flow:** Even better than ResNet

When to use DenseNet vs ResNet:

- **DenseNet:** When parameter efficiency is critical, smaller datasets
- **ResNet:** When memory is limited, faster training/inference, more standard

8.3.6 Skip Connections in Practice

General principles for adding skip connections:

1. **Add skip every 2-3 layers:** Balance between gradient flow and complexity
2. **Match dimensions:** Use projection when channels or spatial size change
3. **Identity when possible:** Projection-free shortcuts are better
4. **Add before final activation:** Preserves identity mapping
5. **Don't overdo it:** Too many skip connections can hurt performance

Example: Adding skip connections to existing MLP:

```
class MLPWithSkipConnections(nn.Module):
    """MLP with residual connections."""

    def __init__(self, input_dim, hidden_dim, output_dim,
    num_layers=4):
        super().__init__()

        self.input_layer = nn.Linear(input_dim, hidden_dim)

        # Hidden layers with skip connections
        self.hidden_layers = nn.ModuleList()
        for _ in range(num_layers):
            self.hidden_layers.append(
                nn.Sequential(
                    nn.Linear(hidden_dim, hidden_dim),
                    nn.ReLU(),
                    nn.Linear(hidden_dim, hidden_dim)
                )
            )

        self.output_layer = nn.Linear(hidden_dim, output_dim)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.input_layer(x))

        # Apply each hidden layer with skip connection
        for layer in self.hidden_layers:
            residual = x
            x = layer(x)
            x = x + residual  # Skip connection
            x = self.relu(x)

        x = self.output_layer(x)
        return x
```

8.4 Implementation: Debugging ResNets

8.4.1 Verifying Skip Connections

```python
def verify_skip_connections(model, input_shape=(1, 3, 32, 32)):
    """Verify that skip connections are working."""
    x = torch.randn(*input_shape, requires_grad=True)

    # Forward pass
    output = model(x)
    loss = output.sum()

    # Backward pass
    loss.backward()

    # Check if gradients reach input
    if x.grad is not None:
        print(f"    Gradients flow to input: {x.grad.abs().mean().item():.6f}")
    else:
        print("    No gradients at input!")

    # Check gradient magnitudes throughout network
    print("\nGradient magnitudes by layer:")
    for name, param in model.named_parameters():
        if param.grad is not None:
            grad_mean = param.grad.abs().mean().item()
            grad_max = param.grad.abs().max().item()
            print(f"{name:30s}: mean={grad_mean:.6f}, max={grad_max:.6f}")

# Usage
# model = ResNet18()
# verify_skip_connections(model)
```

8.4.2 Comparing With and Without Skip Connections

```python
class PlainCNN(nn.Module):
    """Same architecture as ResNet but without skip connections."""

    def __init__(self, num_classes=10):
        super().__init__()

        self.conv1 = nn.Conv2d(3, 64, 3, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(64)

        # Same structure as ResNet-18 but no skip connections
        self.layers = nn.Sequential(
            # Layer 1
            nn.Conv2d(64, 64, 3, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, 64, 3, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(),

            # Layer 2 (with stride=2 downsampling)
            nn.Conv2d(64, 128, 3, stride=2, padding=1, bias=False),
```

```
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(128, 128, 3, padding=1, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(),

            # Continue pattern...
        )

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)

    def forward(self, x):
        x = nn.ReLU()(self.bn1(self.conv1(x)))
        x = self.layers(x)
        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        return self.fc(x)

# Compare training curves
# plain_model = PlainCNN()
# resnet_model = ResNet18()
# Train both and plot losses - ResNet should converge faster and better
```

8.5 Exercises

8.1: Basic Residual Block - ★★

Goal: Implement and understand a basic residual block.

1. Implement a `BasicBlock` with two 3×3 convolutions
2. Include skip connection with projection for dimension changes
3. Test with different input/output channels:
   - Same channels (64 → 64)
   - Different channels (64 → 128)
   - With downsampling (stride=2)
4. Verify output shapes match expected dimensions

Starter code:

```
class MyBasicBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        # Your code here

    def forward(self, x):
        # Your code here
        pass

# Test cases
block1 = MyBasicBlock(64, 64, stride=1)
block2 = MyBasicBlock(64, 128, stride=2)

x = torch.randn(4, 64, 32, 32)
print(block1(x).shape)  # Should be [4, 64, 32, 32]
print(block2(x).shape)  # Should be [4, 128, 16, 16]
```

8.2: Plain vs Residual Comparison - ★★★

Goal: Empirically verify that skip connections help.

1. Build two identical networks (10 layers each):
   - One **without** skip connections (plain CNN)
   - One **with** skip connections (ResNet-style)
2. Train both on CIFAR-10 or MNIST
3. Track and plot:
   - Training loss
   - Validation accuracy
   - Gradient magnitudes in early layers
4. Observe: ResNet should train faster and achieve better accuracy

Questions to answer:

- How much faster does ResNet converge?
- What's the final accuracy difference?
- How do gradient magnitudes compare?

8.3: Pre-Activation vs Post-Activation - ★★★

Goal: Compare residual block variants.

1. Implement both post-activation and pre-activation blocks
2. Build identical ResNets with each type
3. Train on the same dataset
4. Compare:
   - Training stability (loss curves)
   - Final accuracy
   - Training time

Hypothesis: Pre-activation should be slightly better, especially for deeper networks.
Extension: Try with different depths (18, 34, 50 layers) and see if the gap widens.

8.4: Building ResNet-18 - ★★★

Goal: Build a complete ResNet from scratch.

1. Implement ResNet-18 architecture:
   - Initial 7×7 conv (or 3×3 for CIFAR)
   - 4 residual layers with [2, 2, 2, 2] blocks
   - Channels: 64 → 128 → 256 → 512
   - Global average pooling + FC
2. Train on CIFAR-10
3. Achieve >90% test accuracy
4. Save the model and load it for inference

Bonus: Visualize learned features in first convolutional layer.

8.5: Bottleneck Block - ★★★★

Goal: Implement parameter-efficient bottleneck blocks.

1. Implement `BottleneckBlock`: 1×1 → 3×3 → 1×1

2. Compare parameter count with `BasicBlock` for same input/output
3. Build ResNet-50 using bottleneck blocks
4. Train and compare with ResNet-18:
   - Accuracy (ResNet-50 should be better)
   - Training time (ResNet-50 slower)
   - Memory usage

Challenge: How deep can you go before hitting memory limits? Try ResNet-101.

8.6: Skip Connections in U-Net - ★★★★

Goal: Apply skip connections to encoder-decoder architectures.

1. Take the U-Net from CNN section (Exercise 7.5)
2. Add residual blocks in encoder and decoder paths
3. Train on image denoising task
4. Compare with original U-Net:
   - Denoising quality (PSNR/SSIM)
   - Training stability
   - Convergence speed

Architecture:

- Replace conv blocks with residual blocks
- Keep skip connections between encoder/decoder
- Use both residual blocks AND U-Net skip connections

8.7: Dense Block Implementation - ★★★★

Goal: Implement and understand DenseNet.

1. Implement a `DenseBlock` where each layer receives all previous layers
2. Implement a transition layer (1×1 conv + pooling) to reduce channels
3. Build a small DenseNet:
   - 3 dense blocks with growth_rate=12
   - Transition layers between blocks
   - Final global pooling + FC
4. Compare with ResNet of similar depth:
   - Parameter count (DenseNet should be fewer)
   - Memory usage (DenseNet higher during training)
   - Accuracy

Hint: Use `torch.cat()` to concatenate feature maps along channel dimension.

8.6 Key Takeaways

The fundamental problem:

- Very deep networks suffer from degradation (not just overfitting)
- Vanishing gradients make early layers hard to train
- Optimization becomes difficult, not just capacity

How skip connections solve it:

- Create gradient highways: $\frac{\partial \mathcal{L}}{\partial \mathbf{x}}$ always has +1 term
- Learn residuals instead of direct mappings (easier optimization)
- Enable training of 100+ layer networks
- Identity mapping is easy to learn (set weights to zero)

Residual block variants:

- **Post-activation:** Conv → BN → ReLU, then add (original)
- **Pre-activation:** BN → ReLU → Conv, then add (improved)
- **Bottleneck:** 1×1 → 3×3 → 1×1 (for deeper networks, fewer parameters)
- **Dense:** Every layer connects to all previous layers (extreme skip connections)

Implementation guidelines:

- Use projection shortcuts when dimensions change
- Set `bias=False` when using BatchNorm
- Add skip connections every 2-3 layers
- Pre-activation is better for very deep networks (>100 layers)
- Use bottleneck blocks for ResNet-50+

When to use skip connections:

- **Always use for deep networks (>20 layers)**
- **Vision tasks:** ResNet is standard backbone
- **U-Net style architectures:** Encoder-decoder with skip connections
- **Any architecture where gradient flow is critical**
- **When training is unstable or slow**

Architecture choices:

Model	Layers	Use Case	Parameters
ResNet-18	18	General purpose, fast	11M
ResNet-34	34	More capacity	21M
ResNet-50	50	High accuracy	25M
ResNet-101	101	Maximum accuracy	44M
DenseNet-121	121	Parameter efficient	8M

Table 5: Popular ResNet/DenseNet architectures

Impact on deep learning:

- Enabled training of 100+ layer networks
- Now standard in almost all vision models

- Extended to other domains (Transformers use similar ideas)
- Fundamental principle: make it easy for network to learn identity

Debugging tips:

- Verify gradients flow to early layers
- Compare with plain network (without skip connections)
- Check gradient magnitudes throughout network
- Visualize activations in residual blocks
- If training fails, check skip connection implementation

Common mistakes:

1. Forgetting projection when dimensions change
2. Applying activation after addition in pre-activation blocks
3. Not using BatchNorm (skip connections alone aren't enough)
4. Making skip connection too complex (keep it identity when possible)
5. Adding too many skip connections (can hurt performance)

[Rule of Thumb] **For networks >20 layers:** Always use residual connections

For networks >50 layers: Use bottleneck blocks

For networks >100 layers: Use pre-activation blocks

When in doubt: Use ResNet as your starting point and modify from there.

9 Batch Normalization & Layer Normalization

9.1 Introduction: Why Normalization Matters

Training deep networks is hard. As gradients flow backward, they can vanish or explode. As data flows forward, the distribution of activations can shift. These issues slow training and make networks sensitive to initialization.

Normalization techniques stabilize training by controlling the distribution of activations.

Key benefits:

- **Faster training:** Can use higher learning rates (2-10× speedup)
- **Less sensitivity to initialization:** Network more robust
- **Regularization effect:** Acts like dropout (slight noise from batch statistics)
- **Enables deeper networks:** Makes 100+ layer networks trainable

When you need normalization:

- Deep networks (>10 layers)
- Training is slow or unstable
- Network sensitive to learning rate or initialization
- Using high learning rates

9.2 Theory: Batch Normalization

9.2.1 The Problem: Internal Covariate Shift

As we train, the distribution of inputs to each layer changes. This is called **internal covariate shift**.

Example: Consider layer 3 of a network.

- Early in training: inputs to layer 3 might have mean=0.5, std=0.3
- After 100 steps: mean=2.1, std=1.8 (distribution has shifted!)

Each layer must constantly adapt to the changing distribution. This slows learning.

Batch Normalization's solution: Normalize inputs to each layer to have consistent statistics.

9.2.2 How Batch Normalization Works

Given a batch of inputs $\mathbf{x} = \{x_1, x_2, \ldots, x_m\}$:

Step 1: Compute batch statistics

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \quad \text{(batch mean)}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \quad \text{(batch variance)}$$

Step 2: Normalize

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

where ϵ is a small constant (typically 10^{-5}) for numerical stability.

Step 3: Scale and shift (learnable parameters)

$$y_i = \gamma \hat{x}_i + \beta$$

where γ (scale) and β (shift) are learned parameters.

Why scale and shift? The network can learn to undo the normalization if needed! If $\gamma = \sqrt{\sigma_B^2}$ and $\beta = \mu_B$, we recover the original distribution.

[Key Insight] Batch norm has two modes:

Training: Use batch statistics (μ_B, σ_B^2)

- Computed from current batch
- Adds stochasticity (different batches $\rightarrow$ different statistics)
- Acts as regularization

Inference: Use running statistics ($\mu_{running}$, $\sigma_{running}^2$)

- Exponential moving average from training
- Deterministic (same input $\rightarrow$ same output)
- No batch dependency

During training, PyTorch maintains:

$$\mu_{running} \leftarrow (1 - \text{momentum}) \cdot \mu_{running} + \text{momentum} \cdot \mu_B$$

Default momentum = 0.1.

9.2.3 Why Batch Normalization Helps

1. Smooths the optimization landscape

Batch norm makes the loss surface smoother, allowing larger learning rates.

2. Reduces internal covariate shift

Each layer receives inputs with consistent statistics.

3. Acts as regularization

Noise from batch statistics acts like dropout (but usually weaker).

4. Reduces sensitivity to initialization

Even with poor initialization, batch norm helps normalize activations.

5. Allows higher learning rates

The smoothing effect permits learning rates 10× higher than without batch norm.

9.2.4 Limitations of Batch Normalization

1. Requires large enough batches

With batch_size < 8, batch statistics are unreliable. Solution: Use LayerNorm or GroupNorm.

2. Different behavior in train/eval

Must remember to call `model.train()` and `model.eval()`. Forgetting this is a common bug!

3. Doesn't work well for RNNs

Batch statistics across sequence positions don't make sense. Solution: Use LayerNorm.

4. Coupling between samples in batch

Each sample's normalization depends on other samples in the batch. Can cause issues in some scenarios.

9.3 Theory: Layer Normalization

Layer Normalization normalizes across features instead of across the batch.

Batch Norm: Normalize each feature across the batch

$$\hat{x}_{ij} = \frac{x_{ij} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}} \quad \text{where } \mu_j = \frac{1}{m}\sum_{i=1}^{m} x_{ij}$$

(Averages over batch dimension i for each feature j)

Layer Norm: Normalize all features for each sample

$$\hat{x}_{ij} = \frac{x_{ij} - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} \quad \text{where } \mu_i = \frac{1}{d}\sum_{j=1}^{d} x_{ij}$$

(Averages over feature dimension j for each sample i)

Key differences:

Aspect	**Batch Norm**	**Layer Norm**
Normalizes over	Batch dimension	Feature dimension
Batch size dependency	Yes (fails for small batches)	No (works with batch=1)
Train/eval modes	Different	Same
Learnable params	γ, β per feature	γ, β per feature
Best for	CNNs, MLPs	RNNs, Transformers
Running statistics	Yes	No

Table 6: Batch Norm vs Layer Norm

When to use Layer Norm:

- RNNs and sequence models
- Transformers (standard choice)
- Small batch sizes (batch_size < 8)
- Online learning (single sample at a time)
- When you want same behavior in train/eval

9.4 Implementation: Using Normalization in PyTorch

9.4.1 Batch Normalization Usage

```python
import torch
import torch.nn as nn

# For 1D data (MLPs, sequences)
bn1d = nn.BatchNorm1d(num_features=128)

# For 2D data (CNNs)
bn2d = nn.BatchNorm2d(num_features=64)

# For 3D data (3D CNNs, video)
bn3d = nn.BatchNorm3d(num_features=32)

# Example: CNN with batch norm
model = nn.Sequential(
    nn.Conv2d(3, 64, 3, padding=1),
    nn.BatchNorm2d(64),  # Batch norm after conv
    nn.ReLU(),

    nn.Conv2d(64, 128, 3, padding=1),
    nn.BatchNorm2d(128),
    nn.ReLU(),
)

# Training mode (default)
model.train()
x = torch.randn(16, 3, 32, 32)
out = model(x)  # Uses batch statistics

# Evaluation mode
model.eval()
with torch.no_grad():
    out = model(x)  # Uses running statistics
```

Where to place batch norm:

Option 1: Conv → BN → Activation (recommended)

```python
nn.Conv2d(in_ch, out_ch, 3, padding=1),
nn.BatchNorm2d(out_ch),
nn.ReLU()
```

Option 2: Conv → Activation → BN (less common)

```python
nn.Conv2d(in_ch, out_ch, 3, padding=1),
nn.ReLU(),
nn.BatchNorm2d(out_ch)
```

Both work, but Option 1 is more standard.

[Critical: Set bias=False] When using batch norm after a linear/conv layer, set `bias=False`:

```python
# WRONG: Batch norm makes bias redundant
nn.Conv2d(3, 64, 3, padding=1, bias=True),  # Wasteful!
nn.BatchNorm2d(64),  # Has its own bias (beta)

# CORRECT: No need for conv bias
nn.Conv2d(3, 64, 3, padding=1, bias=False),  # No bias
```

```
nn.BatchNorm2d(64),  # Beta parameter serves as bias
```

Why? Batch norm subtracts the mean, so any constant bias gets canceled out. The β parameter in batch norm serves as the bias.

9.4.2 Layer Normalization Usage

```
# Layer norm normalizes over feature dimension
ln = nn.LayerNorm(normalized_shape=128)

# For sequences: (batch, seq_len, features)
x = torch.randn(32, 100, 128)  # 32 sequences, length 100, 128 features
out = ln(x)  # Normalizes over the 128 features for each position

# Example: Transformer-style layer
class TransformerBlock(nn.Module):
    def __init__(self, d_model):
        super().__init__()
        self.attention = nn.MultiheadAttention(d_model, num_heads=8)
        self.ln1 = nn.LayerNorm(d_model)
        self.ffn = nn.Sequential(
            nn.Linear(d_model, 4*d_model),
            nn.ReLU(),
            nn.Linear(4*d_model, d_model)
        )
        self.ln2 = nn.LayerNorm(d_model)

    def forward(self, x):
        # Attention with layer norm
        attn_out, _ = self.attention(x, x, x)
        x = self.ln1(x + attn_out)  # Residual + norm

        # FFN with layer norm
        ffn_out = self.ffn(x)
        x = self.ln2(x + ffn_out)  # Residual + norm

        return x
```

9.4.3 Other Normalization Variants

Instance Normalization (for style transfer):

```
# Normalizes each sample and channel independently
# Used in style transfer and GANs
in_norm = nn.InstanceNorm2d(num_features=64)

# Shape: (batch, channels, height, width)
# Computes mean/std for each (sample, channel) pair over (H, W)
```

Group Normalization (hybrid approach):

```
# Divides channels into groups, normalizes within each group
# Works well with small batch sizes
gn = nn.GroupNorm(num_groups=8, num_channels=64)

# Example: 64 channels → 8 groups of 8 channels each
# Normalizes over spatial dimensions and within each group
```

When to use each:

Normalization	Use Case
Batch Norm	CNNs, MLPs, large batches (≥ 8)
Layer Norm	RNNs, Transformers, any batch size
Instance Norm	Style transfer, GANs
Group Norm	Small batch sizes, object detection

Table 7: Choosing normalization type

9.4.4 Implementing Batch Norm from Scratch

```
class MyBatchNorm1d(nn.Module):
    """Batch normalization for understanding."""

    def __init__(self, num_features, eps=1e-5, momentum=0.1):
        super().__init__()

        # Learnable parameters
        self.gamma = nn.Parameter(torch.ones(num_features))
        self.beta = nn.Parameter(torch.zeros(num_features))

        # Running statistics (not trainable)
        self.register_buffer('running_mean', torch.zeros(num_features))
        self.register_buffer('running_var', torch.ones(num_features))

        self.eps = eps
        self.momentum = momentum

    def forward(self, x):
        # x shape: (batch, features)

        if self.training:
            # Training mode: use batch statistics
            batch_mean = x.mean(dim=0)
```

```
            batch_var = x.var(dim=0, unbiased=False)

            # Normalize
            x_norm = (x - batch_mean) / torch.sqrt(batch_var
    + self.eps)

            # Update running statistics (exponential moving
    average)
            with torch.no_grad():
                self.running_mean = (1 - self.momentum) *
    self.running_mean + \
                                    self.momentum * batch_mean
                self.running_var = (1 - self.momentum) * self
    .running_var + \
                                   self.momentum * batch_var
        else:
            # Eval mode: use running statistics
            x_norm = (x - self.running_mean) / \
                     torch.sqrt(self.running_var + self.eps)

        # Scale and shift
        out = self.gamma * x_norm + self.beta

        return out

# Test
bn = MyBatchNorm1d(10)
x = torch.randn(32, 10)

# Training mode
bn.train()
out_train = bn(x)

# Eval mode
bn.eval()
out_eval = bn(x)

print(f"Train output: {out_train.mean():.4f}, {out_train.std
    ():.4f}")
print(f"Eval output: {out_eval.mean():.4f}, {out_eval.std()
    :.4f}")
```

9.4.5 Common Mistakes and Debugging

Mistake 1: Forgetting to set eval mode

```
# WRONG: Model still in training mode during evaluation
model.train()  # Set at beginning of training
# ... training loop ...
# Evaluation (FORGOT model.eval()!)
with torch.no_grad():
    val_loss = evaluate(model, val_loader)  # Uses batch
    statistics!

# CORRECT:
model.train()
# ... training ...
model.eval()  # Switch to eval mode
with torch.no_grad():
    val_loss = evaluate(model, val_loader)  # Uses running
    statistics
```

Mistake 2: Small batch sizes with Batch Norm

```
# With batch_size=2, batch statistics are unreliable
# Symptoms: High variance in training, poor performance

# Solution 1: Increase batch size
train_loader = DataLoader(dataset, batch_size=32)  # Instead
    of 2

# Solution 2: Use Layer Norm or Group Norm instead
# Replace BatchNorm2d with GroupNorm
nn.GroupNorm(num_groups=8, num_channels=64)
```

Mistake 3: Not setting bias=False

```
# WASTEFUL: Both conv and batch norm have biases
layer = nn.Sequential(
    nn.Conv2d(64, 128, 3, bias=True),  # Has bias
    nn.BatchNorm2d(128)  # Also has bias (beta)
)

# EFFICIENT: Only batch norm has bias
layer = nn.Sequential(
    nn.Conv2d(64, 128, 3, bias=False),  # No bias
    nn.BatchNorm2d(128)  # Beta serves as bias
)
```

Debugging: Check if batch norm is working

```
def check_batch_norm_stats(model, dataloader):
    """Verify batch norm running statistics are being updated
    ."""

    model.train()

    # Save initial running mean
    bn_layer = None
    for module in model.modules():
        if isinstance(module, nn.BatchNorm2d):
            bn_layer = module
            break

```

```
    if bn_layer is None:
        print("No BatchNorm found!")
        return

    initial_mean = bn_layer.running_mean.clone()

    # Run one batch
    x, _ = next(iter(dataloader))
    _ = model(x)

    # Check if running mean changed
    mean_changed = not torch.allclose(initial_mean, bn_layer.
    running_mean)
    print(f"Running mean updated: {mean_changed}")

    # Check train vs eval difference
    model.eval()
    with torch.no_grad():
        out_eval = model(x)

    model.train()
    out_train = model(x)

    different = not torch.allclose(out_train, out_eval)
    print(f"Train vs eval outputs differ: {different}")

# Usage
# check_batch_norm_stats(model, train_loader)
```

9.5 Exercises

9.1: Batch Norm Impact - ★★

Goal: See batch norm's effect empirically.
Train two identical networks on CIFAR-10:

1. **Without batch norm:** Plain CNN
2. **With batch norm:** Same CNN + batch norm after each conv

Compare:

- Training speed (epochs to reach 70% accuracy)
- Learning rate stability (try lr=0.01 and lr=0.1)
- Final test accuracy

Expected observation: Batch norm allows higher learning rate and converges faster.
Starter code:

```
class CNNWithoutBN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, 3, padding=1),
            nn.ReLU(),
            nn.Conv2d(64, 128, 3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),
            # ... more layers
        )

class CNNWithBN(nn.Module):
    def __init__(self):
        super().__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, 3, padding=1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            # ... add batch norm after each conv
        )
```

9.2: Train vs Eval Mode - ★★

Goal: Understand the importance of eval mode.

1. Train a CNN with batch norm on MNIST
2. After training, evaluate on test set **without** calling `model.eval()`
3. Evaluate again **with** `model.eval()`
4. Compare test accuracies
5. Investigate: Try with different batch sizes during evaluation (1, 8, 64)

Questions:

- How much does forgetting eval() hurt performance?
- Does batch size during eval matter when in train mode?
- Why does this happen?

9.3: Batch Norm from Scratch - ★★★

Goal: Implement batch norm to understand it deeply.

1. Complete the `MyBatchNorm1d` implementation from above
2. Extend it to `MyBatchNorm2d` for CNNs
3. Test that it matches PyTorch's `nn.BatchNorm2d`:
   - Initialize both with same parameters
   - Feed same input
   - Verify outputs are close (use `torch.allclose`)
4. Train a small CNN using your custom batch norm
5. Verify training works correctly

Hint for BatchNorm2d:

```
# For input shape (batch, channels, height, width)
# Compute mean and var over dimensions (0, 2, 3)
# This gives per-channel statistics
batch_mean = x.mean(dim=(0, 2, 3), keepdim=True)
batch_var = x.var(dim=(0, 2, 3), keepdim=True, unbiased=
    False)
```

9.4: Layer Norm vs Batch Norm - ★★★

Goal: Compare normalization types for sequences.

1. Generate a sequence classification task (e.g., classify sine vs cosine)
2. Build an RNN with:
   - Batch normalization
   - Layer normalization
   - No normalization
3. Train all three and compare:
   - Training stability
   - Final accuracy
   - Effect of batch size (try 4, 16, 64)

Expected result: Layer norm should work better for sequences, especially with small batches.

9.5: Placement Experiments - ★★★★

Goal: Investigate where to place batch norm.
Test different placements in a residual block:

1. Conv → BN → ReLU
2. Conv → ReLU → BN
3. BN → ReLU → Conv (pre-activation)

For each:

- Train on CIFAR-10
- Measure convergence speed
- Check gradient flow (from Section 8)
- Compare final accuracy

Questions:

- Which placement works best?

- Does the answer change with network depth?
- How does it affect gradient flow?

9.6: Small Batch Challenge - ★★★★

Goal: Handle small batch sizes effectively.

1. Train a network with batch_size=2 (deliberately small)
2. Try:
   - Batch Norm (observe instability)
   - Layer Norm
   - Group Norm (8 groups)
   - No normalization
3. Compare training stability and final performance
4. Investigate: Why does batch norm fail? Look at batch statistics variance.

Analysis code:

```
# Check batch statistics variance
bn_layer = model.features[1]  # Assuming second layer is BN
means = []
vars = []

for batch in train_loader:
    x, _ = batch
    # Hook to capture batch statistics
    means.append(bn_layer.running_mean.clone())
    vars.append(bn_layer.running_var.clone())
    model(x)

# Plot variance of batch means over time
plt.plot([m.std().item() for m in means])
plt.title('Variance of Batch Means')
plt.show()
```

9.6 Key Takeaways

Why normalization helps:

- Reduces internal covariate shift (distribution changes between layers)
- Smooths optimization landscape (allows higher learning rates)
- Provides regularization (slight noise from batch statistics)
- Reduces sensitivity to initialization

Batch Normalization:

- Normalizes over batch dimension
- Requires large enough batches (≥ 8)
- Different behavior in train/eval modes (critical!)
- Best for CNNs and MLPs with large batches
- Can speed up training by 2-10×

Layer Normalization:

- Normalizes over feature dimension
- Works with any batch size (even 1)
- Same behavior in train/eval modes
- Best for RNNs, Transformers, sequences
- Standard in modern NLP models

Implementation best practices:

- Place after Conv/Linear, before activation (standard)
- Set `bias=False` in layer before batch norm
- Always call `model.eval()` during evaluation
- Use `torch.no_grad()` during inference
- For sequences: prefer Layer Norm
- For small batches: use Layer Norm or Group Norm

Common mistakes:

- Forgetting `model.eval()` (very common bug!)
- Using Batch Norm with tiny batches
- Not setting `bias=False` (wastes parameters)
- Wrong normalization type for task (Batch Norm for RNNs)
- Not understanding train vs eval mode difference

Choosing normalization:

- **Default for CNNs:** Batch Norm
- **Default for Transformers/RNNs:** Layer Norm
- **Small batches (<8):** Layer Norm or Group Norm
- **Style transfer:** Instance Norm
- **Object detection:** Group Norm (common in modern detectors)

When NOT to use normalization:

- Very shallow networks (<5 layers)
- Already training stably

- Certain GANs (can cause mode collapse)
- When exact reproducibility needed (batch stats introduce randomness)

10 Recurrent Neural Networks (RNNs)

10.1 Introduction: Processing Sequential Data

So far, we've seen networks that process fixed-size inputs. But many real-world problems involve **sequences**:

- Time series (stock prices, sensor data, climate data)
- Text (words, characters, sentences)
- Audio (speech, music)
- Video (frames over time)
- Biological sequences (DNA, proteins)

Challenges with sequences:

1. Variable length (sentences have different lengths)
2. Temporal dependencies (past affects future)
3. Need to maintain state (remember what happened before)

Why not use MLPs or CNNs?

MLPs:

- Fixed input size (can't handle variable-length sequences)
- No notion of order (position 1 vs position 100 treated same)
- Can't share parameters across time (learn same pattern at different positions)

CNNs:

- Can handle sequences with 1D convolutions
- Limited temporal context (receptive field)
- No true memory of long-term dependencies

RNNs solve these problems by maintaining a **hidden state** that gets updated at each time step.

10.2 Theory: How RNNs Work

10.2.1 The Basic RNN

An RNN processes a sequence one element at a time, maintaining a hidden state:

At each time step t:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

$$y_t = W_{hy}h_t + b_y$$

where:

- x_t: Input at time t
- h_t: Hidden state at time t (memory)
- y_t: Output at time t
- W_{hh}: Hidden-to-hidden weights (memory transformation)
- W_{xh}: Input-to-hidden weights
- W_{hy}: Hidden-to-output weights

Key idea: The hidden state h_t summarizes information from all previous time steps.

[The Hidden State] The hidden state h_t is the RNN's **memory**:

- h_0: Initial memory (usually zeros)
- $h_1 = f(x_1, h_0)$: Combines first input with initial state
- $h_2 = f(x_2, h_1)$: Combines second input with previous state
- $h_3 = f(x_3, h_2)$: And so on...

By the end: h_T contains information about the entire sequence $x_1, x_2, \ldots, x_T$.

Unrolled view:

```
x1 → [RNN] → h1 → [RNN] → h2 → [RNN] → h3 → ...
      ↓            ↓           ↓
      y1           y2          y3
```

All boxes share the **same weights** (parameter sharing across time).

10.2.2 Parameter Sharing

Why parameter sharing matters:

For a sequence of length 100:

- MLP: Would need 100 separate weight matrices (millions of parameters)
- RNN: Uses same weights at each step (thousands of parameters)

The network learns patterns that work at **any position** in the sequence.

10.2.3 Sequence Modeling Patterns

1. One-to-Many (sequence generation):

```
x → [RNN] → [RNN] → [RNN] → ...
             ↓       ↓       ↓
             y1      y2      y3
Example: Image captioning (image → sentence)
```

2. Many-to-One (sequence classification):

```
x1 → [RNN] → x2 → [RNN] → x3 → [RNN]
                                 ↓
                                 y
Example: Sentiment analysis (sentence → positive/negative)
```

3. Many-to-Many (same length):

```
x1 → [RNN] → x2 → [RNN] → x3 → [RNN]
     ↓           ↓           ↓
     y1          y2          y3
Example: Part-of-speech tagging (word → tag)
```

4. Many-to-Many (different length, encoder-decoder):

```
Encoder:  x1 → [RNN] → x2 → [RNN] → h
Decoder:  h → [RNN] → [RNN] → [RNN]
               ↓       ↓       ↓
               y1      y2      y3
Example: Machine translation (English → French)
```

10.2.4 The Vanishing Gradient Problem

Problem: Vanilla RNNs struggle with long sequences.

When backpropagating through time, gradients are multiplied by W_{hh} at each step:

$$\frac{\partial h_t}{\partial h_0} = \prod_{i=1}^{t} W_{hh} \cdot \text{diag}(\tanh'(h_i))$$

If $|W_{hh}| < 1$: Gradients vanish exponentially (network can't learn long-term dependencies)

If $|W_{hh}| > 1$: Gradients explode (training becomes unstable)

In practice: Vanilla RNNs can only remember 10-20 steps back.

Solution: LSTM and GRU architectures (next sections).

10.3 Theory: Long Short-Term Memory (LSTM)

LSTMs solve the vanishing gradient problem through a clever gating mechanism.

10.3.1 LSTM Architecture

LSTM has two states:

- h_t: Hidden state (short-term memory)
- c_t: Cell state (long-term memory)

And three gates:

- f_t: Forget gate (what to remove from memory)
- i_t: Input gate (what new information to add)
- o_t: Output gate (what to output)

LSTM Equations:

1. Forget gate (what to forget):

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$$

2. Input gate (what to add):

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$

$$\tilde{c}_t = \tanh(W_c[h_{t-1}, x_t] + b_c) \quad \text{(candidate values)}$$

3. Update cell state:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

4. Output gate (what to output):

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$

$$h_t = o_t \odot \tanh(c_t)$$

where $\odot$ is element-wise multiplication, σ is sigmoid, $[a, b]$ is concatenation.

[LSTM Intuition] **Cell state (c_t):** The "memory highway"

- Runs through the entire sequence
- Modified only by additions and element-wise multiplications
- Gradients flow easily (no repeated matrix multiplications!)

Forget gate (f_t): "Should I forget the past?"

- Values near 0: Forget most of c_{t-1}
- Values near 1: Keep most of c_{t-1}

Input gate (i_t): "Should I add this new information?"

- Controls how much of $\tilde{c}_t$ to add to memory

Output gate (o_t): "How much of memory should I expose?"

- Controls how much of c_t becomes h_t

Why LSTM works:

The cell state c_t provides a path where gradients can flow without repeated matrix multiplications (just element-wise operations). This solves vanishing gradients!

10.4 Theory: Gated Recurrent Unit (GRU)

GRU is a simplified version of LSTM with fewer parameters.

10.4.1 GRU Architecture

GRU combines forget and input gates into a single **update gate** and merges cell state with hidden state.

GRU Equations:

1. Update gate (how much to update):

$$z_t = \sigma(W_z[h_{t-1}, x_t] + b_z)$$

2. Reset gate (how much past to forget):

$$r_t = \sigma(W_r[h_{t-1}, x_t] + b_r)$$

3. Candidate hidden state:

$$\tilde{h}_t = \tanh(W_h[r_t \odot h_{t-1}, x_t] + b_h)$$

4. Update hidden state:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

Intuition:

- $z_t \approx 0$: Keep old state h_{t-1} (don't update)
- $z_t \approx 1$: Use new candidate $\tilde{h}_t$ (full update)
- $z_t \approx 0.5$: Blend old and new

LSTM vs GRU:

Aspect	LSTM	GRU
Gates	3 (forget, input, output)	2 (update, reset)
States	2 (h_t, c_t)	1 (h_t)
Parameters	More	Fewer (faster)
Performance	Slightly better	Nearly as good
Training speed	Slower	Faster
Memory	More	Less

Table 8: LSTM vs GRU comparison

When to use each:

- **LSTM:** Default choice, especially for complex tasks
- **GRU:** When speed matters or dataset is smaller
- **Rule of thumb:** Try both, see which works better

10.5 Implementation: RNNs in PyTorch

10.5.1 Vanilla RNN

```python
import torch
import torch.nn as nn

# Simple RNN cell
rnn_cell = nn.RNNCell(input_size=10, hidden_size=20)

# Input: (batch, input_size)
# Hidden: (batch, hidden_size)
x = torch.randn(32, 10)  # Batch of 32
h = torch.zeros(32, 20)  # Initial hidden state

# One step
h_next = rnn_cell(x, h)
print(h_next.shape)  # torch.Size([32, 20])

# Full RNN layer (processes entire sequence)
rnn = nn.RNN(input_size=10, hidden_size=20, num_layers=1,
    batch_first=True)

# Input: (batch, seq_len, input_size)
x = torch.randn(32, 100, 10)  # 32 sequences, length 100
h0 = torch.zeros(1, 32, 20)   # (num_layers, batch,
    hidden_size)

# Forward pass
output, h_n = rnn(x, h0)
# output: (32, 100, 20) - outputs at each time step
# h_n: (1, 32, 20) - final hidden state

print(output.shape, h_n.shape)
```

[batch_first Parameter] PyTorch RNNs default to shape (seq_len, batch, features), but this is confusing!

Always use `batch_first=True` for shape (batch, seq_len, features):

```python
rnn = nn.RNN(..., batch_first=True)  # Recommended
# Input: (batch, seq_len, features)
# Output: (batch, seq_len, hidden_size)
```

10.5.2 LSTM

```python
# LSTM cell
lstm_cell = nn.LSTMCell(input_size=10, hidden_size=20)

x = torch.randn(32, 10)
h = torch.zeros(32, 20)
c = torch.zeros(32, 20)  # Cell state

# One step
h_next, c_next = lstm_cell(x, (h, c))  # Note: tuple of (h, c)

# Full LSTM layer
lstm = nn.LSTM(input_size=10, hidden_size=20, num_layers=2,
               batch_first=True, dropout=0.2)

x = torch.randn(32, 100, 10)
h0 = torch.zeros(2, 32, 20)  # (num_layers, batch, hidden)
c0 = torch.zeros(2, 32, 20)  # (num_layers, batch, hidden)

# Forward pass
output, (h_n, c_n) = lstm(x, (h0, c0))
# output: (32, 100, 20) - outputs at each time step
# h_n: (2, 32, 20) - final hidden state for each layer
# c_n: (2, 32, 20) - final cell state for each layer

print(output.shape, h_n.shape, c_n.shape)
```

[LSTM Hidden State is a Tuple!] LSTM returns `(h, c)` as a tuple, not just `h`:

```python
# WRONG:
output, hidden = lstm(x)
h_n = hidden  # This is a tuple, not a tensor!

# CORRECT:
output, (h_n, c_n) = lstm(x, (h0, c0))
# or
output, hidden = lstm(x, (h0, c0))
h_n, c_n = hidden
```

10.5.3 GRU

```python
# GRU (similar to LSTM, but only one hidden state)
gru = nn.GRU(input_size=10, hidden_size=20, num_layers=2,
             batch_first=True, dropout=0.2)

x = torch.randn(32, 100, 10)
h0 = torch.zeros(2, 32, 20)

# Forward pass
output, h_n = gru(x, h0)  # Only h, no cell state
# output: (32, 100, 20)
# h_n: (2, 32, 20)

print(output.shape, h_n.shape)
```

10.5.4 Building a Sequence Classifier

```python
class SequenceClassifier(nn.Module):
    """Many-to-one: sequence → single label."""

    def __init__(self, input_size, hidden_size, num_classes, num_layers=2):
        super().__init__()

        self.lstm = nn.LSTM(
            input_size=input_size,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True,
            dropout=0.2 if num_layers > 1 else 0
        )

        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        # x: (batch, seq_len, input_size)

        # LSTM forward
        output, (h_n, c_n) = self.lstm(x)
        # output: (batch, seq_len, hidden_size)
        # h_n: (num_layers, batch, hidden_size)

        # Use last time step's output
        last_output = output[:, -1, :]  # (batch, hidden_size)

        # Or use final hidden state from last layer
        # last_hidden = h_n[-1]  # (batch, hidden_size)

        # Classify
        logits = self.fc(last_output)
        return logits

# Test
model = SequenceClassifier(input_size=10, hidden_size=64, num_classes=3)
x = torch.randn(32, 100, 10)  # 32 sequences, length 100
output = model(x)
print(output.shape)  # torch.Size([32, 3])
```

10.5.5 Sequence-to-Sequence (Many-to-Many)

```
class SequenceLabeler(nn.Module):
    """Many-to-many: each input      each output (same length)
    ."""

    def __init__(self, input_size, hidden_size, num_classes):
        super().__init__()

        self.lstm = nn.LSTM(
            input_size=input_size,
            hidden_size=hidden_size,
            num_layers=2,
            batch_first=True,
            dropout=0.2
        )

        # Apply classifier at each time step
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        # x: (batch, seq_len, input_size)

        output, _ = self.lstm(x)
        # output: (batch, seq_len, hidden_size)

        # Apply classifier to all time steps
        logits = self.fc(output)
        # logits: (batch, seq_len, num_classes)

        return logits

# Test
model = SequenceLabeler(input_size=10, hidden_size=64,
    num_classes=5)
x = torch.randn(32, 100, 10)
output = model(x)
print(output.shape)  # torch.Size([32, 100, 5])
```

10.5.6 Bidirectional RNNs

Process sequence in both directions (forward and backward):

```
class BidirectionalRNN(nn.Module):
    """Bidirectional LSTM for sequence classification."""

    def __init__(self, input_size, hidden_size, num_classes):
        super().__init__()

        self.lstm = nn.LSTM(
            input_size=input_size,
            hidden_size=hidden_size,
            num_layers=2,
            batch_first=True,
            dropout=0.2,
            bidirectional=True  # Process both directions
        )

        # Hidden size is doubled (forward + backward)
        self.fc = nn.Linear(hidden_size * 2, num_classes)

    def forward(self, x):
```

```
        # x: (batch, seq_len, input_size)

        output, _ = self.lstm(x)
        # output: (batch, seq_len, hidden_size * 2)

        # Use last time step (or all for seq-to-seq)
        last_output = output[:, -1, :]

        logits = self.fc(last_output)
        return logits

# Test
model = BidirectionalRNN(input_size=10, hidden_size=64,
    num_classes=3)
x = torch.randn(32, 100, 10)
output = model(x)
print(output.shape)  # torch.Size([32, 3])
```

When to use bidirectional:

- Text classification (can see future context)
- Part-of-speech tagging
- Named entity recognition
- Any task where you have the full sequence before prediction

Don't use bidirectional for:

- Time series prediction (can't see future!)
- Language generation (need causal/autoregressive)
- Online/streaming applications

10.5.7 Handling Variable-Length Sequences

```python
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence

class VariableLengthRNN(nn.Module):
    """Efficiently handle sequences of different lengths."""

    def __init__(self, input_size, hidden_size, num_classes):
        super().__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x, lengths):
        """
        Args:
            x: Padded sequences (batch, max_len, input_size)
            lengths: Actual lengths of each sequence
        """
        # Pack sequences (skip padding in computation)
        packed = pack_padded_sequence(x, lengths, batch_first=True,
                                      enforce_sorted=False)

        # LSTM forward on packed sequence
        packed_output, (h_n, c_n) = self.lstm(packed)

        # Unpack
        output, _ = pad_packed_sequence(packed_output, batch_first=True)

        # Use final hidden state
        logits = self.fc(h_n[-1])

        return logits

# Example usage
# Sequences of different lengths: [50, 30, 40]
lengths = torch.tensor([50, 30, 40])
max_len = lengths.max()

# Pad to max length
x = torch.randn(3, max_len, 10)  # Some values are padding

model = VariableLengthRNN(10, 64, 3)
output = model(x, lengths)
print(output.shape)  # torch.Size([3, 3])
```

Why pack sequences?

- Skip computation on padding (faster)
- More accurate (padding doesn't affect hidden state)
- More memory efficient

10.5.8 Gradient Clipping (Essential for RNNs)

RNNs are prone to exploding gradients. Gradient clipping is **essential**.

```
# Method 1: Clip by norm (recommended)
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=5.0)

# Method 2: Clip by value
torch.nn.utils.clip_grad_value_(model.parameters(), clip_value=0.5)

# In training loop
for epoch in range(num_epochs):
    for batch in dataloader:
        optimizer.zero_grad()

        output = model(batch)
        loss = criterion(output, target)
        loss.backward()

        # Clip gradients AFTER backward, BEFORE step
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=5.0)

        optimizer.step()
```

[Always Use Gradient Clipping for RNNs] Without gradient clipping:

- Loss may suddenly spike to NaN
- Training becomes unstable
- Network may never converge

Typical values:

- `max_norm=1.0`: Conservative (very stable)
- `max_norm=5.0`: Standard (good default)
- `max_norm=10.0`: Aggressive (faster but less stable)

10.5.9 Training Tips for RNNs

1. Learning rate

```
# RNNs typically need lower learning rates than CNNs
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)  # Good start

# Use learning rate scheduling
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='min', factor=0.5, patience=5
)
```

2. Weight initialization

```
def init_lstm_weights(lstm):
    """Initialize LSTM weights for better convergence."""
    for name, param in lstm.named_parameters():
        if 'weight_ih' in name:
            # Input-hidden weights: Xavier
            nn.init.xavier_uniform_(param)
```

```
        elif 'weight_hh' in name:
            # Hidden-hidden weights: Orthogonal (better for RNNs)
            nn.init.orthogonal_(param)
        elif 'bias' in name:
            # Biases: zeros, except forget gate bias = 1
            nn.init.zeros_(param)
            # Set forget gate bias to 1 (helps gradient flow)
            n = param.size(0)
            param.data[n//4:n//2].fill_(1.0)

# Apply to model
for name, module in model.named_modules():
    if isinstance(module, nn.LSTM):
        init_lstm_weights(module)
```

3. Dropout placement

```
# Dropout between layers (built-in)
lstm = nn.LSTM(input_size, hidden_size, num_layers=3, dropout=0.2)
# This applies dropout between layers, not within recurrent connections

# For dropout on inputs/outputs:
class RNNWithDropout(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super().__init__()
        self.input_dropout = nn.Dropout(0.2)
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers=2)
        self.output_dropout = nn.Dropout(0.5)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        x = self.input_dropout(x)
        output, _ = self.lstm(x)
        output = self.output_dropout(output[:, -1, :])
        return self.fc(output)
```

4. Batch size considerations

RNNs benefit from larger batch sizes:

- Batch size 32-128 typically good
- Too small (<16): Noisy gradients, unstable training
- Too large (>256): May hurt generalization

10.5.10 Common RNN Architectures

Stacked RNNs (Deep RNNs):

```python
class DeepLSTM(nn.Module):
    """Multiple LSTM layers stacked."""

    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super().__init__()

        # num_layers > 1 creates stacked LSTM
        self.lstm = nn.LSTM(
            input_size=input_size,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True,
            dropout=0.3 if num_layers > 1 else 0  # Dropout between layers
        )

        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        output, (h_n, c_n) = self.lstm(x)
        # h_n[-1]: hidden state from the last layer
        return self.fc(h_n[-1])

# Typical depths: 2-4 layers
# Deeper than 4 often doesn't help much
model = DeepLSTM(input_size=10, hidden_size=128, num_layers=3,
                 num_classes=10)
```

Encoder-Decoder (Seq2Seq):

```python
class Seq2Seq(nn.Module):
    """Encoder-decoder for sequence-to-sequence tasks."""

    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()

        # Encoder: processes input sequence
        self.encoder = nn.LSTM(input_size, hidden_size, batch_first=True)

        # Decoder: generates output sequence
        self.decoder = nn.LSTM(output_size, hidden_size, batch_first=True)

        # Output projection
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, src, tgt, teacher_forcing_ratio=0.5):
        """
        Args:
            src: Source sequence (batch, src_len, input_size)
            tgt: Target sequence (batch, tgt_len, output_size)
            teacher_forcing_ratio: Probability of using true target vs prediction
        """
```

```
        batch_size = src.size(0)
        tgt_len = tgt.size(1)
        output_size = tgt.size(2)

        # Encode source
        _, (h_n, c_n) = self.encoder(src)

        # Initialize decoder hidden state with encoder final state
        decoder_hidden = (h_n, c_n)

        # First decoder input: start token (zeros)
        decoder_input = torch.zeros(batch_size, 1, output_size).to(src.device)

        outputs = []

        # Generate sequence step by step
        for t in range(tgt_len):
            # Decode one step
            decoder_output, decoder_hidden = self.decoder(
                decoder_input, decoder_hidden
            )

            # Project to output space
            prediction = self.fc(decoder_output)
            outputs.append(prediction)

            # Teacher forcing: use true target or prediction?
            use_teacher_forcing = torch.rand(1).item() < teacher_forcing_ratio

            if use_teacher_forcing:
                decoder_input = tgt[:, t:t+1, :]  # Use true target
            else:
                decoder_input = prediction  # Use prediction

        # Stack all outputs
        outputs = torch.cat(outputs, dim=1)
        return outputs

# Test
model = Seq2Seq(input_size=10, hidden_size=64, output_size=10)
src = torch.randn(32, 20, 10)  # 32 sequences, length 20
tgt = torch.randn(32, 15, 10)  # Target length 15
output = model(src, tgt)
print(output.shape)  # torch.Size([32, 15, 10])
```

10.5.11 Time Series Forecasting Example

```
class TimeSeriesForecaster(nn.Module):
    """Predict future values from past observations."""

    def __init__(self, input_size=1, hidden_size=64, num_layers=2,
                 forecast_horizon=10):
        super().__init__()

        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.forecast_horizon = forecast_horizon

        # Encoder LSTM
        self.lstm = nn.LSTM(
            input_size=input_size,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True,
            dropout=0.2 if num_layers > 1 else 0
        )

        # Decoder: predict next step
        self.fc = nn.Linear(hidden_size, input_size)

    def forward(self, x, future_steps=0):
        """
        Args:
            x: Past observations (batch, seq_len, input_size)
            future_steps: How many steps to predict into future

        Returns:
            predictions: (batch, seq_len + future_steps, input_size)
        """
        batch_size = x.size(0)

        # Process historical data
        output, (h, c) = self.lstm(x)

        # Predict on historical data
        predictions = self.fc(output)

        # If we need to predict future steps
        if future_steps > 0:
            future_preds = []

            # Use last prediction as next input
            last_pred = predictions[:, -1:, :]

            for _ in range(future_steps):
                # Predict next step
                output, (h, c) = self.lstm(last_pred, (h, c))
                pred = self.fc(output)
                future_preds.append(pred)

                # Use prediction as next input (autoregressive)
                last_pred = pred

```

```
            # Concatenate all predictions
            future_preds = torch.cat(future_preds, dim=1)
            predictions = torch.cat([predictions,
    future_preds], dim=1)

        return predictions

# Example usage
model = TimeSeriesForecaster(input_size=1, hidden_size=64)

# Historical data: 100 time steps
x = torch.randn(32, 100, 1)

# Predict on historical + 20 future steps
predictions = model(x, future_steps=20)
print(predictions.shape)  # torch.Size([32, 120, 1])
```

10.6 Exercises

10.1: Simple Sequence Classification - ★★

Goal: Build your first RNN.

1. Generate synthetic sequences: sine waves (class 0) and cosine waves (class 1)
2. Each sequence: 50 time steps
3. Build an LSTM classifier
4. Train for 20 epochs
5. Achieve >95% accuracy

Starter code:

```
import numpy as np

def generate_sine_data(n_samples=1000, seq_len=50):
    """Generate sine wave sequences."""
    X = []
    y = []
    for _ in range(n_samples):
        # Random frequency and phase
        freq = np.random.uniform(0.5, 2.0)
        phase = np.random.uniform(0, 2*np.pi)
        t = np.linspace(0, 4*np.pi, seq_len)

        if np.random.rand() < 0.5:
            # Sine wave
            seq = np.sin(freq * t + phase)
            label = 0
        else:
            # Cosine wave
            seq = np.cos(freq * t + phase)
            label = 1

        X.append(seq)
        y.append(label)

    return torch.FloatTensor(X).unsqueeze(-1), torch.
    LongTensor(y)

# Your LSTM classifier here
```

10.2: LSTM vs GRU Comparison - ★★★

Goal: Compare LSTM and GRU empirically.

1. Generate a moderately complex sequence task
2. Train identical architectures with:
   - LSTM (2 layers, 128 hidden)
   - GRU (2 layers, 128 hidden)
   - Vanilla RNN (2 layers, 128 hidden)
3. Compare:
   - Training speed (time per epoch)
   - Convergence speed (epochs to 90% accuracy)
   - Final accuracy
   - Number of parameters

Expected results:

- LSTM and GRU should perform similarly
- GRU should be faster (fewer parameters)
- Vanilla RNN should struggle (vanishing gradients)

10.3: Sequence-to-Sequence Labeling - ★★★

Goal: Many-to-many prediction.

1. Task: Given a noisy sine wave, denoise it at each time step
2. Input: sine + Gaussian noise
3. Target: clean sine wave
4. Build a bidirectional LSTM
5. Output prediction at each time step
6. Visualize: noisy input vs clean output

Evaluation:

- Mean squared error on test set
- Visual inspection of denoised signals

10.4: Variable-Length Sequences - ★★★

Goal: Handle sequences of different lengths efficiently.

1. Generate sequences with random lengths (20-100)
2. Implement proper padding and packing:
   - Pad sequences to same length
   - Use `pack_padded_sequence`
   - Use `pad_packed_sequence`
3. Train two models:
   - Without packing (processes padding)
   - With packing (skips padding)
4. Compare:
   - Training time
   - Accuracy

- Verify packing gives same results

10.5: Time Series Forecasting - ★★★★

Goal: Predict future values.

1. Use real time series data (or generate complex synthetic data)
2. Sliding window approach:
   - Input: past 50 steps
   - Target: next 10 steps
3. Build forecasting model with LSTM
4. Implement autoregressive prediction (use predictions as inputs)
5. Evaluate:
   - One-step-ahead prediction
   - Multi-step-ahead prediction
   - Compare with baseline (simple moving average)
6. Visualize predictions vs ground truth

Metrics:

```python
def calculate_metrics(y_true, y_pred):
    """Calculate forecasting metrics."""
    mse = torch.mean((y_true - y_pred) ** 2)
    mae = torch.mean(torch.abs(y_true - y_pred))

    # MAPE (Mean Absolute Percentage Error)
    mape = torch.mean(torch.abs((y_true - y_pred) / (y_true + 1e-8))) * 100

    return {
        'MSE': mse.item(),
        'MAE': mae.item(),
        'MAPE': mape.item()
    }
```

10.6: Gradient Clipping Investigation - ★★★★

Goal: Understand why gradient clipping is essential.

1. Train an RNN on a long sequence task (100+ steps)
2. Train multiple times with different settings:
   - No gradient clipping
   - `max_norm=1.0`
   - `max_norm=5.0`
   - `max_norm=10.0`
3. Monitor and plot:
   - Gradient norms over time
   - Loss curves
   - Percentage of batches where gradients are clipped
4. Observe:
   - Without clipping: loss spikes, NaN values
   - With clipping: stable training

Visualization code:

```python
def track_gradient_norms(model):
    """Track gradient norms during training."""
    total_norm = 0
    for p in model.parameters():
        if p.grad is not None:
            param_norm = p.grad.data.norm(2)
            total_norm += param_norm.item() ** 2
    total_norm = total_norm ** 0.5
    return total_norm

# In training loop
gradient_norms = []
for epoch in range(num_epochs):
    for batch in dataloader:
        optimizer.zero_grad()
        loss = ...
        loss.backward()

        # Track before clipping
        norm_before = track_gradient_norms(model)
        gradient_norms.append(norm_before)

        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=5.0)
        optimizer.step()

# Plot
plt.plot(gradient_norms)
plt.axhline(y=5.0, color='r', linestyle='--', label='Clip threshold')
plt.xlabel('Step')
plt.ylabel('Gradient Norm')
plt.legend()
plt.show()
```

10.7 Key Takeaways

When to use RNNs:

- Sequential data with temporal dependencies
- Variable-length sequences
- Time series forecasting
- Text processing (though Transformers now dominate)
- When order matters

Architecture choices:

- **Vanilla RNN:** Almost never use (vanishing gradients)
- **LSTM:** Default choice, especially for complex tasks
- **GRU:** When speed matters or dataset is smaller
- **Bidirectional:** When you have full sequence (not streaming)
- **Stacked (2-4 layers):** For complex patterns

Essential training practices:

- **Always use gradient clipping** (max_norm=5.0 is good default)
- Initialize forget gate bias to 1 (helps gradient flow)
- Use orthogonal initialization for hidden-to-hidden weights
- Lower learning rates than CNNs (0.001 typical)
- Larger batch sizes (32-128)
- Pack sequences when lengths vary (efficiency)

Common patterns:

- **Many-to-one:** Classification (use last hidden state)
- **Many-to-many (same length):** Sequence labeling (output at each step)
- **Many-to-many (different length):** Seq2seq with encoder-decoder
- **One-to-many:** Generation (feed output back as input)

Why LSTMs work better than vanilla RNNs:

- Cell state provides gradient highway (no repeated matrix multiplications)
- Gates control information flow (selective memory)
- Can learn dependencies over 100+ steps
- Forget gate prevents exploding activations

Limitations of RNNs:

- Sequential processing (slow, can't parallelize)
- Still struggle with very long sequences (>1000 steps)
- Transformers now preferred for many NLP tasks
- Require careful tuning (gradient clipping, learning rate)

Practical tips:

- Start with 2-layer LSTM, hidden size = 128 or 256
- Use `batch_first=True` always
- Remember LSTM returns `(h, c)` tuple
- Dropout between layers (0.2-0.3), higher on outputs (0.5)

- Monitor gradient norms (should be <10)
- Use early stopping (RNNs prone to overfitting)

Debugging checklist:

- Loss explodes? → Add/increase gradient clipping
- Loss stuck? → Check learning rate, try lower
- Poor performance? → Try LSTM instead of GRU, add layers
- Slow training? → Use packing for variable lengths
- Different train/test? → Check dropout mode

Modern alternatives:

- **Transformers:** Better for most NLP tasks (parallelizable)
- **Temporal CNNs:** Good for some time series (efficient)
- **State space models:** Emerging alternative (S4, Mamba)

However, RNNs still useful for:

- Online/streaming applications (Transformers need full context)
- Very long sequences where Transformers too expensive
- Small datasets (fewer parameters than Transformers)
- Scientific time series (physics-informed RNNs)

11 Attention & Transformers

11.1 Introduction: Beyond RNNs

RNNs revolutionized sequence modeling, but they have fundamental limitations:

Problems with RNNs:

1. **Sequential processing:** Can't parallelize (slow!)
2. **Long-range dependencies:** Even LSTM struggles beyond 100 steps
3. **Information bottleneck:** All info must pass through hidden state
4. **Vanishing gradients:** Still an issue for very long sequences

The Attention mechanism (2014) and **Transformers** (2017) solved these problems.

Key innovation: Instead of compressing everything into a fixed-size hidden state, let the model **attend to** any part of the input directly.

Impact:

- GPT, BERT, ChatGPT: All based on Transformers
- Transformers now dominate NLP, vision (ViT), multimodal (CLIP), and more
- Enabled training on massive datasets (parallelization)
- State-of-the-art on virtually all sequence tasks

11.2 Theory: The Attention Mechanism

11.2.1 Intuition: What is Attention?

Imagine reading a document and answering a question. You don't re-read the entire document—you **focus** on relevant parts.

Example:

- Document: "The cat sat on the mat. The dog played in the garden."
- Query: "Where is the cat?"
- You attend to: "The cat sat on the mat" (relevant)
- You ignore: "The dog played in the garden" (irrelevant)

Attention does this automatically:

- Given a query, compute relevance scores for all inputs
- Higher scores = more attention
- Output is weighted sum of values, weighted by attention scores

11.2.2 Mathematical Formulation

Attention has three components:

- **Query (Q):** What we're looking for
- **Keys (K):** What each position represents
- **Values (V):** The actual content at each position

Step 1: Compute attention scores

$$\text{scores} = QK^T$$

Each element (i, j) is how relevant key j is to query i.

Step 2: Normalize with softmax

$$\text{attention\_weights} = \text{softmax}(\text{scores})$$

Converts scores to probabilities (sum to 1).

Step 3: Weighted sum of values

$$\text{output} = \text{attention\_weights} \cdot V$$

Complete formula (scaled dot-product attention):

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where d_k is the dimension of keys (scaling prevents large dot products).

[Why Scaling by $\sqrt{d_k}$?] Without scaling, dot products can become very large when d_k is large:

- Large dot products $\rightarrow$ softmax saturates $\rightarrow$ tiny gradients
- Scaling by $\sqrt{d_k}$ keeps values in reasonable range
- Ensures stable gradients

Example: If $d_k = 512$, dot products could be ~ 512 without scaling. After scaling: $\sim \sqrt{512} \approx 22.6$.

11.2.3 Self-Attention vs Cross-Attention

Self-Attention: Attention within same sequence

Query, Key, Value all come from the same source.

Use case: Understanding relationships between words in a sentence.

Example:

- Sentence: "The animal didn't cross the street because it was too tired"
- "it" attends to "animal" (not "street")

Cross-Attention: Attention between two sequences

Query from one sequence, Keys and Values from another.

Use case: Machine translation, encoder-decoder models.

Example:

- English (Keys/Values): "The cat is black"
- French decoder (Query): "Le chat est..."
- Decoder attends to relevant English words

11.2.4 Multi-Head Attention

Instead of one attention mechanism, use multiple in parallel:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

where each head is:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

Why multiple heads?

- Different heads can focus on different aspects
- Head 1: Syntax (subject-verb agreement)
- Head 2: Semantics (meaning relationships)
- Head 3: Position (nearby words)
- Richer representations than single attention

Typical values:

- $h = 8$ or $h = 16$ heads
- $d_{model} = 512$ (total dimension)
- $d_k = d_v = d_{model}/h = 64$ per head

11.2.5 Positional Encoding

Problem: Attention has no notion of order!

The sentence "cat chased dog" and "dog chased cat" look identical to attention (same words, same attention scores).

Solution: Add positional information to input embeddings.

Sinusoidal positional encoding:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

where pos is position, i is dimension index.

Properties:

- Different frequency for each dimension
- Can extrapolate to longer sequences than seen during training
- Model can learn to attend by relative position

Alternative: Learned positional embeddings (fixed maximum length).

11.3 Theory: The Transformer Architecture

11.3.1 Complete Architecture

Transformer consists of:

- **Encoder:** Processes input sequence
- **Decoder:** Generates output sequence

Encoder Block (repeated N times):

1. Multi-head self-attention
2. Add & Layer Norm (residual connection)
3. Feed-forward network (two linear layers with ReLU)
4. Add & Layer Norm (residual connection)

Decoder Block (repeated N times):

1. Masked multi-head self-attention (can't see future)
2. Add & Layer Norm
3. Multi-head cross-attention (attend to encoder output)
4. Add & Layer Norm
5. Feed-forward network
6. Add & Layer Norm

Diagram:

```
Input → Embedding → + Positional Encoding
                          ↓
                  [Encoder Block] x N
                          ↓
                  Encoder Output
                          ↓
Target → Embedding → + Positional Encoding
                          ↓
                  [Decoder Block] x N
                          ↓
                  Linear → Softmax → Output
```

11.3.2 Why Transformers Work So Well

1. Parallelization

RNN: Must process step-by-step (slow)

Transformer: All positions processed in parallel (fast!)

Training on long sequences: 10-100× faster than RNNs.

2. Long-range dependencies

RNN: Information must flow through many steps (degrades)

Transformer: Direct connection between any two positions (one attention operation)

3. Scalability

Transformers scale beautifully with:

- More data (billions of tokens)

- More parameters (billions to trillions)
- More compute (hundreds of GPUs)

4. Transfer learning

Pre-train once on massive data → Fine-tune for specific tasks.

Examples: BERT, GPT, T5.

5. Inductive biases

Minimal assumptions about data structure:

- CNNs assume local structure (good for images)
- RNNs assume sequential processing (good for sequences)
- Transformers learn structure from data (flexible!)

11.4 Implementation: Building Attention from Scratch

11.4.1 Scaled Dot-Product Attention

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

def scaled_dot_product_attention(query, key, value, mask=None):
    """
    Scaled dot-product attention.

    Args:
        query: (batch, seq_len_q, d_k)
        key: (batch, seq_len_k, d_k)
        value: (batch, seq_len_k, d_v)
        mask: Optional mask (batch, seq_len_q, seq_len_k)

    Returns:
        output: (batch, seq_len_q, d_v)
        attention_weights: (batch, seq_len_q, seq_len_k)
    """
    d_k = query.size(-1)

    # Compute attention scores: Q @ K^T
    scores = torch.matmul(query, key.transpose(-2, -1))  # (batch, seq_len_q, seq_len_k)

    # Scale by sqrt(d_k)
    scores = scores / math.sqrt(d_k)

    # Apply mask if provided (set masked positions to -inf)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)

    # Softmax to get attention weights
    attention_weights = F.softmax(scores, dim=-1)  # (batch, seq_len_q, seq_len_k)

    # Weighted sum of values
    output = torch.matmul(attention_weights, value)  # (batch, seq_len_q, d_v)

    return output, attention_weights

# Test
batch_size = 2
seq_len = 10
d_k = 64

Q = torch.randn(batch_size, seq_len, d_k)
K = torch.randn(batch_size, seq_len, d_k)
V = torch.randn(batch_size, seq_len, d_k)

output, weights = scaled_dot_product_attention(Q, K, V)
print(f"Output shape: {output.shape}")  # (2, 10, 64)
print(f"Attention weights shape: {weights.shape}")  # (2, 10, 10)
print(f"Weights sum to 1: {weights.sum(dim=-1)}")  # All ones
```

11.4.2 Multi-Head Attention

```python
class MultiHeadAttention(nn.Module):
    """Multi-head attention mechanism."""

    def __init__(self, d_model, num_heads, dropout=0.1):
        """
        Args:
            d_model: Model dimension (e.g., 512)
            num_heads: Number of attention heads (e.g., 8)
            dropout: Dropout probability
        """
        super().__init__()

        assert d_model % num_heads == 0, "d_model must be divisible by num_heads"

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads  # Dimension per head

        # Linear projections for Q, K, V
        self.W_q = nn.Linear(d_model, d_model)
        self.W_k = nn.Linear(d_model, d_model)
        self.W_v = nn.Linear(d_model, d_model)

        # Output projection
        self.W_o = nn.Linear(d_model, d_model)

        self.dropout = nn.Dropout(dropout)

    def split_heads(self, x):
        """
        Split last dimension into (num_heads, d_k).

        Input: (batch, seq_len, d_model)
        Output: (batch, num_heads, seq_len, d_k)
        """
        batch_size, seq_len, d_model = x.size()

        # Reshape to (batch, seq_len, num_heads, d_k)
        x = x.view(batch_size, seq_len, self.num_heads, self.d_k)

        # Transpose to (batch, num_heads, seq_len, d_k)
        return x.transpose(1, 2)

    def combine_heads(self, x):
        """
        Combine heads back.

        Input: (batch, num_heads, seq_len, d_k)
        Output: (batch, seq_len, d_model)
        """
        batch_size, num_heads, seq_len, d_k = x.size()

        # Transpose to (batch, seq_len, num_heads, d_k)
        x = x.transpose(1, 2)

        # Reshape to (batch, seq_len, d_model)
        return x.contiguous().view(batch_size, seq_len, self.d_model)
```

```
58
59      def forward(self, query, key, value, mask=None):
60          """
61          Args:
62              query: (batch, seq_len_q, d_model)
63              key: (batch, seq_len_k, d_model)
64              value: (batch, seq_len_k, d_model)
65              mask: Optional (batch, seq_len_q, seq_len_k)
66          """
67          batch_size = query.size(0)
68
69          # Linear projections
70          Q = self.W_q(query)  # (batch, seq_len_q, d_model)
71          K = self.W_k(key)    # (batch, seq_len_k, d_model)
72          V = self.W_v(value)  # (batch, seq_len_k, d_model)
73
74          # Split into multiple heads
75          Q = self.split_heads(Q)  # (batch, num_heads,
    seq_len_q, d_k)
76          K = self.split_heads(K)  # (batch, num_heads,
    seq_len_k, d_k)
77          V = self.split_heads(V)  # (batch, num_heads,
    seq_len_k, d_k)
78
79          # Adjust mask for multiple heads
80          if mask is not None:
81              # Add head dimension: (batch, 1, seq_len_q,
    seq_len_k)
82              mask = mask.unsqueeze(1)
83
84          # Scaled dot-product attention
85          d_k = Q.size(-1)
86          scores = torch.matmul(Q, K.transpose(-2, -1)) / math.
    sqrt(d_k)
87
88          if mask is not None:
89              scores = scores.masked_fill(mask == 0, -1e9)
90
91          attention_weights = F.softmax(scores, dim=-1)
92          attention_weights = self.dropout(attention_weights)
93
94          # Apply attention to values
95          attention_output = torch.matmul(attention_weights, V)
96          # (batch, num_heads, seq_len_q, d_k)
97
98          # Combine heads
99          attention_output = self.combine_heads(
    attention_output)
100         # (batch, seq_len_q, d_model)
101
102         # Final linear projection
103         output = self.W_o(attention_output)
104
105         return output, attention_weights
106
107 # Test
108 mha = MultiHeadAttention(d_model=512, num_heads=8)
109 x = torch.randn(2, 10, 512)  # (batch, seq_len, d_model)
110 output, weights = mha(x, x, x)  # Self-attention
111
112 print(f"Output shape: {output.shape}")  # (2, 10, 512)
```

```
print(f"Attention weights shape: {weights.shape}")  # (2, 8, 10, 10)
```

11.4.3 Positional Encoding

```python
class PositionalEncoding(nn.Module):
    """Sinusoidal positional encoding."""

    def __init__(self, d_model, max_len=5000, dropout=0.1):
        super().__init__()
        self.dropout = nn.Dropout(dropout)

        # Create positional encoding matrix
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)

        # Compute div_term for scaling
        div_term = torch.exp(
            torch.arange(0, d_model, 2).float() *
            (-math.log(10000.0) / d_model)
        )

        # Apply sine to even indices
        pe[:, 0::2] = torch.sin(position * div_term)

        # Apply cosine to odd indices
        pe[:, 1::2] = torch.cos(position * div_term)

        # Add batch dimension: (1, max_len, d_model)
        pe = pe.unsqueeze(0)

        # Register as buffer (not a parameter, but part of state)
        self.register_buffer('pe', pe)

    def forward(self, x):
        """
        Args:
            x: (batch, seq_len, d_model)
        """
        # Add positional encoding
        x = x + self.pe[:, :x.size(1), :]
        return self.dropout(x)

# Test
pe = PositionalEncoding(d_model=512, max_len=100)
x = torch.randn(2, 50, 512)  # (batch, seq_len, d_model)
x_with_pos = pe(x)
print(x_with_pos.shape)  # (2, 50, 512)

# Visualize positional encoding
import matplotlib.pyplot as plt
plt.figure(figsize=(12, 4))
plt.imshow(pe.pe[0, :100, :].detach().numpy(), aspect='auto', cmap='viridis')
plt.xlabel('Dimension')
plt.ylabel('Position')
plt.colorbar()
plt.title('Positional Encoding')
plt.show()
```

11.4.4 Feed-Forward Network

```python
class PositionWiseFeedForward(nn.Module):
    """Position-wise feed-forward network."""

    def __init__(self, d_model, d_ff, dropout=0.1):
        """
        Args:
            d_model: Input/output dimension (e.g., 512)
            d_ff: Hidden dimension (typically 4 * d_model = 2048)
            dropout: Dropout probability
        """
        super().__init__()

        self.fc1 = nn.Linear(d_model, d_ff)
        self.fc2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        """
        Args:
            x: (batch, seq_len, d_model)
        """
        # First linear + ReLU
        x = F.relu(self.fc1(x))
        x = self.dropout(x)

        # Second linear
        x = self.fc2(x)

        return x

# Test
ffn = PositionWiseFeedForward(d_model=512, d_ff=2048)
x = torch.randn(2, 10, 512)
output = ffn(x)
print(output.shape)  # (2, 10, 512)
```

11.4.5 Complete Transformer Encoder Block

```python
class TransformerEncoderBlock(nn.Module):
    """Single Transformer encoder block."""

    def __init__(self, d_model, num_heads, d_ff, dropout=0.1):
        super().__init__()

        # Multi-head attention
        self.attention = MultiHeadAttention(d_model, num_heads, dropout)

        # Feed-forward network
        self.ffn = PositionWiseFeedForward(d_model, d_ff, dropout)

        # Layer normalization
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

        # Dropout
```

```
        self.dropout1 = nn.Dropout(dropout)
        self.dropout2 = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        """
        Args:
            x: (batch, seq_len, d_model)
            mask: Optional attention mask
        """
        # Multi-head self-attention with residual and layer norm
        attn_output, _ = self.attention(x, x, x, mask)
        x = x + self.dropout1(attn_output)  # Residual connection
        x = self.norm1(x)  # Layer normalization

        # Feed-forward with residual and layer norm
        ffn_output = self.ffn(x)
        x = x + self.dropout2(ffn_output)  # Residual connection
        x = self.norm2(x)  # Layer normalization

        return x

# Test
encoder_block = TransformerEncoderBlock(d_model=512, num_heads=8, d_ff=2048)
x = torch.randn(2, 10, 512)
output = encoder_block(x)
print(output.shape)  # (2, 10, 512)
```

[Layer Norm Placement: Pre-Norm vs Post-Norm] **Original Transformer (post-norm):**

```
x = norm(x + sublayer(x))
```

Modern practice (pre-norm):

```
x = x + sublayer(norm(x))
```

Pre-norm is now preferred:

- More stable training
- Can train deeper networks
- Better gradient flow

The code above uses post-norm (original paper), but many modern implementations use pre-norm.

11.4.6 Using PyTorch's Built-in Attention

```
# PyTorch provides nn.MultiheadAttention
attention = nn.MultiheadAttention(
    embed_dim=512,
    num_heads=8,
    dropout=0.1,
    batch_first=True  # Important! Use batch_first=True
)

# Self-attention
x = torch.randn(2, 10, 512)  # (batch, seq_len, embed_dim)
attn_output, attn_weights = attention(x, x, x)

print(attn_output.shape)  # (2, 10, 512)
print(attn_weights.shape)  # (2, 10, 10) if need_weights=True

# Cross-attention (e.g., encoder-decoder)
encoder_output = torch.randn(2, 20, 512)
decoder_input = torch.randn(2, 10, 512)

attn_output, _ = attention(
    query=decoder_input,
    key=encoder_output,
    value=encoder_output
)
print(attn_output.shape)  # (2, 10, 512)
```

[batch_first in MultiheadAttention] PyTorch's `nn.MultiheadAttention` defaults to `batch_first=False`, expecting shape (seq_len, batch, embed_dim).

Always use batch_first=True for consistency:

```
# CONFUSING (default):
attention = nn.MultiheadAttention(embed_dim=512, num_heads=8)
x = torch.randn(10, 2, 512)  # (seq_len, batch, embed_dim)

# CLEAR (recommended):
attention = nn.MultiheadAttention(embed_dim=512, num_heads=8,
                                  batch_first=True)
x = torch.randn(2, 10, 512)  # (batch, seq_len, embed_dim)
```

11.4.7 Masking for Causal Attention

For autoregressive models (e.g., language generation), prevent attending to future positions:

```
def create_causal_mask(seq_len):
    """
    Create causal mask to prevent attending to future
    positions.

    Returns:
        mask: (seq_len, seq_len) with 1s on and below
    diagonal
    """
    mask = torch.tril(torch.ones(seq_len, seq_len))
    return mask

# Example
mask = create_causal_mask(5)
```

```
print(mask)
"""
tensor([[1., 0., 0., 0., 0.],
        [1., 1., 0., 0., 0.],
        [1., 1., 1., 0., 0.],
        [1., 1., 1., 1., 0.],
        [1., 1., 1., 1., 1.]])
"""

# Use with attention
seq_len = 10
causal_mask = create_causal_mask(seq_len)

# PyTorch MultiheadAttention expects mask where True = ignore
# So we invert: 0 = attend, 1 = ignore
attn_mask = (1 - causal_mask).bool()

x = torch.randn(2, seq_len, 512)
attention = nn.MultiheadAttention(embed_dim=512, num_heads=8,
                                  batch_first=True)
output, _ = attention(x, x, x, attn_mask=attn_mask)
```

11.4.8 Complete Transformer for Classification

```
class TransformerClassifier(nn.Module):
    """Transformer for sequence classification."""

    def __init__(self, vocab_size, d_model, num_heads, num_layers,
                 num_classes, d_ff, max_len=512, dropout=0.1):
        super().__init__()

        # Token embedding
        self.embedding = nn.Embedding(vocab_size, d_model)

        # Positional encoding
        self.pos_encoding = PositionalEncoding(d_model, max_len, dropout)

        # Transformer encoder layers
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=d_model,
            nhead=num_heads,
            dim_feedforward=d_ff,
            dropout=dropout,
            batch_first=True
        )

        self.transformer_encoder = nn.TransformerEncoder(
            encoder_layer,
            num_layers=num_layers
        )

        # Classification head
        self.fc = nn.Linear(d_model, num_classes)

        self.d_model = d_model

    def forward(self, x, mask=None):
        """
```

```
        Args:
            x: (batch, seq_len) - token indices
            mask: Optional padding mask (batch, seq_len)
        """
        # Embedding and scaling
        x = self.embedding(x) * math.sqrt(self.d_model)

        # Add positional encoding
        x = self.pos_encoding(x)

        # Transformer encoding
        x = self.transformer_encoder(x, src_key_padding_mask=
    mask)

        # Global average pooling
        x = x.mean(dim=1)  # (batch, d_model)

        # Classification
        logits = self.fc(x)

        return logits

# Test
model = TransformerClassifier(
    vocab_size=10000,
    d_model=512,
    num_heads=8,
    num_layers=6,
    num_classes=3,
    d_ff=2048,
    dropout=0.1
)

# Random token sequences
x = torch.randint(0, 10000, (32, 50))  # (batch, seq_len)
output = model(x)
print(output.shape)  # (32, 3)

# Count parameters
total_params = sum(p.numel() for p in model.parameters())
print(f"Total parameters: {total_params:,}")
```

11.4.9 Transformer for Sequence-to-Sequence

```python
class TransformerSeq2Seq(nn.Module):
    """Complete Transformer with encoder and decoder."""

    def __init__(self, src_vocab_size, tgt_vocab_size, d_model=512,
                 num_heads=8, num_layers=6, d_ff=2048, dropout=0.1):
        super().__init__()

        # Source embeddings
        self.src_embedding = nn.Embedding(src_vocab_size, d_model)
        self.src_pos_encoding = PositionalEncoding(d_model, dropout=dropout)

        # Target embeddings
        self.tgt_embedding = nn.Embedding(tgt_vocab_size, d_model)
        self.tgt_pos_encoding = PositionalEncoding(d_model, dropout=dropout)

        # Transformer
        self.transformer = nn.Transformer(
            d_model=d_model,
            nhead=num_heads,
            num_encoder_layers=num_layers,
            num_decoder_layers=num_layers,
            dim_feedforward=d_ff,
            dropout=dropout,
            batch_first=True
        )

        # Output projection
        self.fc_out = nn.Linear(d_model, tgt_vocab_size)

        self.d_model = d_model

    def forward(self, src, tgt, src_mask=None, tgt_mask=None,
                src_padding_mask=None, tgt_padding_mask=None):
        """
        Args:
            src: (batch, src_seq_len) - source token indices
            tgt: (batch, tgt_seq_len) - target token indices
            src_mask: Optional source attention mask
            tgt_mask: Causal mask for target (prevent seeing future)
            src_padding_mask: Padding mask for source
            tgt_padding_mask: Padding mask for target
        """
        # Embed and add positional encoding
        src = self.src_embedding(src) * math.sqrt(self.d_model)
        src = self.src_pos_encoding(src)

        tgt = self.tgt_embedding(tgt) * math.sqrt(self.d_model)
        tgt = self.tgt_pos_encoding(tgt)

        # Transformer
```

```
        output = self.transformer(
            src, tgt,
            src_mask=src_mask,
            tgt_mask=tgt_mask,
            src_key_padding_mask=src_padding_mask,
            tgt_key_padding_mask=tgt_padding_mask
        )

        # Project to vocabulary
        logits = self.fc_out(output)

        return logits

    def generate(self, src, max_len, start_token, end_token):
        """
        Generate sequence autoregressively.

        Args:
            src: (batch, src_seq_len)
            max_len: Maximum generation length
            start_token: Token to start generation
            end_token: Token to stop generation
        """
        self.eval()
        batch_size = src.size(0)
        device = src.device

        # Encode source
        src = self.src_embedding(src) * math.sqrt(self.
    d_model)
        src = self.src_pos_encoding(src)
        memory = self.transformer.encoder(src)

        # Start with start_token
        tgt = torch.full((batch_size, 1), start_token,
                         dtype=torch.long, device=device)

        for _ in range(max_len):
            # Create causal mask
            tgt_mask = nn.Transformer.
    generate_square_subsequent_mask(
                tgt.size(1)
            ).to(device)

            # Decode
            tgt_embedded = self.tgt_embedding(tgt) * math.
    sqrt(self.d_model)
            tgt_embedded = self.tgt_pos_encoding(tgt_embedded
    )

            output = self.transformer.decoder(tgt_embedded,
    memory,
                                              tgt_mask=
    tgt_mask)

            # Project to vocabulary
            logits = self.fc_out(output[:, -1, :])  # Last
    position

            # Greedy decoding (take argmax)
            next_token = logits.argmax(dim=-1, keepdim=True)

```

```
            # Append to sequence
            tgt = torch.cat([tgt, next_token], dim=1)

            # Stop if all sequences generated end_token
            if (next_token == end_token).all():
                break

        return tgt

# Test
model = TransformerSeq2Seq(
    src_vocab_size=10000,
    tgt_vocab_size=10000,
    d_model=512,
    num_heads=8,
    num_layers=6
)

src = torch.randint(0, 10000, (2, 20))  # Source sequences
tgt = torch.randint(0, 10000, (2, 15))  # Target sequences

# Training forward pass
output = model(src, tgt[:, :-1])  # Shift target by 1
print(output.shape)  # (2, 14, 10000)

# Generation
generated = model.generate(src, max_len=20, start_token=1,
    end_token=2)
print(generated.shape)  # (2, <=20)
```

11.5 Exercises

11.1: Attention from Scratch - ★★

Goal: Implement and understand attention mechanism.

1. Implement scaled dot-product attention from scratch (without using the provided code)
2. Test on random Q, K, V matrices
3. Verify:
   - Attention weights sum to 1
   - Output shape is correct
   - Scaling by $\sqrt{d_k}$ stabilizes scores
4. Visualize attention weights as a heatmap

Visualization:

```python
import matplotlib.pyplot as plt

# Compute attention
output, weights = scaled_dot_product_attention(Q, K, V)

# Visualize attention weights (first sample in batch)
plt.figure(figsize=(8, 6))
plt.imshow(weights[0].detach().numpy(), cmap='viridis')
plt.colorbar()
plt.xlabel('Key Position')
plt.ylabel('Query Position')
plt.title('Attention Weights')
plt.show()
```

11.2: Multi-Head Attention - ★★★

Goal: Implement multi-head attention.

1. Complete the `MultiHeadAttention` class
2. Test with different numbers of heads (1, 2, 4, 8)
3. Verify outputs are identical when using 1 head vs your scaled dot-product attention
4. Visualize attention patterns from different heads
5. Compare parameter counts: single-head vs multi-head

Questions:

- Do different heads learn different attention patterns?
- How does performance change with number of heads?

11.3: Positional Encoding Analysis - ★★★

Goal: Understand positional encoding.

1. Implement sinusoidal positional encoding
2. Visualize the encoding for positions 0-100
3. Experiment:
   - Train a Transformer with positional encoding
   - Train without positional encoding
   - Use learned positional embeddings instead

4. Compare:
   - Final accuracy
   - Does model learn order information without PE?
   - Which PE works best?

Task: Sequence classification (e.g., sentiment analysis)

11.4: Complete Transformer Encoder - ★★★★

Goal: Build a full Transformer encoder for classification.

1. Implement complete Transformer encoder:
   - Multi-head attention
   - Position-wise FFN
   - Layer normalization
   - Residual connections
   - Positional encoding
2. Train on text classification task (e.g., IMDB sentiment)
3. Use vocabulary size = 10,000 (most common words)
4. Hyperparameters:
   - $d_{model} = 256$
   - num_heads = 8
   - num_layers = 4
   - $d_{ff} = 1024$
5. Achieve >85% accuracy
6. Visualize attention weights on example sentences

Dataset preparation:

```
from torchtext.datasets import IMDB
from torchtext.data.utils import get_tokenizer
from collections import Counter

tokenizer = get_tokenizer('basic_english')

# Build vocabulary from training data
def build_vocab(data_iter, tokenizer, max_size=10000):
    counter = Counter()
    for label, text in data_iter:
        counter.update(tokenizer(text))

    vocab = {word: i+2 for i, (word, _) in
             enumerate(counter.most_common(max_size))}
    vocab['<pad>'] = 0
    vocab['<unk>'] = 1

    return vocab

# Encode text
def encode_text(text, vocab, tokenizer, max_len=256):
    tokens = tokenizer(text)[:max_len]
    indices = [vocab.get(token, vocab['<unk>']) for
    token in tokens]

    # Pad to max_len
    indices += [vocab['<pad>']] * (max_len - len(indices
```

```
    ))

    return torch.LongTensor(indices)
```

11.5: Causal Attention for Language Modeling - ★★★★

Goal: Implement autoregressive generation with causal masking.

1. Build a decoder-only Transformer (like GPT)
2. Use causal masking (can't attend to future tokens)
3. Train on character-level language modeling:
   - Dataset: Shakespeare text or similar
   - Task: Predict next character
4. Implement generation:
   - Greedy decoding (argmax)
   - Top-k sampling
   - Temperature sampling
5. Generate text samples and evaluate quality

Generation strategies:

```
def generate_text(model, start_text, max_len=100, temperature=1.0,
                  top_k=None):
    """Generate text autoregressively."""
    model.eval()

    # Encode start text
    tokens = encode(start_text)

    with torch.no_grad():
        for _ in range(max_len):
            # Forward pass
            logits = model(tokens)

            # Get logits for last position
            logits = logits[-1] / temperature

            # Top-k filtering
            if top_k is not None:
                values, _ = torch.topk(logits, top_k)
                logits[logits < values[-1]] = -float('inf')

            # Sample from distribution
            probs = F.softmax(logits, dim=-1)
            next_token = torch.multinomial(probs, 1)

            # Append to sequence
            tokens = torch.cat([tokens, next_token])

    return decode(tokens)
```

11.6: Attention Visualization - ★★★★

Goal: Understand what Transformers learn.

1. Train a Transformer on sequence task
2. Extract attention weights from different layers and heads
3. Visualize:
   - Which heads focus on local context?
   - Which heads capture long-range dependencies?
   - How do patterns differ across layers?
4. Create attention heatmaps for example sentences
5. Analyze:
   - Do lower layers learn syntax?
   - Do higher layers learn semantics?

Visualization code:

```
def visualize_attention_heads(model, sentence, tokenizer):
    """Visualize attention patterns from all heads."""

    # Encode sentence
    tokens = encode(sentence, tokenizer)

    # Forward pass with hooks to capture attention
    attentions = []

    def hook_fn(module, input, output):
        # output[1] is attention weights
        attentions.append(output[1].detach())

    hooks = []
    for layer in model.transformer_encoder.layers:
        hook = layer.self_attn.register_forward_hook(hook_fn)
        hooks.append(hook)

    # Forward
    _ = model(tokens.unsqueeze(0))

    # Remove hooks
    for hook in hooks:
        hook.remove()

    # Plot attention from each layer and head
    num_layers = len(attentions)
    num_heads = attentions[0].size(1)

    fig, axes = plt.subplots(num_layers, num_heads,
                             figsize=(num_heads*3, num_layers*3))

    for layer in range(num_layers):
        for head in range(num_heads):
            ax = axes[layer, head]
            attn = attentions[layer][0, head].cpu().numpy()

            im = ax.imshow(attn, cmap='viridis')
            ax.set_title(f'L{layer+1} H{head+1}')
```

```
            ax.set_xlabel('Key')
            ax.set_ylabel('Query')

    plt.tight_layout()
    plt.show()

# Example
visualize_attention_heads(model, "The cat sat on the mat
    ", tokenizer)
```

11.6 Key Takeaways

Why Transformers dominate:

- **Parallelization:** All positions processed simultaneously (10-100× faster than RNNs)
- **Long-range dependencies:** Direct connections between any positions
- **Scalability:** Works with billions of parameters and tokens
- **Flexibility:** Minimal inductive biases, learns from data

Core components:

- **Scaled dot-product attention:** $\text{Attention}(Q, K, V) = \text{softmax}(QK^T/\sqrt{d_k})V$
- **Multi-head attention:** Multiple attention mechanisms in parallel
- **Positional encoding:** Inject position information (no inherent order)
- **Feed-forward networks:** Position-wise transformation
- **Residual connections + Layer norm:** Stable training

Architecture patterns:

- **Encoder-only:** BERT (classification, understanding)
- **Decoder-only:** GPT (generation, autoregressive)
- **Encoder-decoder:** T5, BART (translation, summarization)

Implementation tips:

- Always use `batch_first=True` in PyTorch
- Scale attention scores by $\sqrt{d_k}$ (stability)
- Use causal masking for autoregressive tasks
- Pre-norm is more stable than post-norm for deep networks
- Typical hyperparameters: $d_{model} = 512$, $h = 8$, $d_{ff} = 2048$

Training considerations:

- Transformers need lots of data (millions of examples)
- Use warmup learning rate schedule
- Gradient clipping less critical than RNNs (but still useful)
- Dropout on attention weights (0.1) and residual connections
- Label smoothing helps (0.1)

When to use Transformers:

- NLP tasks (now standard)
- Long sequences (>100 tokens)
- When you have lots of data
- When parallelization matters
- Vision tasks (Vision Transformer - ViT)

Limitations:

- **Quadratic complexity:** $O(n^2)$ in sequence length (memory and compute)
- **Data hungry:** Requires large datasets for good performance
- **Memory intensive:** Attention matrices can be huge
- **Position limit:** Most models have fixed maximum sequence length

Solutions to limitations:

- Sparse attention (only attend to subset of positions)
- Linear attention approximations (e.g., Linformer)
- Local attention windows (e.g., Longformer)
- Efficient Transformers (e.g., Reformer, Performer)

Common mistakes:

- Forgetting positional encoding (model can't distinguish positions)
- Wrong mask shape or type
- Not using causal mask for generation
- Forgetting to scale attention scores
- Using post-norm for very deep networks (use pre-norm)

Modern developments:

- **Vision Transformers (ViT):** Transformers for images
- **BERT:** Bidirectional encoder (masked language modeling)
- **GPT:** Decoder-only (autoregressive generation)
- **T5:** Unified text-to-text framework
- **LLaMA, ChatGPT:** Large language models at scale

Part III

Practical Skills

12 Debugging & Best Practices

12.1 Introduction: Why Debugging Matters

Deep learning is full of silent failures:

- Model trains but doesn't learn (loss stuck at baseline)
- Loss is NaN after a few iterations
- Training loss decreases but validation doesn't
- Model works on toy data but fails on real data
- Gradients vanish or explode

Unlike traditional programming where bugs crash immediately, deep learning bugs often manifest as **poor performance**. You need systematic debugging strategies.

12.2 Common Bugs and How to Fix Them

12.2.1 Bug 1: Loss is NaN

Symptoms:

```
Epoch 1: loss = 2.345
Epoch 2: loss = 1.876
Epoch 3: loss = nan
```

Causes and fixes:

1. Learning rate too high

```
# Check: Does loss explode before NaN?
# Fix: Reduce learning rate
optimizer = torch.optim.Adam(model.parameters(), lr=1e-5)  # Try lower
```

2. Numerical instability in loss

```
# BAD: Manual log can cause NaN
loss = -torch.log(predictions)  # NaN if predictions = 0

# GOOD: Use stable implementations
loss = F.cross_entropy(logits, targets)  # Numerically stable
# or
loss = F.binary_cross_entropy_with_logits(logits, targets)
```

3. Exploding gradients (RNNs)

```
# Fix: Add gradient clipping
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
```

4. Data contains NaN or Inf

```
# Check data
print(torch.isnan(data).any())
print(torch.isinf(data).any())

# Fix: Clean data or add checks
data = torch.nan_to_num(data, nan=0.0, posinf=1e6, neginf=-1e6)
```

Detection code:

```
def check_for_nan(model, data, target):
    """Debug NaN issues."""
    # Check inputs
    print(f"Data has NaN: {torch.isnan(data).any()}")
    print(f"Data has Inf: {torch.isinf(data).any()}")

    # Forward pass
    output = model(data)
    print(f"Output has NaN: {torch.isnan(output).any()}")

    # Backward pass
    loss = criterion(output, target)
    print(f"Loss has NaN: {torch.isnan(loss).any()}")
    loss.backward()

    # Check gradients
    for name, param in model.named_parameters():
        if param.grad is not None:
            if torch.isnan(param.grad).any():
                print(f"NaN gradient in {name}")
            if torch.isinf(param.grad).any():
                print(f"Inf gradient in {name}")
```

12.2.2 Bug 2: Loss Not Decreasing

Symptoms:

```
Epoch 1: loss = 2.345
Epoch 2: loss = 2.342
Epoch 3: loss = 2.340
...
Epoch 100: loss = 2.300  # Barely moving
```

Causes and fixes:

1. Learning rate too low

```
# Fix: Increase learning rate
optimizer = torch.optim.Adam(model.parameters(), lr=1e-2)  # Try higher

# Or use learning rate finder
def find_lr(model, train_loader, init_lr=1e-8, final_lr=10):
    """Find optimal learning rate."""
    lrs = []
    losses = []

    optimizer = torch.optim.SGD(model.parameters(), lr=init_lr)
    lr_scheduler = torch.optim.lr_scheduler.ExponentialLR(
        optimizer, gamma=(final_lr/init_lr)**(1/100)
    )

    for batch in train_loader:
        optimizer.zero_grad()
        loss = compute_loss(model, batch)
        loss.backward()
        optimizer.step()

        lrs.append(optimizer.param_groups[0]['lr'])
        losses.append(loss.item())

        lr_scheduler.step()

        if len(lrs) >= 100:
            break

    # Plot
    import matplotlib.pyplot as plt
    plt.plot(lrs, losses)
    plt.xscale('log')
    plt.xlabel('Learning Rate')
    plt.ylabel('Loss')
    plt.title('Learning Rate Finder')
    plt.show()

    # Optimal LR is usually where loss decreases fastest
    # (steepest negative gradient)
```

2. Wrong loss function

```
# BAD: Using MSE for classification
criterion = nn.MSELoss()  # Wrong for discrete labels!

# GOOD: Use appropriate loss
```

```
criterion = nn.CrossEntropyLoss()  # For classification
```

3. Forgot to zero gradients

```
# BAD: Gradients accumulate
for batch in dataloader:
    loss = criterion(model(x), y)
    loss.backward()
    optimizer.step()  # FORGOT optimizer.zero_grad()!

# GOOD:
for batch in dataloader:
    optimizer.zero_grad()  # Always zero first!
    loss = criterion(model(x), y)
    loss.backward()
    optimizer.step()
```

4. Data not normalized

```
# Check data range
print(f"Data min: {data.min()}, max: {data.max()}")

# Fix: Normalize
mean = data.mean()
std = data.std()
data = (data - mean) / (std + 1e-8)
```

5. Dead ReLUs or vanishing gradients

```
# Check activation distributions
def check_activations(model, data):
    activations = {}

    def hook(name):
        def hook_fn(module, input, output):
            activations[name] = output.detach()
        return hook_fn

    # Register hooks
    for name, module in model.named_modules():
        if isinstance(module, nn.ReLU):
            module.register_forward_hook(hook(name))

    model(data)

    # Check what percentage is zero
    for name, act in activations.items():
        zero_pct = (act == 0).float().mean() * 100
        print(f"{name}: {zero_pct:.1f}% zeros")

        if zero_pct > 50:
            print(f"  WARNING: More than 50% dead neurons!")

# Fix: Use Leaky ReLU or adjust initialization
```

12.2.3 Bug 3: Training Loss Decreases, Validation Doesn't

Symptoms: Overfitting

Causes and fixes:

1. Model too complex for data

```
# Fix: Reduce model size
model = SmallNet()  # Fewer layers, fewer parameters

# Or increase data
# Or add regularization (see below)
```

2. No regularization

```
# Add dropout
model = nn.Sequential(
    nn.Linear(100, 128),
    nn.ReLU(),
    nn.Dropout(0.5),  # Add dropout
    nn.Linear(128, 10)
)

# Add weight decay
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3,
                             weight_decay=1e-4)  # L2
    regularization

# Add data augmentation
transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ToTensor()
])
```

3. Training too long

```
# Implement early stopping
class EarlyStopping:
    def __init__(self, patience=10, min_delta=0):
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = None
        self.should_stop = False

    def __call__(self, val_loss):
        if self.best_loss is None:
            self.best_loss = val_loss
        elif val_loss > self.best_loss - self.min_delta:
            self.counter += 1
            if self.counter >= self.patience:
                self.should_stop = True
        else:
            self.best_loss = val_loss
            self.counter = 0

# Usage
early_stopping = EarlyStopping(patience=10)

for epoch in range(max_epochs):
```

```
    train_loss = train_epoch()
    val_loss = validate()

    early_stopping(val_loss)
    if early_stopping.should_stop:
        print(f"Early stopping at epoch {epoch}")
        break
```

12.2.4 Bug 4: Model Works on Small Data, Fails on Full Dataset

Causes and fixes:

1. Batch size too small

```
# Batch norm unstable with small batches
# Fix: Increase batch size or use Layer Norm

# Or use Gradient Accumulation
accumulation_steps = 4

for i, batch in enumerate(dataloader):
    loss = criterion(model(x), y)
    loss = loss / accumulation_steps  # Scale loss
    loss.backward()

    if (i + 1) % accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
```

2. Data distribution shift

```
# Check: Are train and val distributions similar?
def check_distribution(train_data, val_data):
    print(f"Train mean: {train_data.mean()}, std: {train_data.std()}")
    print(f"Val mean: {val_data.mean()}, std: {val_data.std()}")

    # Visualize
    import matplotlib.pyplot as plt
    plt.hist(train_data.flatten(), bins=50, alpha=0.5, label='Train')
    plt.hist(val_data.flatten(), bins=50, alpha=0.5, label='Val')
    plt.legend()
    plt.show()

# Fix: Normalize using training statistics
mean = train_data.mean()
std = train_data.std()

train_data = (train_data - mean) / std
val_data = (val_data - mean) / std  # Use training stats!
```

12.3 Debugging Strategies

12.3.1 Start Simple, Add Complexity

Step 1: Overfit on one batch

If you can't overfit one batch, something is fundamentally wrong.

```
# Get one batch
x, y = next(iter(train_loader))

# Try to overfit it
for epoch in range(1000):
    optimizer.zero_grad()
    output = model(x)
    loss = criterion(output, y)
    loss.backward()
    optimizer.step()

    if epoch % 100 == 0:
        print(f"Epoch {epoch}: Loss = {loss.item():.4f}")

# Should reach near-zero loss
# If not, check:
# - Model has enough capacity
# - Loss function is correct
# - Learning rate is reasonable
```

Step 2: Overfit on small dataset

Try 100 samples. Should still overfit easily.

Step 3: Add validation

Once overfitting works, add validation and regularization.

Step 4: Scale to full dataset

Now train on full data with proper regularization.

12.3.2 Check Shapes at Every Step

```
def debug_shapes(model, x):
    """Print shapes through the network."""
    print(f"Input: {x.shape}")

    for i, (name, module) in enumerate(model.named_children()
    ):
        x = module(x)
        print(f"{i}. {name}: {x.shape}")

    return x

# Usage
x = torch.randn(4, 3, 32, 32)
output = debug_shapes(model, x)
"""
Input: torch.Size([4, 3, 32, 32])
0. conv1: torch.Size([4, 64, 32, 32])
1. relu: torch.Size([4, 64, 32, 32])
2. pool: torch.Size([4, 64, 16, 16])
...
"""
```

12.3.3 Visualize Gradients

```python
def plot_grad_flow(named_parameters):
    """
    Plots gradient flow through the network.
    Helps identify vanishing/exploding gradients.
    """
    ave_grads = []
    max_grads = []
    layers = []

    for n, p in named_parameters:
        if p.requires_grad and p.grad is not None:
            layers.append(n)
            ave_grads.append(p.grad.abs().mean().item())
            max_grads.append(p.grad.abs().max().item())

    import matplotlib.pyplot as plt
    plt.figure(figsize=(12, 6))
    plt.bar(range(len(ave_grads)), ave_grads, alpha=0.5, label='mean')
    plt.bar(range(len(max_grads)), max_grads, alpha=0.5, label='max')
    plt.hlines(0, 0, len(ave_grads), linewidth=2, color='k')
    plt.xticks(range(len(layers)), layers, rotation='vertical')
    plt.xlabel('Layers')
    plt.ylabel('Gradient magnitude')
    plt.legend()
    plt.title('Gradient Flow')
    plt.grid(True)
    plt.tight_layout()
    plt.show()

# Usage after loss.backward()
plot_grad_flow(model.named_parameters())
```

12.3.4 Use Hooks for Debugging

```python
# Forward hook: inspect activations
def forward_hook(module, input, output):
    print(f"Forward: {output.shape}, mean={output.mean():.4f}")

# Backward hook: inspect gradients
def backward_hook(module, grad_input, grad_output):
    print(f"Backward: grad mean={grad_output[0].mean():.4f}")

# Register hooks
for name, module in model.named_modules():
    if isinstance(module, nn.Linear):
        module.register_forward_hook(forward_hook)
        module.register_backward_hook(backward_hook)

# Now run forward and backward
output = model(x)
loss = criterion(output, y)
loss.backward()

# Hooks will print debug info
```

12.4 Best Practices for Development

12.4.1 Code Organization

Project structure:

```
project/
 data/
    __init__.py
    dataset.py          # Dataset classes
    transforms.py       # Data augmentation
 models/
    __init__.py
    resnet.py           # Model architectures
    utils.py            # Model utilities
 utils/
    __init__.py
    train.py            # Training utilities
    eval.py             # Evaluation utilities
    metrics.py          # Custom metrics
 configs/
    config.yaml         # Hyperparameters
 train.py                # Main training script
 evaluate.py             # Evaluation script
 requirements.txt        # Dependencies
```

Separate concerns:

```python
# models/resnet.py
class ResNet(nn.Module):
    """Model definition only."""
    def __init__(self, ...):
        # Architecture

    def forward(self, x):
        # Forward pass
        return x

# utils/train.py
def train_epoch(model, loader, optimizer, criterion, device):
    """Training logic."""
    model.train()
    total_loss = 0

    for batch in loader:
        # Training step
        ...

    return total_loss / len(loader)

# train.py
def main():
    # Configuration
    # Data loading
    # Model creation
    # Training loop
    pass

if __name__ == "__main__":
    main()
```

12.4.2 Configuration Management

```yaml
# config.yaml
model:
  type: resnet18
  num_classes: 10

training:
  batch_size: 128
  epochs: 100
  learning_rate: 0.001
  weight_decay: 0.0001

data:
  root: ./data
  num_workers: 4
  augmentation: true
```

```python
# Load config
import yaml

with open('config.yaml', 'r') as f:
    config = yaml.safe_load(f)

# Use config
batch_size = config['training']['batch_size']
lr = config['training']['learning_rate']
```

12.4.3 Experiment Tracking

```python
# Simple logging
import logging

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('training.log'),
        logging.StreamHandler()
    ]
)

logger = logging.getLogger(__name__)

# Use in training
logger.info(f"Epoch {epoch}: Train Loss = {train_loss:.4f}")

# TensorBoard (better)
from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter('runs/experiment_1')

for epoch in range(num_epochs):
    train_loss = train_epoch()
    val_loss = validate()

    writer.add_scalar('Loss/train', train_loss, epoch)
    writer.add_scalar('Loss/val', val_loss, epoch)
    writer.add_scalar('LR', optimizer.param_groups[0]['lr'],
    epoch)

```

```
writer.close()

# View: tensorboard --logdir=runs
```

12.4.4 Reproducibility

```python
import random
import numpy as np
import torch

def set_seed(seed=42):
    """Set all random seeds for reproducibility."""
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)

    # Make cudnn deterministic
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False

# Call at start of training
set_seed(42)

# Note: This makes training slower but reproducible
# For production: set benchmark=True for speed
```

12.4.5 Checkpointing

```python
def save_checkpoint(model, optimizer, epoch, loss, path):
    """Save model checkpoint."""
    torch.save({
        'epoch': epoch,
        'model_state_dict': model.state_dict(),
        'optimizer_state_dict': optimizer.state_dict(),
        'loss': loss,
    }, path)

def load_checkpoint(model, optimizer, path):
    """Load model checkpoint."""
    checkpoint = torch.load(path)
    model.load_state_dict(checkpoint['model_state_dict'])
    optimizer.load_state_dict(checkpoint['
    optimizer_state_dict'])
    epoch = checkpoint['epoch']
    loss = checkpoint['loss']
    return epoch, loss

# Save best model
best_loss = float('inf')

for epoch in range(num_epochs):
    train_loss = train_epoch()
    val_loss = validate()

    # Save if best
    if val_loss < best_loss:
        best_loss = val_loss
        save_checkpoint(model, optimizer, epoch, val_loss,
                        'best_model.pth')

    # Save periodic checkpoints
    if epoch % 10 == 0:
        save_checkpoint(model, optimizer, epoch, val_loss,
```

```
35                          f'checkpoint_epoch_{epoch}.pth')
```

12.5 Debugging Checklist

When starting a new project:

1. Can model overfit one batch?
2. Can model overfit small dataset (100 samples)?
3. Are shapes correct at every layer?
4. Is loss function appropriate for task?
5. Is learning rate reasonable? (Try LR finder)
6. Is data normalized?
7. Are you using the right optimizer?
8. Did you set model.train() / model.eval()?

When loss is NaN:

1. Check for NaN in data
2. Reduce learning rate
3. Add gradient clipping
4. Use stable loss functions
5. Check for division by zero

When loss doesn't decrease:

1. Try higher learning rate
2. Check if you're zeroing gradients
3. Verify loss function is correct
4. Check data normalization
5. Verify model has enough capacity
6. Check for dead ReLUs

When overfitting:

1. Add dropout
2. Add weight decay
3. Use data augmentation
4. Reduce model size
5. Get more data
6. Use early stopping

12.6 Key Takeaways

Systematic debugging:

- Start simple (overfit one batch)
- Add complexity gradually
- Check shapes at every step
- Visualize gradients and activations
- Use hooks for inspection

Common bugs:

- NaN loss: LR too high, numerical instability, exploding gradients
- Loss not decreasing: LR too low, wrong loss, forgot zero_grad, bad data
- Overfitting: Model too complex, no regularization, training too long
- Works on toy data, fails on real: Batch size, distribution shift

Best practices:

- Separate code into modules (data, models, training)
- Use configuration files
- Track experiments (TensorBoard)
- Set random seeds for reproducibility
- Save checkpoints regularly
- Log everything

Development workflow:

1. Start with simple baseline
2. Verify can overfit small data
3. Add regularization
4. Scale to full dataset
5. Tune hyperparameters
6. Validate on held-out test set

Tools for debugging:

- Print statements (shapes, values, statistics)
- Hooks (forward and backward)
- TensorBoard (visualize training)
- Gradient visualization
- Learning rate finder
- Activation distribution checks

Remember:

- Deep learning bugs are often silent (poor performance, not crashes)
- Always start with simplest possible setup
- Check your data first (garbage in = garbage out)
- One change at a time (isolate what works)
- Keep good records (what did you try, what worked)

13 Training Best Practices

13.1 Introduction: From Working to Working Well

You've built a model that trains. Now: how do you make it train **well**?

This section covers practical strategies for:

- Choosing hyperparameters
- Learning rate strategies
- Efficient training
- Avoiding common pitfalls

13.2 Hyperparameter Tuning

13.2.1 Which Hyperparameters Matter Most?

Priority order:

1. Learning rate (MOST IMPORTANT)

Can make 10× difference in performance. Get this right first.

2. Batch size

Affects both training speed and generalization.

3. Architecture (width, depth)

Number of layers and neurons per layer.

4. Regularization (dropout, weight decay)

Prevents overfitting.

5. Learning rate schedule

How LR changes over time.

6. Everything else

Optimizer choice, activation functions, initialization—usually matter less.

13.2.2 Learning Rate Strategies

Strategy 1: Learning Rate Finder

Find optimal LR before training:

```
from torch_lr_finder import LRFinder

model = YourModel()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-7)
criterion = nn.CrossEntropyLoss()

lr_finder = LRFinder(model, optimizer, criterion, device="
    cuda")
lr_finder.range_test(train_loader, end_lr=10, num_iter=100)
lr_finder.plot()  # Shows loss vs LR

# Pick LR where loss decreases fastest (steepest slope)
# Usually 10   smaller than where loss explodes
optimal_lr = 1e-3  # Read from plot

```

```
lr_finder.reset()  # Reset model and optimizer
```

Strategy 2: Start Large, Decay

```
# Common schedules

# 1. Step decay (reduce every N epochs)
scheduler = torch.optim.lr_scheduler.StepLR(
    optimizer, step_size=30, gamma=0.1
)
# LR    0.1 every 30 epochs

# 2. Exponential decay
scheduler = torch.optim.lr_scheduler.ExponentialLR(
    optimizer, gamma=0.95
)
# LR    0.95 every epoch

# 3. Cosine annealing (smooth decay)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
    optimizer, T_max=100, eta_min=1e-6
)
# Smooth cosine decay from initial_lr to eta_min over T_max
    epochs

# 4. Reduce on plateau (adaptive)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(
    optimizer, mode='min', factor=0.5, patience=10
)
# Reduce LR by 0.5 if val loss doesn't improve for 10 epochs

# Usage
for epoch in range(num_epochs):
    train_loss = train_epoch()
    val_loss = validate()

    # For ReduceLROnPlateau, pass validation loss
    scheduler.step(val_loss)  # or scheduler.step() for
    others
```

Strategy 3: Warmup + Cosine Decay

State-of-the-art for Transformers:

```
from torch.optim.lr_scheduler import LambdaLR

def get_cosine_schedule_with_warmup(optimizer,
    num_warmup_steps,
                                    num_training_steps):
    """Warmup + cosine decay."""

    def lr_lambda(current_step):
        # Warmup
        if current_step < num_warmup_steps:
            return float(current_step) / float(max(1,
    num_warmup_steps))

        # Cosine decay
        progress = float(current_step - num_warmup_steps) / \
                   float(max(1, num_training_steps -
    num_warmup_steps))
```

```
        return max(0.0, 0.5 * (1.0 + math.cos(math.pi *
    progress)))

    return LambdaLR(optimizer, lr_lambda)

# Usage
num_epochs = 100
steps_per_epoch = len(train_loader)
num_training_steps = num_epochs * steps_per_epoch
num_warmup_steps = 5 * steps_per_epoch  # 5 epochs warmup

optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3)
scheduler = get_cosine_schedule_with_warmup(
    optimizer, num_warmup_steps, num_training_steps
)

for epoch in range(num_epochs):
    for batch in train_loader:
        # Training step
        ...
        optimizer.step()
        scheduler.step()  # Update LR every step, not epoch!
```

13.2.3 Batch Size Selection

Trade-offs:

Batch Size	Pros	Cons
Small (8-32)	Better generalization	Slower, noisy gradients
	Less memory	Batch norm unstable
Medium (64-256)	Good balance	Standard choice
Large (512+)	Faster training	Worse generalization
	More stable	Requires careful tuning

Table 9: Batch size trade-offs

Rules of thumb:

- **Start with 32-128:** Usually works well
- **Larger batches need higher LR:** Scale by $\sqrt{\text{batch size}}$
- **Use gradient accumulation** if memory limited
- **Don't go below 8** if using batch norm

Linear scaling rule:

If you double batch size, double learning rate (approximately).

```
# Base setup
batch_size = 64
lr = 0.001

# Double batch size
batch_size = 128
lr = 0.002  # Double LR

# But: this is approximate! Always validate on your task
```

13.2.4 Model Architecture Selection

Width vs Depth:

```
# Wider networks (fewer layers, more neurons)
model_wide = MLP([100, 1024, 1024, 10])  # 2 hidden layers, 1024 neurons

# Deeper networks (more layers, fewer neurons)
model_deep = MLP([100, 256, 256, 256, 256, 10])  # 4 hidden layers, 256 neurons

# Modern preference: deeper is usually better (with skip connections)
```

Start small, scale up:

1. Start with small model that trains fast
2. Check if it underfits (high train loss)
3. If underfitting: increase capacity
4. If overfitting: add regularization

13.3 Regularization Strategies

13.3.1 Dropout

```
# Standard dropout
model = nn.Sequential(
    nn.Linear(100, 256),
    nn.ReLU(),
    nn.Dropout(0.5),  # Drop 50% during training
    nn.Linear(256, 10)
)

# Guidelines:
# - Start with 0.5, tune if needed
# - Higher dropout = more regularization
# - Use lower dropout (0.1-0.2) for convolutional layers
# - Higher dropout (0.5-0.7) for large fully connected layers
```

13.3.2 Weight Decay (L2 Regularization)

```
# Add weight decay to optimizer
optimizer = torch.optim.Adam(
    model.parameters(),
    lr=1e-3,
    weight_decay=1e-4  # L2 penalty
)

# Typical values: 1e-5 to 1e-3
# Larger weight decay = stronger regularization

# Note: Use AdamW for better weight decay
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3,
                              weight_decay=1e-4)
```

13.3.3 Data Augmentation

```
# For images
train_transform = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.RandomCrop(32, padding=4),
    transforms.ColorJitter(brightness=0.2, contrast=0.2),
    transforms.RandomRotation(15),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])

# NO augmentation for validation/test
val_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
                         std=[0.229, 0.224, 0.225])
])
```

13.4 Training Monitoring

13.4.1 What to Track

Essential metrics:

- Training loss (per epoch)
- Validation loss (per epoch)
- Validation accuracy/metric (per epoch)
- Learning rate (per epoch or step)

Useful but optional:

- Gradient norms
- Weight norms
- Training time per epoch
- GPU memory usage

```python
# Complete training loop with monitoring
from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter('runs/experiment')

for epoch in range(num_epochs):
    # Training
    model.train()
    train_loss = 0
    train_correct = 0
    train_total = 0

    for batch in train_loader:
        optimizer.zero_grad()

        x, y = batch
        x, y = x.to(device), y.to(device)

        output = model(x)
        loss = criterion(output, y)
        loss.backward()

        # Track gradient norm
        grad_norm = torch.nn.utils.clip_grad_norm_(
            model.parameters(), max_norm=1.0
        )

        optimizer.step()

        train_loss += loss.item()
        pred = output.argmax(dim=1)
        train_correct += (pred == y).sum().item()
        train_total += y.size(0)

    # Validation
    model.eval()
    val_loss = 0
    val_correct = 0
    val_total = 0

    with torch.no_grad():
        for batch in val_loader:
            x, y = batch
```

```
            x, y = x.to(device), y.to(device)

            output = model(x)
            loss = criterion(output, y)

            val_loss += loss.item()
            pred = output.argmax(dim=1)
            val_correct += (pred == y).sum().item()
            val_total += y.size(0)

    # Compute metrics
    train_loss /= len(train_loader)
    train_acc = train_correct / train_total
    val_loss /= len(val_loader)
    val_acc = val_correct / val_total

    # Log to TensorBoard
    writer.add_scalar('Loss/train', train_loss, epoch)
    writer.add_scalar('Loss/val', val_loss, epoch)
    writer.add_scalar('Accuracy/train', train_acc, epoch)
    writer.add_scalar('Accuracy/val', val_acc, epoch)
    writer.add_scalar('LR', optimizer.param_groups[0]['lr'],
    epoch)
    writer.add_scalar('GradNorm', grad_norm, epoch)

    # Print
    print(f"Epoch {epoch}: Train Loss={train_loss:.4f}, "
          f"Val Loss={val_loss:.4f}, Val Acc={val_acc:.4f}")

    # Step scheduler
    scheduler.step()

writer.close()
```

13.4.2 Recognizing Training Issues from Curves

Healthy training:

```
Train loss: Decreasing smoothly
Val loss: Decreasing, tracking train loss
Gap: Small (< 10% difference)
```

Overfitting:

```
Train loss: Very low
Val loss: High and increasing
Gap: Large and growing
→ Solution: More regularization, more data
```

Underfitting:

```
Train loss: High
Val loss: High (similar to train)
Gap: Small
→ Solution: Bigger model, train longer
```

Unstable training:

```
Both losses: Oscillating wildly
→ Solution: Lower learning rate, gradient clipping
```

13.5 Model Selection and Evaluation

13.5.1 Proper Train/Val/Test Split

```
# WRONG: Tune on test set
# This gives overly optimistic results!

# RIGHT: Three separate sets
# 1. Training set (70%): Train model
# 2. Validation set (15%): Tune hyperparameters
# 3. Test set (15%): Final evaluation (ONCE!)

from sklearn.model_selection import train_test_split

# Split data
X_train, X_temp, y_train, y_temp = train_test_split(
    X, y, test_size=0.3, stratify=y, random_state=42
)
X_val, X_test, y_val, y_test = train_test_split(
    X_temp, y_temp, test_size=0.5, stratify=y_temp,
    random_state=42
)

# Training process:
# 1. Train many models with different hyperparameters
# 2. Evaluate all on validation set
# 3. Pick best model
# 4. Train best model on train+val (optional)
# 5. Evaluate ONCE on test set
# 6. Report test performance
```

13.5.2 Cross-Validation for Small Datasets

```
from sklearn.model_selection import KFold

kfold = KFold(n_splits=5, shuffle=True, random_state=42)

scores = []

for fold, (train_idx, val_idx) in enumerate(kfold.split(X)):
    print(f"Fold {fold + 1}")

    # Split data
    X_train, X_val = X[train_idx], X[val_idx]
    y_train, y_val = y[train_idx], y[val_idx]

    # Create datasets
    train_dataset = YourDataset(X_train, y_train)
    val_dataset = YourDataset(X_val, y_val)

    # Train model (reset for each fold!)
    model = YourModel()
    optimizer = torch.optim.Adam(model.parameters())

    # Train...
    val_score = train_and_evaluate(model, train_dataset,
    val_dataset)
    scores.append(val_score)

# Report mean and std
print(f"CV Score: {np.mean(scores):.4f}    {np.std(scores):.4
    f}")
```

13.6 Practical Tips

13.6.1 Start with Good Defaults

```
# Proven configuration for most tasks:

optimizer = torch.optim.AdamW(
    model.parameters(),
    lr=1e-3,           # Will likely need tuning
    weight_decay=1e-4,
    betas=(0.9, 0.999)
)

scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
    optimizer, T_max=num_epochs
)

# Initialization: PyTorch defaults are usually good
# But for RNNs, use orthogonal init for hidden-to-hidden

# Batch size: 64 or 128
# Epochs: Start with 100, use early stopping
# Gradient clipping: max_norm=1.0 (for RNNs)
```

13.6.2 Hyperparameter Search Strategies

Grid search (exhaustive):

```
lrs = [1e-4, 1e-3, 1e-2]
weight_decays = [0, 1e-5, 1e-4, 1e-3]
dropouts = [0.2, 0.5]

best_val_loss = float('inf')
best_config = None

for lr in lrs:
    for wd in weight_decays:
        for dropout in dropouts:
            val_loss = train_and_evaluate(lr, wd, dropout)

            if val_loss < best_val_loss:
                best_val_loss = val_loss
                best_config = (lr, wd, dropout)

print(f"Best: LR={best_config[0]}, WD={best_config[1]}, "
      f"Dropout={best_config[2]}")
```

Random search (more efficient):

```
import random

def random_config():
    lr = 10 ** random.uniform(-5, -2)  # 1e-5 to 1e-2
    wd = 10 ** random.uniform(-6, -3)  # 1e-6 to 1e-3
    dropout = random.uniform(0.1, 0.7)
    return lr, wd, dropout

num_trials = 20
results = []

for trial in range(num_trials):
```

```
    lr, wd, dropout = random_config()
    val_loss = train_and_evaluate(lr, wd, dropout)
    results.append((val_loss, lr, wd, dropout))

# Sort by validation loss
results.sort()
best = results[0]
print(f"Best: Val Loss={best[0]:.4f}, LR={best[1]:.2e}, "
      f"WD={best[2]:.2e}, Dropout={best[3]:.2f}")
```

Bayesian optimization (most efficient):

```
# Use libraries like Optuna or Ray Tune
import optuna

def objective(trial):
    # Suggest hyperparameters
    lr = trial.suggest_loguniform('lr', 1e-5, 1e-2)
    wd = trial.suggest_loguniform('weight_decay', 1e-6, 1e-3)
    dropout = trial.suggest_uniform('dropout', 0.1, 0.7)

    # Train and return validation loss
    val_loss = train_and_evaluate(lr, wd, dropout)
    return val_loss

# Optimize
study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=50)

print(f"Best hyperparameters: {study.best_params}")
print(f"Best validation loss: {study.best_value:.4f}")
```

13.7 Key Takeaways

Hyperparameter priorities:

1. Learning rate (MOST IMPORTANT)
2. Batch size
3. Architecture (depth, width)
4. Regularization (dropout, weight decay)
5. Everything else

Learning rate strategies:

- Use LR finder to find good starting point
- Start high, decay over time
- Cosine annealing works well
- Warmup helps for Transformers
- ReduceLROnPlateau is safe fallback

Batch size selection:

- Start with 32-128
- Larger batch = faster but may hurt generalization
- Scale LR with batch size (approximately)
- Use gradient accumulation if memory limited

Regularization:

- Dropout: 0.5 for FC layers, 0.1-0.2 for conv layers
- Weight decay: 1e-5 to 1e-3
- Data augmentation: Always for images
- Early stopping: Always use

Training monitoring:

- Track train/val loss and metrics
- Use TensorBoard for visualization
- Watch for overfitting (gap between train/val)
- Watch for underfitting (both losses high)

Model selection:

- Always use separate test set
- Tune on validation set only
- Test set evaluated ONCE at the end
- Use cross-validation for small datasets

Hyperparameter search:

- Random search better than grid search
- Bayesian optimization most efficient
- Start with good defaults
- Log everything

Common mistakes:

- Not tuning learning rate
- Evaluating on test set during development
- Using validation statistics to normalize test data
- Not using early stopping
- Training for too many epochs

14 Performance & Optimization

14.1 Introduction: Making Training Faster

Training can be slow. A model that takes 1 hour vs 10 hours per epoch makes a huge difference in development speed.

This section covers:

- Mixed precision training (2× speedup)
- GPU optimization
- DataLoader efficiency
- Memory management
- Profiling tools

14.2 Mixed Precision Training

Idea: Use FP16 (half precision) instead of FP32 (full precision) for most operations.

Benefits:

- 2-3× faster training
- 2× less memory (can use larger batch sizes)
- Minimal accuracy loss

14.2.1 Automatic Mixed Precision (AMP)

PyTorch provides automatic mixed precision via `torch.cuda.amp`:

```
from torch.cuda.amp import autocast, GradScaler

model = YourModel().cuda()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

# Create gradient scaler
scaler = GradScaler()

for epoch in range(num_epochs):
    for x, y in train_loader:
        x, y = x.cuda(), y.cuda()

        optimizer.zero_grad()

        # Forward pass in mixed precision
        with autocast():
            output = model(x)
            loss = criterion(output, y)

        # Backward pass with gradient scaling
        scaler.scale(loss).backward()

        # Unscale gradients and step
        scaler.step(optimizer)
        scaler.update()
```

How it works:

1. `autocast()`: Automatically chooses FP16 or FP32 for each operation
2. `GradScaler`: Scales loss to prevent underflow in FP16 gradients

[When to Use Mixed Precision] **Use AMP when:**

- Training on modern GPUs (Volta, Turing, Ampere)
- You want faster training
- You're memory-limited

Don't use if:

- GPU doesn't support FP16 (e.g., older GPUs)
- Model has numerical instability issues
- You need exact FP32 precision (rare)

Typical speedup: 1.5-3× depending on model and GPU.

14.2.2 Gradient Checkpointing

Trade computation for memory:

```
from torch.utils.checkpoint import checkpoint

class MemoryEfficientBlock(nn.Module):
    def __init__(self, ...):
        super().__init__()
        self.layer = ExpensiveLayer()

    def forward(self, x):
        # Use checkpointing for this layer
        return checkpoint(self.layer, x)

# Saves activations strategically
# Recomputes forward pass during backward (uses more time,
    less memory)
```

When to use:

- Very deep models (100+ layers)
- Running out of GPU memory
- Willing to trade 20-30% slower training for 2× memory reduction

14.3 GPU Optimization

14.3.1 Data Transfer Optimization

Pin memory for faster CPU→GPU transfer:

```
train_loader = DataLoader(
    dataset,
    batch_size=128,
    num_workers=4,
    pin_memory=True  # Faster transfers
)

# Also call .cuda(non_blocking=True)
for x, y in train_loader:
    x = x.cuda(non_blocking=True)
    y = y.cuda(non_blocking=True)

    # Model forward...
```

Keep data on GPU when possible:

```
# BAD: Moving data back and forth
for x, y in train_loader:
    x = x.cuda()
    output = model(x)
    output = output.cpu()  # Unnecessary transfer!
    loss = criterion(output, y.cpu())  # Another transfer!

# GOOD: Keep everything on GPU
for x, y in train_loader:
    x, y = x.cuda(), y.cuda()
    output = model(x)  # Stays on GPU
    loss = criterion(output, y)  # All on GPU
```

14.3.2 Batch Size and GPU Utilization

```
import torch

# Check GPU utilization
# nvidia-smi in terminal

# If GPU utilization < 80%, try:
# 1. Increase batch size
train_loader = DataLoader(dataset, batch_size=256)  # Larger

# 2. Increase num_workers
train_loader = DataLoader(dataset, batch_size=128,
    num_workers=8)

# 3. Use larger model (if not memory-limited)
```

14.3.3 Multi-GPU Training

DataParallel (simple but not recommended):

```
if torch.cuda.device_count() > 1:
    print(f"Using {torch.cuda.device_count()} GPUs")
    model = nn.DataParallel(model)

```

```
model = model.cuda()

# Training as usual
# Batch automatically split across GPUs
```

DistributedDataParallel (better, recommended):

```
import torch.distributed as dist
from torch.nn.parallel import DistributedDataParallel as DDP

def setup(rank, world_size):
    dist.init_process_group("nccl", rank=rank, world_size=world_size)

def cleanup():
    dist.destroy_process_group()

def train(rank, world_size):
    setup(rank, world_size)

    # Create model and move to GPU
    model = YourModel().to(rank)
    model = DDP(model, device_ids=[rank])

    # Create distributed sampler
    sampler = torch.utils.data.distributed.DistributedSampler(
        dataset, num_replicas=world_size, rank=rank
    )

    train_loader = DataLoader(dataset, batch_size=128, sampler=sampler)

    # Training loop as usual
    for epoch in range(num_epochs):
        sampler.set_epoch(epoch)  # Important for shuffling!
        for x, y in train_loader:
            # Training step...
            pass

    cleanup()

# Launch with torch.multiprocessing
import torch.multiprocessing as mp
mp.spawn(train, args=(world_size,), nprocs=world_size, join=True)

# Or use torchrun:
# torchrun --nproc_per_node=4 train.py
```

DDP vs DataParallel:

- DDP: Faster, more efficient, recommended
- DP: Simpler but slower, single-process

14.4 DataLoader Optimization

14.4.1 Optimal Number of Workers

```python
import time

def benchmark_dataloader(num_workers):
    """Find optimal num_workers."""
    loader = DataLoader(dataset, batch_size=128,
                        num_workers=num_workers, pin_memory=
    True)

    start = time.time()
    for _ in loader:
        pass  # Just iterate
    elapsed = time.time() - start

    return elapsed

# Test different values
for num_workers in [0, 2, 4, 8, 16]:
    elapsed = benchmark_dataloader(num_workers)
    print(f"num_workers={num_workers}: {elapsed:.2f}s")

# Typical optimal: 4-8 workers
# More workers = more CPU usage but faster
# Too many workers = overhead dominates
```

Guidelines:

- Start with `num_workers=4`
- If CPU usage low: increase
- If CPU usage 100%: decrease
- Typical range: 2-8 workers

14.4.2 Prefetching

PyTorch DataLoader automatically prefetches, but you can optimize:

```python
# Use persistent_workers to avoid recreating workers
train_loader = DataLoader(
    dataset,
    batch_size=128,
    num_workers=4,
    pin_memory=True,
    persistent_workers=True,  # Keep workers alive between
    epochs
    prefetch_factor=2  # Number of batches to prefetch per
    worker
)
```

14.5 Memory Management

14.5.1 Monitoring Memory Usage

```python
import torch

def print_memory_usage():
    """Print current GPU memory usage."""
    if torch.cuda.is_available():
```

```
        allocated = torch.cuda.memory_allocated() / 1e9
        reserved = torch.cuda.memory_reserved() / 1e9
        print(f"Allocated: {allocated:.2f} GB")
        print(f"Reserved: {reserved:.2f} GB")

# Check memory before and after operations
print("Before model creation:")
print_memory_usage()

model = LargeModel().cuda()

print("After model creation:")
print_memory_usage()
```

14.5.2 Reducing Memory Usage

1. Use gradient accumulation instead of large batches:

```
# Instead of batch_size=512 (OOM)
# Use batch_size=128 with accumulation

accumulation_steps = 4
effective_batch_size = 128 * 4  # = 512

optimizer.zero_grad()

for i, (x, y) in enumerate(train_loader):
    output = model(x)
    loss = criterion(output, y)
    loss = loss / accumulation_steps  # Scale loss
    loss.backward()  # Accumulate gradients

    if (i + 1) % accumulation_steps == 0:
        optimizer.step()
        optimizer.zero_grad()
```

2. Clear cache periodically:

```
# If encountering OOM during long training
if epoch % 10 == 0:
    torch.cuda.empty_cache()
```

3. Delete unnecessary variables:

```
# In evaluation, don't keep gradients
with torch.no_grad():
    for x, y in val_loader:
        output = model(x)
        # ...

# If you need to keep large tensors temporarily
large_tensor = compute_something()
result = process(large_tensor)
del large_tensor  # Free memory immediately
torch.cuda.empty_cache()
```

4. Use smaller data types:

```
# FP32 (default)
x = torch.randn(1000, 1000)  # 4 MB
```

```
3
4 # FP16 (half)
5 x = torch.randn(1000, 1000, dtype=torch.float16)  # 2 MB
6
7 # For labels, use appropriate type
8 labels = torch.randint(0, 10, (1000,), dtype=torch.long)  #
    Not float!
```

14.6 Profiling and Bottleneck Analysis

14.6.1 PyTorch Profiler

```
from torch.profiler import profile, ProfilerActivity

model = YourModel().cuda()

# Profile training
with profile(
    activities=[ProfilerActivity.CPU, ProfilerActivity.CUDA],
    record_shapes=True,
    profile_memory=True,
    with_stack=True
) as prof:
    for i, (x, y) in enumerate(train_loader):
        if i >= 10:  # Profile first 10 batches
            break

        x, y = x.cuda(), y.cuda()

        optimizer.zero_grad()
        output = model(x)
        loss = criterion(output, y)
        loss.backward()
        optimizer.step()

# Print profiling results
print(prof.key_averages().table(sort_by="cuda_time_total",
    row_limit=10))

# Export for Chrome tracing
prof.export_chrome_trace("trace.json")
# View at chrome://tracing
```

14.6.2 Simple Timing

```
import time

# Time entire epoch
start = time.time()
for x, y in train_loader:
    # Training step...
    pass
elapsed = time.time() - start
print(f"Epoch time: {elapsed:.2f}s")

# Time individual operations
torch.cuda.synchronize()  # Wait for GPU
start = time.time()

output = model(x)

torch.cuda.synchronize()
elapsed = time.time() - start
print(f"Forward pass: {elapsed*1000:.2f}ms")
```

14.6.3 Identify Bottlenecks

```
def profile_training_loop(model, train_loader, num_batches
    =10):
```

```
    """Profile different parts of training."""

    times = {
        'data_loading': 0,
        'to_gpu': 0,
        'forward': 0,
        'backward': 0,
        'optimizer': 0,
        'total': 0
    }

    total_start = time.time()
    data_start = time.time()

    for i, (x, y) in enumerate(train_loader):
        if i >= num_batches:
            break

        times['data_loading'] += time.time() - data_start

        # Transfer to GPU
        gpu_start = time.time()
        x, y = x.cuda(), y.cuda()
        torch.cuda.synchronize()
        times['to_gpu'] += time.time() - gpu_start

        # Forward
        forward_start = time.time()
        output = model(x)
        torch.cuda.synchronize()
        times['forward'] += time.time() - forward_start

        # Backward
        backward_start = time.time()
        loss = criterion(output, y)
        loss.backward()
        torch.cuda.synchronize()
        times['backward'] += time.time() - backward_start

        # Optimizer
        opt_start = time.time()
        optimizer.step()
        optimizer.zero_grad()
        torch.cuda.synchronize()
        times['optimizer'] += time.time() - opt_start

        data_start = time.time()

    times['total'] = time.time() - total_start

    # Print breakdown
    print("Time breakdown:")
    for key, value in times.items():
        if key != 'total':
            pct = (value / times['total']) * 100
            print(f"  {key}: {value:.3f}s ({pct:.1f}%)")
        else:
            print(f"  {key}: {value:.3f}s")

# Usage
profile_training_loop(model, train_loader)
"""
```

```
Time breakdown:
  data_loading: 0.421s (25.3%)
  to_gpu: 0.089s (5.3%)
  forward: 0.567s (34.1%)
  backward: 0.423s (25.4%)
  optimizer: 0.165s (9.9%)
  total: 1.665s
"""

# If data_loading is high: increase num_workers
# If forward/backward is high: model is slow (expected)
# If to_gpu is high: use pin_memory, non_blocking transfers
```

14.7 Production Deployment

14.7.1 Model Export

TorchScript (JIT compilation):

```python
# Option 1: Tracing
model = YourModel()
model.eval()

example_input = torch.randn(1, 3, 224, 224)
traced_model = torch.jit.trace(model, example_input)

# Save
traced_model.save("model_traced.pt")

# Load and use
loaded_model = torch.jit.load("model_traced.pt")
output = loaded_model(example_input)
```

```python
# Option 2: Scripting (handles control flow)
scripted_model = torch.jit.script(model)
scripted_model.save("model_scripted.pt")
```

ONNX (cross-framework):

```python
import torch.onnx

model = YourModel()
model.eval()

dummy_input = torch.randn(1, 3, 224, 224)

torch.onnx.export(
    model,
    dummy_input,
    "model.onnx",
    export_params=True,
    opset_version=11,
    input_names=['input'],
    output_names=['output'],
    dynamic_axes={'input': {0: 'batch_size'},
                  'output': {0: 'batch_size'}}
)

# Can now use in other frameworks (TensorFlow, etc.)
```

14.7.2 Inference Optimization

```python
# 1. Set to eval mode
model.eval()

# 2. Disable gradient computation
with torch.no_grad():
    output = model(input)

# 3. Use FP16 if possible
with torch.cuda.amp.autocast():
    output = model(input)

# 4. Batch inference
```

```
# Process multiple inputs together
batch = torch.stack([input1, input2, input3])
outputs = model(batch)

# 5. Use torch.jit for production
model = torch.jit.load("model.pt")
```

14.8 Key Takeaways

Mixed precision training:

- Use `torch.cuda.amp` for 2× speedup
- Minimal code changes required
- Works on modern GPUs (Volta+)
- Typical speedup: 1.5-3×

GPU optimization:

- Keep data on GPU (avoid CPUGPU transfers)
- Use `pin_memory=True` and `non_blocking=True`
- Maximize batch size (within memory limits)
- Use DistributedDataParallel for multi-GPU

DataLoader optimization:

- Start with `num_workers=4`
- Use `persistent_workers=True`
- Benchmark different worker counts
- Typical optimal: 4-8 workers

Memory management:

- Use gradient accumulation for large effective batch sizes
- Clear cache periodically if needed
- Delete unnecessary variables
- Use gradient checkpointing for very deep models

Profiling:

- Use PyTorch Profiler for detailed analysis
- Identify bottlenecks (data loading, forward, backward)
- Focus optimization efforts on slowest parts
- Profile regularly during development

Production deployment:

- Use TorchScript for inference
- Export to ONNX for cross-framework compatibility
- Always use `model.eval()` and `torch.no_grad()`
- Batch inference when possible
- Consider FP16 inference

Common performance issues:

- Data loading too slow → increase `num_workers`

- GPU underutilized → increase batch size or model size
- Out of memory → reduce batch size, use gradient accumulation, or AMP
- Training too slow → use mixed precision, profile bottlenecks

Part IV
Integrative Challenges

15 Grand Challenges

15.1 Introduction: Putting It All Together

You've learned the fundamentals. Now it's time to combine everything into complete, real-world projects.

These challenges integrate multiple concepts:

- Multiple architecture types (CNNs, RNNs, Transformers)
- Proper training pipelines (data loading, optimization, monitoring)
- Best practices (regularization, debugging, performance)
- Scientific computing applications

How to approach these challenges:

1. Start simple (baseline model)
2. Get it working (overfit small data)
3. Make it good (full dataset + regularization)
4. Make it great (hyperparameter tuning + tricks)
5. Document everything (what worked, what didn't)

15.2 Challenge 1: Time Series Forecasting System

Difficulty: ★★★

Goal: Build a complete forecasting system for multivariate time series.

Dataset: Use real data (weather, stock prices, sensor readings) or generate complex synthetic data.

Requirements:

1. Data Pipeline

- Load and preprocess time series data
- Create sliding windows (past 100 steps → predict next 10)
- Handle missing values
- Normalize using training statistics
- Split into train/val/test (temporal split, not random!)

2. Model Architecture

Implement and compare three approaches:

- **LSTM-based:** Stacked LSTM with attention
- **CNN-based:** 1D CNN with residual connections
- **Transformer-based:** Encoder-only Transformer

3. Training

- Proper learning rate schedule (warmup + decay)

- Early stopping on validation set
- Gradient clipping (essential for RNNs)
- Mixed precision training
- TensorBoard logging

4. Evaluation

- Multiple metrics: MSE, MAE, MAPE
- One-step ahead vs multi-step ahead prediction
- Visualization: predicted vs actual
- Confidence intervals (optional)

5. Analysis

- Which architecture works best?
- How far ahead can you predict accurately?
- What patterns does the model learn?
- Where does it fail?

Starter template:

```
class TimeSeriesDataset(Dataset):
    def __init__(self, data, window_size, forecast_horizon):
        # Your implementation
        pass

    def __len__(self):
        return len(self.data) - self.window_size - self.
    forecast_horizon + 1

    def __getitem__(self, idx):
        # Return (input_window, target_window)
        pass

class LSTMForecaster(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers,
    forecast_horizon):
        super().__init__()
        self.lstm = nn.LSTM(input_size, hidden_size,
    num_layers,
                            batch_first=True)
        self.fc = nn.Linear(hidden_size, forecast_horizon)

    def forward(self, x):
        # LSTM encoding
        output, (h_n, c_n) = self.lstm(x)
        # Predict from final hidden state
        prediction = self.fc(h_n[-1])
        return prediction

# Your training loop with all best practices
```

Bonus challenges:

- Add attention mechanism to LSTM
- Implement probabilistic forecasting (predict distribution, not point estimate)
- Handle multiple related time series (multivariate)
- Deploy as REST API

15.3 Challenge 2: Hybrid Vision-Language Model

Difficulty: ★★★★

Goal: Build a model that combines visual and textual information.

Task: Image captioning or visual question answering.

Requirements:

1. Architecture

Implement encoder-decoder with:

- **Image encoder:** ResNet or Vision Transformer
- **Text decoder:** LSTM or Transformer decoder
- **Cross-modal attention:** Decoder attends to image features

```
class ImageCaptioningModel(nn.Module):
    def __init__(self, vocab_size, embed_dim=512, hidden_dim=512):
        super().__init__()

        # Image encoder (pretrained ResNet)
        resnet = models.resnet50(pretrained=True)
        # Remove final classification layer
        self.encoder = nn.Sequential(*list(resnet.children())[:-2])

        # Freeze encoder (transfer learning)
        for param in self.encoder.parameters():
            param.requires_grad = False

        # Project image features
        self.image_proj = nn.Linear(2048, embed_dim)

        # Text decoder
        self.embedding = nn.Embedding(vocab_size, embed_dim)
        self.lstm = nn.LSTM(embed_dim, hidden_dim, num_layers=2,
                            batch_first=True)

        # Attention mechanism
        self.attention = nn.MultiheadAttention(embed_dim, num_heads=8)

        # Output projection
        self.fc_out = nn.Linear(hidden_dim, vocab_size)

    def forward(self, images, captions):
        # Encode images: (batch, 2048, 7, 7)
        img_features = self.encoder(images)

        # Reshape: (batch, 49, 2048)
        batch_size = img_features.size(0)
        img_features = img_features.view(batch_size, 2048, -1)
        img_features = img_features.transpose(1, 2)

        # Project: (batch, 49, embed_dim)
        img_features = self.image_proj(img_features)

        # Embed captions
```

```
        caption_embeds = self.embedding(captions)

        # LSTM decoding with attention
        lstm_out, _ = self.lstm(caption_embeds)

        # Attend to image features
        attn_out, _ = self.attention(lstm_out, img_features,
    img_features)

        # Combine and project
        combined = lstm_out + attn_out
        output = self.fc_out(combined)

        return output
```

2. Data Preparation

- Use COCO dataset or similar
- Build vocabulary from captions
- Tokenize captions
- Implement proper data augmentation for images
- Handle variable-length captions (padding + masking)

3. Training

- Teacher forcing during training
- Proper cross-entropy loss (ignore padding tokens)
- Learning rate warmup + decay
- Mixed precision training
- Gradient accumulation (if memory limited)

4. Generation

Implement multiple decoding strategies:

```
def generate_caption(model, image, vocab, max_len=20,
                     strategy='greedy', temperature=1.0,
    beam_width=5):
    """
    Generate caption from image.

    Args:
        strategy: 'greedy', 'sampling', or 'beam_search'
    """
    model.eval()

    # Encode image
    with torch.no_grad():
        img_features = model.encode_image(image)

    # Start token
    caption = [vocab['<start>']]

    if strategy == 'greedy':
        # Greedy decoding (take argmax at each step)
        for _ in range(max_len):
            logits = model.decode_step(img_features, caption)
            next_word = logits.argmax()
            caption.append(next_word.item())
```

```
            if next_word == vocab['<end>']:
                break

    elif strategy == 'sampling':
        # Sample from distribution
        for _ in range(max_len):
            logits = model.decode_step(img_features, caption)
            logits = logits / temperature
            probs = F.softmax(logits, dim=-1)
            next_word = torch.multinomial(probs, 1)
            caption.append(next_word.item())
            if next_word == vocab['<end>']:
                break

    elif strategy == 'beam_search':
        # Beam search (keep top-k candidates)
        # Your implementation
        pass

    return caption
```

5. Evaluation

- BLEU score (text similarity metric)
- Qualitative: Visual inspection of generated captions
- Error analysis: Where does model fail?
- Attention visualization: What does model look at?

Bonus challenges:

- Implement beam search decoding
- Add image-text contrastive learning (CLIP-style)
- Fine-tune encoder (not just freeze)
- Add copy mechanism for rare words

15.4 Challenge 3: Denoising Scientific Data

Difficulty: ★★★

Goal: Build a U-Net style architecture to denoise 2D scientific data.

Application: Medical images, microscopy, sensor data, astronomical images.

Requirements:

1. Data Generation

Create noisy data:

```
def generate_noisy_data(clean_images, noise_level=0.1):
    """Add Gaussian noise to clean images."""
    noise = torch.randn_like(clean_images) * noise_level
    noisy = clean_images + noise
    return noisy, clean_images

# Use real dataset (medical images, etc.) or synthetic
```

2. U-Net Architecture

Implement complete U-Net with:

- Encoder: Downsampling path with residual blocks
- Decoder: Upsampling path with skip connections
- Attention gates (optional but recommended)
- Batch normalization throughout

```
class UNet(nn.Module):
    """U-Net for image denoising/reconstruction."""

    def __init__(self, in_channels=1, out_channels=1):
        super().__init__()

        # Encoder
        self.enc1 = self.conv_block(in_channels, 64)
        self.enc2 = self.conv_block(64, 128)
        self.enc3 = self.conv_block(128, 256)
        self.enc4 = self.conv_block(256, 512)

        # Bottleneck
        self.bottleneck = self.conv_block(512, 1024)

        # Decoder with skip connections
        self.upconv4 = nn.ConvTranspose2d(1024, 512, 2,
    stride=2)
        self.dec4 = self.conv_block(1024, 512)  # 1024 = 512
    + 512 from skip

        self.upconv3 = nn.ConvTranspose2d(512, 256, 2, stride
    =2)
        self.dec3 = self.conv_block(512, 256)

        self.upconv2 = nn.ConvTranspose2d(256, 128, 2, stride
    =2)
        self.dec2 = self.conv_block(256, 128)

        self.upconv1 = nn.ConvTranspose2d(128, 64, 2, stride
    =2)
```

```
        self.dec1 = self.conv_block(128, 64)

        # Output
        self.out = nn.Conv2d(64, out_channels, 1)

        self.pool = nn.MaxPool2d(2, 2)

    def conv_block(self, in_ch, out_ch):
        """Double convolution block."""
        return nn.Sequential(
            nn.Conv2d(in_ch, out_ch, 3, padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_ch, out_ch, 3, padding=1),
            nn.BatchNorm2d(out_ch),
            nn.ReLU(inplace=True)
        )

    def forward(self, x):
        # Encoder
        enc1 = self.enc1(x)
        enc2 = self.enc2(self.pool(enc1))
        enc3 = self.enc3(self.pool(enc2))
        enc4 = self.enc4(self.pool(enc3))

        # Bottleneck
        bottleneck = self.bottleneck(self.pool(enc4))

        # Decoder with skip connections
        dec4 = self.upconv4(bottleneck)
        dec4 = torch.cat([dec4, enc4], dim=1)
        dec4 = self.dec4(dec4)

        dec3 = self.upconv3(dec4)
        dec3 = torch.cat([dec3, enc3], dim=1)
        dec3 = self.dec3(dec3)

        dec2 = self.upconv2(dec3)
        dec2 = torch.cat([dec2, enc2], dim=1)
        dec2 = self.dec2(dec2)

        dec1 = self.upconv1(dec2)
        dec1 = torch.cat([dec1, enc1], dim=1)
        dec1 = self.dec1(dec1)

        return self.out(dec1)
```

3. Training

- Loss function: MSE + Perceptual loss (optional)
- Data augmentation: Random flips, rotations
- Learning rate schedule
- Monitor both MSE and PSNR metrics

4. Evaluation

```
def calculate_psnr(img1, img2):
    """Calculate Peak Signal-to-Noise Ratio."""
    mse = torch.mean((img1 - img2) ** 2)
    if mse == 0:
        return float('inf')
```

```
    max_pixel = 1.0
    psnr = 20 * torch.log10(max_pixel / torch.sqrt(mse))
    return psnr.item()

def calculate_ssim(img1, img2):
    """Structural Similarity Index (use skimage or pytorch).
    """
    from skimage.metrics import structural_similarity
    return structural_similarity(
        img1.cpu().numpy(), img2.cpu().numpy(),
        data_range=1.0
    )

# Evaluate
model.eval()
psnrs = []
ssims = []

with torch.no_grad():
    for noisy, clean in test_loader:
        denoised = model(noisy)

        for i in range(len(noisy)):
            psnr = calculate_psnr(denoised[i], clean[i])
            ssim = calculate_ssim(denoised[i], clean[i])
            psnrs.append(psnr)
            ssims.append(ssim)

print(f"Average PSNR: {np.mean(psnrs):.2f} dB")
print(f"Average SSIM: {np.mean(ssims):.4f}")
```

5. Visualization

```
def visualize_denoising(model, noisy_images, clean_images):
    """Visualize denoising results."""
    model.eval()

    with torch.no_grad():
        denoised = model(noisy_images)

    fig, axes = plt.subplots(len(noisy_images), 3, figsize
    =(12, 4*len(noisy_images)))

    for i in range(len(noisy_images)):
        # Noisy
        axes[i, 0].imshow(noisy_images[i].squeeze().cpu(),
    cmap='gray')
        axes[i, 0].set_title('Noisy')
        axes[i, 0].axis('off')

        # Denoised
        axes[i, 1].imshow(denoised[i].squeeze().cpu(), cmap='
    gray')
        axes[i, 1].set_title('Denoised')
        axes[i, 1].axis('off')

        # Clean
        axes[i, 2].imshow(clean_images[i].squeeze().cpu(),
    cmap='gray')
        axes[i, 2].set_title('Clean (Target)')
        axes[i, 2].axis('off')

```

```
    plt.tight_layout()
    plt.show()
```

Bonus challenges:

- Add attention gates between encoder and decoder
- Implement progressive training (start with low resolution)
- Handle different noise types (Gaussian, salt-and-pepper, Poisson)
- Add perceptual loss using pretrained VGG

15.5 Challenge 4: Custom Transformer for Sequence Tasks

Difficulty: ★★★★

Goal: Build a complete Transformer from scratch for sequence classification or generation.

Task: Text classification, sequence generation, or time series classification.

Requirements:

1. Complete Transformer Implementation

Build all components from scratch (no `nn.Transformer`):

- Multi-head attention
- Position-wise feed-forward
- Positional encoding
- Layer normalization
- Encoder and/or decoder blocks

2. Training Infrastructure

- Warmup learning rate schedule
- Gradient clipping
- Label smoothing
- Mixed precision training
- Checkpointing and recovery

3. Advanced Features

Implement at least two of:

- Learning rate warmup + cosine annealing
- Custom learning rate schedule
- Gradient accumulation
- Model ensemble
- Knowledge distillation

4. Comprehensive Evaluation

- Validation metrics
- Attention visualization
- Error analysis
- Compare with baseline (LSTM, simple MLP)
- Ablation studies (remove components, measure impact)

5. Analysis

Answer these questions:

- How many heads are optimal?
- What does each attention head learn?
- How deep should the model be?
- What's the effect of positional encoding?
- Where does the model fail?

15.6 Challenge 5: End-to-End ML Pipeline

Difficulty: ★★★★

Goal: Build a production-ready machine learning system.

Task: Choose your own (classification, regression, generation).

Requirements:

1. Complete Pipeline

```
project/
 data/
    raw/              # Raw data
    processed/        # Preprocessed data
    preprocessing.py  # Data processing scripts
 models/
    architectures.py  # Model definitions
    pretrained/       # Saved models
 training/
    train.py          # Training script
    evaluate.py       # Evaluation script
    utils.py          # Training utilities
 configs/
    base_config.yaml  # Base configuration
    experiment_*.yaml # Experiment configs
 notebooks/
    exploration.ipynb # Data exploration
    analysis.ipynb    # Results analysis
 tests/
    test_*.py         # Unit tests
 requirements.txt
 README.md
 main.py               # Entry point
```

2. Code Quality

- Type hints
- Docstrings for all functions
- Unit tests for critical functions
- Configuration via YAML
- Logging (not just print statements)
- Error handling

3. Experiment Tracking

- TensorBoard or Weights & Biases
- Log hyperparameters
- Log metrics over time
- Save model checkpoints
- Version control (Git)

4. Reproducibility

- Set random seeds
- Save exact hyperparameters

- Save training logs
- Document environment (requirements.txt)
- Save data preprocessing steps

5. Deployment

- Export model (TorchScript or ONNX)
- Create inference script
- Docker container (optional)
- REST API using FastAPI (optional)
- Documentation for usage

Evaluation criteria:

- Model performance
- Code quality and organization
- Reproducibility
- Documentation
- Completeness of pipeline

15.7 Evaluation and Next Steps

How to know you've succeeded:

1. **It works:** Model trains without errors
2. **It learns:** Loss decreases, metrics improve
3. **It generalizes:** Validation performance is good
4. **You understand it:** Can explain why it works
5. **It's reproducible:** Others can replicate your results

Beyond this guide:

Keep learning:

- Read recent papers (arXiv, conferences)
- Implement papers from scratch
- Contribute to open source projects
- Join competitions (Kaggle, etc.)
- Build projects in your domain

Advanced topics to explore:

- Self-supervised learning
- Meta-learning
- Neural architecture search
- Efficient networks (quantization, pruning)
- Diffusion models
- Large language models
- Multimodal learning
- Reinforcement learning

Stay updated:

- Follow PyTorch blog and releases

- Join ML communities (Reddit, Discord, forums)
- Attend conferences (NeurIPS, ICML, ICLR)
- Read technical blogs
- Experiment with new techniques

15.8 Final Thoughts

You now have a comprehensive toolkit for deep learning with PyTorch. You've learned:

- **Foundations:** Tensors, autograd, modules, training loops, data loading
- **Architectures:** MLPs, CNNs, ResNets, RNNs, Transformers
- **Techniques:** Normalization, regularization, optimization
- **Practical skills:** Debugging, training best practices, performance optimization

Remember:

- Start simple, add complexity gradually
- Always check if your model can overfit small data
- Visualization and monitoring are essential
- Good engineering matters as much as good models
- Deep learning is iterative—experiment and learn

The best way to master these concepts is to **build things**. Pick a challenge that excites you and start coding!

Good luck, and happy building!