Mathematical Foundations of Neural Networks (Vol 1)

A Rigorous Treatment with Detailed Derivations

February 12, 2026

Contents

1 Mathematical Preliminaries **15**
- 1.1 Vectors and Matrices . . . 15
  - 1.1.1 Vector Spaces and Norms . . . 15
  - 1.1.2 Dot Product and Geometric Interpretation . . . 15
  - 1.1.3 Matrix Operations . . . 16
  - 1.1.4 Important Matrix Properties . . . 16
- 1.2 Calculus: Foundations of Optimization . . . 17
  - 1.2.1 Derivatives and the Chain Rule . . . 17
  - 1.2.2 Partial Derivatives and Gradients . . . 17
  - 1.2.3 Matrix Calculus . . . 18

Notation, Dimensions, and Label Conventions **19**

2 The Perceptron **23**
- 2.1 Decision Boundary as a Hyperplane . . . 23
  - 2.1.1 Hyperplane interpretation . . . 23
  - 2.1.2 Scaling invariance . . . 23
- 2.2 Signed Distance to the Hyperplane . . . 24
  - 2.2.1 Derivation . . . 24
- 2.3 Perceptron Learning Algorithm . . . 24
  - 2.3.1 Label convention . . . 24
  - 2.3.2 Update rule . . . 24
- 2.4 Perceptron Convergence Theorem (Sketch) . . . 25
  - 2.4.1 Linear separability and margin . . . 25
  - 2.4.2 Theorem . . . 25
  - 2.4.3 Proof sketch . . . 25
  - 2.4.4 Remarks . . . 26

3 Feedforward Neural Networks **27**
- 3.1 From Scalar Sums to Matrix Form . . . 27
  - 3.1.1 Single neuron (scalar form) . . . 27
  - 3.1.2 Layer of neurons (summation index notation) . . . 27
  - 3.1.3 Layer of neurons (matrix form) . . . 27
  - 3.1.4 Mini-batch (fully vectorized) form . . . 28
- 3.2 Network as Function Composition . . . 28
  - 3.2.1 Parameter count . . . 28
- 3.3 Activation Functions and Derivatives . . . 28
  - 3.3.1 Why nonlinearity is required . . . 29

3.3.2 Sigmoid . . . 29
3.3.3 Hyperbolic tangent . . . 29
3.3.4 ReLU . . . 29
3.3.5 Softmax . . . 29
3.3.6 Vanishing gradients (preview) . . . 30
3.4 A Fully Worked Tiny Example (Optional) . . . 30

4 Loss Functions **31**
4.1 Empirical Risk Minimization . . . 31
4.2 Regression: Mean Squared Error . . . 31
4.2.1 Definition . . . 31
4.2.2 Gradient w.r.t. the prediction . . . 31
4.3 Binary Classification: Binary Cross-Entropy . . . 32
4.3.1 Binary cross-entropy (negative log-likelihood) . . . 32
4.3.2 Gradient w.r.t. $\hat{y}$ . . . 32
4.3.3 Sigmoid + BCE simplification (logit gradient) . . . 32
4.4 Multi-class Classification: Softmax and Categorical Cross-Entropy . . . 32
4.4.1 Categorical cross-entropy . . . 32
4.4.2 Softmax Jacobian . . . 33
4.4.3 Softmax + CCE simplification (logit gradient) . . . 33
4.5 (Optional) Huber Loss for Robust Regression . . . 33

5 Backpropagation Algorithm **35**
5.1 Setup: Notation and Forward Pass . . . 35
5.2 The Delta Terms . . . 35
5.2.1 Definition . . . 35
5.2.2 Output layer delta: general form . . . 35
5.2.3 Hidden layer delta . . . 36
5.3 Gradients for Weights and Biases . . . 36
5.3.1 Single example . . . 36
5.3.2 Mini-batch (average gradient) . . . 36
5.3.3 Mini-batch matrix backpropagation (vectorized deltas) . . . 37
5.4 Two Classic "Cancellations" . . . 38
5.4.1 Sigmoid + Binary Cross-Entropy . . . 38
5.4.2 Softmax + Categorical Cross-Entropy . . . 38
5.5 Vanishing Gradients (Why Depth Is Hard) . . . 39
5.5.1 Mitigations (mathematical view) . . . 39
5.6 Algorithm Summary (One Iteration) . . . 39

6 Concrete Numerical Example: A Network from Scratch **41**
6.1 Problem Setup . . . 41
6.2 Parameter Initialization . . . 41
6.3 Forward Propagation: General Form . . . 42
6.4 Forward Propagation: Sample 1 (Fully Expanded) . . . 42
6.4.1 Layer 1 pre-activation . . . 42
6.4.2 Layer 1 activation (ReLU) . . . 42
6.4.3 Layer 2 pre-activation . . . 42
6.4.4 Output (sigmoid) . . . 43
6.4.5 Loss for sample 1 . . . 43

6.5 Forward Propagation: Sample 2 (Detailed) . . . 43
6.5.1 Layer 1 . . . 43
6.5.2 Layer 2 and output . . . 43
6.6 Batch Loss . . . 44
6.7 Backpropagation: General Form . . . 44
6.8 Backpropagation: Sample 1 (Fully Expanded) . . . 44
6.8.1 Output delta . . . 44
6.8.2 Gradients for layer 2 . . . 45
6.8.3 Hidden delta . . . 45
6.8.4 Gradients for layer 1 . . . 45
6.9 Mini-batch Gradients (Averaging) . . . 45
6.10 Parameter Updates (SGD) . . . 46
6.10.1 Update layer 2 . . . 46
6.10.2 Update layer 1 . . . 46
6.11 Second Iteration (Forward Pass Check) . . . 46

7 Advanced Numerical Demonstrations 47
7.1 Optimization Dynamics . . . 47
7.1.1 Effect of the learning rate . . . 47
7.1.2 Loss curve (illustrative) . . . 48
7.2 Regularization Example: L2 . . . 48
7.2.1 Definition . . . 48
7.2.2 Concrete computation . . . 48
7.2.3 Gradient effect (weight decay view) . . . 48
7.3 Momentum Optimization . . . 49
7.3.1 Update rule . . . 49
7.3.2 Two-step numerical example (Layer 2 weights) . . . 49
7.4 Batch Normalization (Numerical Calculation) . . . 49
7.4.1 Definition . . . 49
7.4.2 Concrete computation for one neuron . . . 50
7.5 Dropout Regularization (Numerical Calculation) . . . 50
7.5.1 Training-time dropout . . . 50
7.5.2 Test-time scaling . . . 51
7.6 Multi-sample Vectorized Processing . . . 51

8 Optimization Techniques 53
8.1 Stochastic Gradient Descent (SGD) . . . 53
8.1.1 Full-batch gradient descent . . . 53
8.1.2 Mini-batch SGD . . . 53
8.1.3 Learning-rate schedules (common choices) . . . 54
8.1.4 A basic descent inequality (smooth case) . . . 54
8.2 Momentum . . . 54
8.2.1 Heavy-ball momentum . . . 54
8.2.2 Nesterov accelerated gradient (NAG) . . . 54
8.3 Adaptive Methods (RMSProp, Adam, AdamW) . . . 55
8.3.1 RMSProp (core idea) . . . 55
8.3.2 Adam (Adaptive Moment Estimation) . . . 55
8.3.3 AdamW (decoupled weight decay) . . . 55

8.4 Regularization as Optimization . . . 55
8.4.1 L2 regularization (weight decay) and MAP interpretation . . . 56
8.4.2 L1 regularization and sparsity . . . 56
8.4.3 Dropout (inverted dropout) . . . 56
8.4.4 Batch normalization (BN) . . . 57
8.5 Stability Tricks (practical) . . . 57
8.5.1 Gradient clipping . . . 57
8.5.2 Mini-batch size trade-off . . . 57

9 Analysis and Theory **59**
9.1 Function Approximation . . . 59
9.1.1 Setting and notation . . . 59
9.1.2 Universal Approximation Theorem (UAT) . . . 59
9.1.3 Approximation rates (why UAT is not enough) . . . 60
9.1.4 Why depth helps (compositional structure) . . . 60
9.2 Depth vs. Width . . . 60
9.2.1 Expressivity measures . . . 60
9.2.2 Piecewise linear regions (ReLU intuition) . . . 60
9.2.3 Separation results (functions needing depth) . . . 61
9.3 Optimization Landscapes . . . 61
9.3.1 Nonconvexity and critical points . . . 61
9.3.2 Saddle points in high dimension (intuition) . . . 61
9.3.3 Overparameterization and benign landscapes (idea) . . . 61
9.4 Generalization . . . 62
9.4.1 Train vs. test . . . 62
9.4.2 Bias–variance decomposition (squared loss) . . . 62
9.4.3 Capacity control and uniform convergence (sketch) . . . 62
9.4.4 Implicit regularization (phenomenon) . . . 62
9.5 Interpolation and Double Descent . . . 63
9.5.1 Classical U-shaped curve . . . 63
9.5.2 Interpolation threshold . . . 63
9.5.3 Double descent (empirical phenomenon) . . . 63
9.6 What theory does *not* yet explain . . . 63

10 Computational Graphs and Automatic Differentiation **65**
10.1 Computational Graphs: Formal Definition . . . 65
10.1.1 Directed acyclic graphs (DAGs) . . . 65
10.1.2 Example: Simple expression graph . . . 65
10.1.3 Forward evaluation . . . 66
10.2 Automatic Differentiation: Backpropagation in DAGs . . . 66
10.2.1 Generalized chain rule . . . 66
10.2.2 Example: Computing adjoints for $y = (x_1 + x_2) \cdot x_1$ . . . 66
10.3 Forward Mode vs. Reverse Mode Differentiation . . . 67
10.3.1 Reverse mode (backpropagation) . . . 67
10.3.2 Forward mode (tangent linear) . . . 67
10.3.3 Comparison table . . . 68
10.4 Chain Rule in Multivariate Form . . . 68
10.4.1 Jacobian-vector products . . . 68

10.4.2 Example: Softmax backward . . . 68

11 Numerical Stability and Precision **69**
11.1 Softmax and the Log-Sum-Exp Trick . . . 69
11.1.1 Naive softmax (numerically unstable) . . . 69
11.1.2 Stable variant (log-sum-exp) . . . 69
11.1.3 Log-domain computation . . . 69
11.2 Underflow and Overflow in Deep Networks . . . 70
11.2.1 Activation norms . . . 70
11.2.2 Initialization and gradient norms . . . 70
11.2.3 Gradient clipping . . . 70
11.3 Mixed Precision Training . . . 70
11.3.1 Strategy . . . 70
11.3.2 Why scaling helps . . . 70

12 Tensor Operations and Notation **71**
12.1 Tensors and Index Notation . . . 71
12.1.1 Definition . . . 71
12.1.2 Einstein notation (summation convention) . . . 71
12.2 Broadcasting and Element-wise Operations . . . 72
12.2.1 Broadcasting rules (NumPy/PyTorch convention) . . . 72
12.3 Reshape and Transpose . . . 72
12.3.1 Reshape (view) . . . 72
12.3.2 Transpose (permutation) . . . 72
12.3.3 Flattening (vectorization) . . . 73

13 Hyperparameter Tuning and Learning Rate Schedules **75**
13.1 Learning Rate Selection . . . 75
13.1.1 Learning rate finder (LRFinder) . . . 75
13.1.2 Learning rate schedules . . . 75
13.2 Warmup . . . 76
13.2.1 Linear warmup . . . 76
13.2.2 Gradient accumulation + warmup . . . 76
13.3 Hyperparameter Search Methods . . . 76
13.3.1 Grid search . . . 76
13.3.2 Random search . . . 77
13.3.3 Bayesian optimization . . . 77

14 Data Preprocessing and Normalization **79**
14.1 Input Normalization . . . 79
14.1.1 Standardization (Z-score) . . . 79
14.1.2 Min-Max scaling . . . 79
14.1.3 Data statistics (train vs. test) . . . 79
14.2 Batch Normalization (Revisited) . . . 80
14.2.1 Running mean and variance (inference) . . . 80
14.3 Layer Normalization . . . 80
14.4 Group Normalization and Instance Normalization . . . 80
14.4.1 Group normalization . . . 80
14.4.2 Instance normalization . . . 80

14.5 Comparison of Normalization Methods 81

15 Recurrent Neural Networks (RNNs) 83
15.1 Sequence Data and Mathematical Formulation 83
15.1.1 Temporal Data Representation 83
15.1.2 Notation and Convention 83
15.1.3 Common Task Architectures 83
15.2 Vanilla RNN Definition and Forward Propagation 84
15.2.1 Recurrent Computation 84
15.2.2 Parameter Sharing Across Time 84
15.2.3 Vectorized Mini-batch Forward Pass 84
15.3 Computational Graph Unrolling in Time 85
15.3.1 Unrolled Graph Representation 85
15.3.2 Temporal Dependencies 85
15.4 Backpropagation Through Time (BPTT) 86
15.4.1 Loss Function and Objective 86
15.4.2 Backpropagation Through Time Algorithm 86
15.4.3 Parameter Gradients 86
15.5 Vanishing and Exploding Gradients: Mathematical Analysis 87
15.5.1 Gradient Flow Through Hidden States 87
15.5.2 Spectral Analysis 87
15.5.3 Mathematical Condition for Stability 88
15.6 Common Questions (RNNs) 88
15.6.1 Q1: Why do we share $\mathbf{W}_{hh}$? 88
15.6.2 Q2: What exactly is "vanishing gradient"? 88
15.6.3 Q3: What is the computational cost of BPTT? 89
15.6.4 Q4: Why can't RNNs be parallelized? 89
15.7 Common Questions (Gradient Problems) 90
15.7.1 Q5: What is the danger of exploding gradients? 90
15.7.2 Q6: Why is the spectral radius important? 90

16 Long Short-Term Memory (LSTM) 91
16.1 Motivation and Design Principles 91
16.1.1 The Problem with Vanilla RNNs 91
16.1.2 The LSTM Solution: Additive State Update 91
16.2 Complete LSTM Cell Definition 92
16.2.1 Gate Computations 92
16.2.2 State and Hidden State Updates 92
16.3 Conceptual Intuition: The Notebook Analogy 93
16.4 Common Questions (LSTM) 94
16.4.1 Q1: Why doesn't the cell state gradient vanish? 94
16.4.2 Q2: Are 4 gates really necessary? 94
16.4.3 Q3: Why initialize the Forget gate to 1? 94
16.4.4 Q4: Difference between Cell State and Hidden State? 94

17 Sequence-to-Sequence Models and Attention **95**
17.1 Encoder-Decoder Architecture . . . 95
17.1.1 Motivation . . . 95
17.1.2 Fixed-Length Context Vector . . . 95
17.2 Attention Mechanism . . . 95
17.2.1 The Bottleneck Problem . . . 95
17.2.2 Attention Weights . . . 95
17.3 Common Questions (Seq2Seq & Attention) . . . 96
17.3.1 Q1: Limitations of Fixed Context Vectors? . . . 96
17.3.2 Q2: What do Attention weights mean? . . . 96
17.3.3 Q3: Which Attention score is best? . . . 96
17.3.4 Q4: Why Multi-Head Attention? . . . 96

18 Transformer Architecture **97**
18.1 Self-Attention Mechanism . . . 97
18.1.1 Query, Key, Value Projection . . . 97
18.1.2 Scaled Dot-Product Attention . . . 97
18.1.3 Why Scale by $\sqrt{d_k}$? . . . 98
18.2 Multi-Head Attention . . . 98
18.2.1 Multiple Attention Heads . . . 98
18.3 Positional Encoding . . . 98
18.3.1 Sinusoidal Positional Encoding . . . 98
18.4 Transformer Encoder Block . . . 99
18.4.1 Block Structure . . . 99
18.4.2 Feed-Forward Network . . . 99
18.4.3 Layer Normalization . . . 99
18.5 Transformer Decoder Block . . . 99
18.5.1 Three Sub-layers . . . 99
18.5.2 Masked Attention . . . 99
18.5.3 Masked Attention . . . 99
18.6 Conceptual Intuition: The Conference Room Analogy . . . 100
18.6.1 1. Inputs: Participants and Seating . . . 100
18.6.2 2. Self-Attention: The Q-K-V Mechanism . . . 101
18.6.3 3. Multi-Head and Masking . . . 101
18.6.4 4. The Building Blocks . . . 101
18.7 Common Questions (Transformer) . . . 101
18.7.1 Q1: Why does Self-Attention solve the RNN problem? . . . 101
18.7.2 Q2: What is the difference between Query, Key, and Value? . . . 102
18.7.3 Q3: Why scale by $\sqrt{d_k}$? . . . 102
18.7.4 Q4: Sinusoidal vs. Learnable Positional Encoding? . . . 102
18.7.5 Q5: Layer Norm vs. Batch Norm? . . . 102
18.7.6 Q6: What is Masked Attention? . . . 103
18.7.7 Q7: Is Transformer really faster than RNN? . . . 103

19 Scaling Laws and Foundation Models **105**
19.1 Scaling Laws . . . 105
19.1.1 Empirical Observations . . . 105
19.2 In-Context Learning . . . 105

19.3 Common Questions (Foundation Models) . . . 105
19.3.1 Q1: Do Scaling Laws hold forever? . . . 105
19.3.2 Q2: Why do Emergent Abilities occur? . . . 106
19.3.3 Q3: How does In-Context Learning work? . . . 106
19.3.4 Q4: What makes a "Foundation" Model? . . . 106
19.4 Summary Table: Architecture Comparison . . . 106

20 Transformer Variants and Modern Architectures 107
20.1 Overview of Transformer Evolution . . . 107
20.1.1 Timeline and Motivation . . . 107
20.2 Encoder-Only: BERT and Variants . . . 107
20.2.1 BERT Architecture . . . 107
20.2.2 Masked Language Modeling (MLM) . . . 107
20.3 Decoder-Only: GPT and Variants . . . 108
20.3.1 GPT Architecture . . . 108
20.3.2 Causal Language Modeling . . . 108
20.3.3 Instruction Tuning and RLHF . . . 108
20.4 Encoder-Decoder: T5 and Variants . . . 108
20.4.1 T5: Unified Framework . . . 108
20.5 Sparse and Efficient Variants . . . 108
20.5.1 The $O(T^2)$ Problem . . . 108
20.5.2 Longformer & BigBird . . . 109
20.5.3 Mixture of Experts (MoE) . . . 109
20.6 Multimodal and Vision Transformers . . . 109
20.6.1 Vision Transformer (ViT) . . . 109
20.6.2 CLIP . . . 109
20.7 Recent Trends . . . 109
20.7.1 RAG (Retrieval-Augmented Generation) . . . 109
20.7.2 LoRA (Low-Rank Adaptation) . . . 109
20.8 Common Questions (Transformer Variants) . . . 109
20.8.1 Q1: Why is BERT bidirectional but GPT unidirectional? . . . 109
20.8.2 Q2: Does Masked LM leak data? . . . 110
20.8.3 Q3: Why do models suddenly get smarter with scale? . . . 110
20.8.4 Q4: What does Temperature do? . . . 110
20.8.5 Q5: Why is the KL penalty needed in RLHF? . . . 110
20.8.6 Q6: How can T5 unify all tasks? . . . 110
20.8.7 Q7: Does Sparse Attention lose information? . . . 110
20.8.8 Q8: Is Linear Transformer exactly equivalent? . . . 111
20.8.9 Q9: How to decide the number of Experts (MoE)? . . . 111
20.8.10 Q10: Why is ViT better than CNNs? . . . 111
20.8.11 Q11: RAG vs. Fine-tuning? . . . 111
20.8.12 Q12: Prefix Tuning vs. LoRA? . . . 111

21 BERT: Bidirectional Encoder with Masked Language Modeling 113
21.1 Architecture Overview . . . 113
21.1.1 Intuitive Understanding . . . 113
21.2 Notation and Input Representation . . . 114
21.3 Encoder Architecture: Bidirectional Self-Attention . . . 115

21.3.1 Multi-Head Self-Attention . . . 115
21.3.2 Residual Connection and Layer Normalization . . . 115
21.3.3 Position-Wise Feed-Forward Network . . . 116
21.4 Pretraining Objective I: Masked Language Modeling . . . 116
21.4.1 Masking Strategy . . . 116
21.4.2 Token-Level Logits and Probabilities . . . 116
21.4.3 Gradient w.r.t. Logits . . . 117
21.5 Pretraining Objective II: Next Sentence Prediction . . . 117
21.5.1 Pair Representation . . . 117
21.5.2 Joint Loss . . . 117
21.6 Batching, Masks, and Complexity . . . 118
21.6.1 Attention Mask Matrix . . . 118
21.6.2 Computational Cost . . . 118
21.7 Summary . . . 118

22 GPT: Decoder-Only Autoregressive Transformer 119
22.1 Architecture Overview . . . 119
22.1.1 Intuitive Understanding . . . 119
22.2 Notation and Factorization of the Language Model . . . 120
22.3 Token, Position, and Input Embeddings . . . 120
22.4 Causal Multi-Head Self-Attention . . . 121
22.4.1 Single Head Self-Attention with Causal Mask . . . 121
22.4.2 Multi-Head Attention and Output Projection . . . 122
22.4.3 Residual and Pre/Post-Norm Variants . . . 122
22.5 Position-Wise Feed-Forward Network . . . 122
22.6 Output Layer and Conditional Distribution . . . 123
22.7 Training Objective and Gradients . . . 123
22.7.1 Sequence Loss and Per-Token Loss . . . 123
22.7.2 Gradient w.r.t. Logits and Hidden States . . . 123
22.8 Teacher Forcing and Inference . . . 124
22.8.1 Teacher Forcing During Training . . . 124
22.8.2 Autoregressive Generation at Test Time . . . 124
22.9 Perplexity and Evaluation Metric . . . 124
22.10 Batching, Causal Mask, and Complexity . . . 125
22.10.1 Batch-Wise Causal Attention . . . 125
22.10.2 Computational Complexity . . . 125
22.11 Summary . . . 126

23 RoBERTa: Robustly Optimized BERT Pretraining 127
23.1 Architectural Overview . . . 127
23.2 Token and Segment Representation . . . 127
23.3 Bidirectional Self-Attention Encoder . . . 128
23.3.1 Multi-Head Self-Attention (Unmasked) . . . 128
23.3.2 Pre-LN Encoder Block . . . 128
23.4 Masked Language Modeling with Dynamic Masking . . . 128
23.4.1 Masking Strategy as a Random Process . . . 128
23.4.2 MLM Objective as Conditional Likelihood . . . 129
23.4.3 Per-Token Cross-Entropy and Gradients . . . 129

23.5 Dynamic vs. Static Masking: Distributional View . . . 129
23.6 Pretraining Objective and Optimization . . . 129

24 Longformer: Efficient Long-Document Transformer 131
24.1 Motivation and the $O(T^2)$ Bottleneck . . . 131
24.2 Attention Mask and Sparsity Pattern . . . 131
24.2.1 Full Attention Recap . . . 131
24.2.2 Sparse Attention as a Structured Mask . . . 131
24.3 Sliding Window Attention . . . 131
24.3.1 Definition of Window Size w . . . 131
24.3.2 Multi-Layer Reception Field . . . 132
24.4 Dilated Sliding Window (Optional) . . . 132
24.5 Global Attention . . . 132
24.5.1 Global Token Set $\mathcal{G} \subset \{1, \ldots, T\}$ . . . 132
24.5.2 Global Token Selection Strategies . . . 132
24.6 Computational Complexity Analysis . . . 132
24.6.1 Per-Layer Complexity . . . 132
24.6.2 Total Model Complexity . . . 132
24.7 Formal Attention Definition in Longformer . . . 133
24.7.1 Score and Mask Computation . . . 133
24.7.2 Sparse Softmax . . . 133
24.8 Gradient Flow Through Sparse Attention . . . 133
24.9 Comparison with Other Efficient Transformers . . . 133
24.9.1 BigBird . . . 133
24.9.2 Linformer / Performer . . . 133
24.10Summary . . . 133

25 T5: Text-to-Text Transfer Transformer 135
25.1 Architecture Overview . . . 135
25.1.1 Intuitive Understanding . . . 135
25.2 Unified Text-to-Text Framework . . . 136
25.2.1 Task Formulation as Conditional Generation . . . 136
25.2.2 Task Prefix and Examples . . . 136
25.3 Encoder-Decoder Architecture . . . 136
25.3.1 Encoder: Bidirectional Self-Attention . . . 136
25.3.2 Decoder: Causal Self-Attention + Cross-Attention . . . 137
25.3.3 Masked Self-Attention in Decoder . . . 137
25.3.4 Cross-Attention to Encoder . . . 137
25.3.5 Decoder Block Composition . . . 137
25.4 Relative Position Bias . . . 137
25.4.1 Bias Definition . . . 137
25.4.2 Bucketed Relative Position . . . 138
25.5 Pretraining Objective: Span Corruption . . . 138
25.5.1 Span Masking as a Random Process . . . 138
25.5.2 Target Sequence Construction . . . 138
25.5.3 Denoising Objective as Conditional Likelihood . . . 138
25.5.4 Per-Token Loss and Gradient . . . 138
25.6 Teacher Forcing and Autoregressive Decoding . . . 138

25.6.1 Training with Teacher Forcing . . . 138
25.6.2 Inference with Autoregressive Generation . . . 139
25.7 Simplified Layer Normalization (RMSNorm) . . . 139
25.8 Computational Complexity . . . 139
25.8.1 Encoder Complexity . . . 139
25.8.2 Decoder Complexity . . . 139
25.9 Training Strategies and Hyperparameters . . . 139
25.9.1 Pre-Training Corpus: C4 . . . 139
25.9.2 Model Sizes . . . 139
25.10 Comparison with BERT and GPT . . . 139
25.11 Summary . . . 140

26 Vision Transformer (ViT): Transformers for Image Classification 141
26.1 Architecture Overview . . . 141
26.1.1 Intuitive Understanding . . . 142
26.2 Motivation: From Convolution to Self-Attention . . . 142
26.3 Image as a Sequence: Patch Embedding . . . 142
26.3.1 Image Partitioning into Patches . . . 142
26.3.2 Patch Extraction as Tensor Reshaping . . . 142
26.3.3 Linear Projection to Embedding Dimension . . . 142
26.4 Prepending the Class Token . . . 143
26.5 Positional Encoding . . . 143
26.5.1 Learnable 1D Position Embedding . . . 143
26.5.2 Input Embedding Composition . . . 143
26.6 Transformer Encoder . . . 143
26.6.1 Multi-Head Self-Attention . . . 143
26.6.2 Layer Normalization and Residual Connections . . . 143
26.6.3 Position-Wise Feed-Forward Network . . . 143
26.7 Classification Head . . . 144
26.7.1 Extracting the CLS Token . . . 144
26.7.2 Linear Classification Layer . . . 144
26.7.3 Cross-Entropy Loss . . . 144
26.8 Pre-Training and Transfer Learning . . . 144
26.8.1 Pre-Training on Large Datasets . . . 144
26.8.2 Fine-Tuning on Target Datasets . . . 144
26.9 Model Variants and Scaling . . . 144
26.10 Computational Complexity . . . 145
26.10.1 Self-Attention Complexity . . . 145
26.10.2 Comparison with CNN . . . 145
26.11 Inductive Bias and Data Efficiency . . . 145
26.12 Extensions and Variants . . . 145
26.12.1 DeiT (Data-efficient image Transformer) . . . 145
26.12.2 Swin Transformer . . . 145
26.12.3 BEiT . . . 145
26.13 Summary . . . 145

27 PaLM: Pathways Language Model **147**
27.1 Overview and Scaling Philosophy . . . 147
27.2 Autoregressive Language Modeling . . . 147
27.3 Parallel Layers Architecture . . . 147
27.4 Multi-Query Attention . . . 148
27.5 RoPE: Rotary Position Embedding . . . 148
27.6 SwiGLU Activation . . . 148
27.7 Model Configurations . . . 148
27.8 Summary . . . 148

28 LLaMA: Large Language Model Meta AI **149**
28.1 Overview and Design Philosophy . . . 149
28.2 RMSNorm: Root Mean Square Layer Normalization . . . 149
28.3 Pre-Normalization Architecture . . . 150
28.4 Causal Self-Attention with RoPE . . . 150
28.5 SwiGLU Feed-Forward Network . . . 150
28.6 Model Configurations . . . 150
28.7 Training Data . . . 150
28.8 LLaMA 2 Improvements . . . 150
28.9 Summary . . . 151

29 Mixtral: Sparse Mixture of Experts Language Model **153**
29.1 Architecture Overview . . . 153
29.1.1 Intuitive Understanding . . . 153
29.2 Sparse MoE Architecture Details . . . 154
29.3 Mixture of Experts Formulation . . . 154
29.4 Top-k Routing . . . 154
29.5 Load Balancing Loss . . . 155
29.6 Sliding Window Attention . . . 155
29.7 Parameter Count . . . 155
29.8 Comparison with Dense Models . . . 155
29.9 Summary . . . 155

30 Complete Backpropagation Walkthroughs: RNN to Transformer **157**
30.1 Part I: Simple RNN - Complete Backpropagation Example . . . 157
30.1.1 Setup: 2-Layer RNN with Concrete Numbers . . . 157
30.1.2 Forward Propagation . . . 158
30.1.3 Loss Calculation . . . 158
30.1.4 Backward Propagation Through Time (BPTT) . . . 158
30.2 Part II: LSTM - Complete Backpropagation Example . . . 159
30.2.1 Setup . . . 159
30.2.2 Forward Pass (Abstract) . . . 159
30.2.3 Backward Pass Logic . . . 159
30.3 Part III: Transformer - Complete Backpropagation Example . . . 160
30.3.1 Setup: Minimal Attention . . . 160
30.3.2 Forward: Multi-Head Attention . . . 160
30.3.3 Backward: Gradients . . . 160
30.4 Summary of Backprop Complexity . . . 160

Chapter 1

Mathematical Preliminaries

1.1 Vectors and Matrices

1.1.1 Vector Spaces and Norms

A vector $\mathbf{v} \in \mathbb{R}^n$ is an ordered list of n real numbers:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \tag{1.1}$$

The Euclidean norm (or L^2 norm) of a vector is defined as:

$$\|\mathbf{v}\|_2 = \sqrt{\sum_{i=1}^{n} v_i^2} = \sqrt{\mathbf{v}^T\mathbf{v}} \tag{1.2}$$

More generally, the L^p norm is:

$$\|\mathbf{v}\|_p = \left(\sum_{i=1}^{n} |v_i|^p\right)^{1/p} \tag{1.3}$$

For $p = 1$ (Manhattan distance):

$$\|\mathbf{v}\|_1 = \sum_{i=1}^{n} |v_i| \tag{1.4}$$

For $p = \infty$ (maximum absolute value):

$$\|\mathbf{v}\|_\infty = \max_i |v_i| \tag{1.5}$$

1.1.2 Dot Product and Geometric Interpretation

The dot product (inner product) of two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^n$ is:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n = \mathbf{a}^T\mathbf{b} \tag{1.6}$$

Geometric interpretation: The dot product relates to the angle θ between vectors via:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|\|\mathbf{b}\| \cos\theta \tag{1.7}$$

This implies:

$$\cos\theta = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|} \tag{1.8}$$

Key observations:

- When $\theta = 0$ (parallel): $\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|\|\mathbf{b}\|$ (maximum)
- When $\theta = 90$ (orthogonal): $\mathbf{a} \cdot \mathbf{b} = 0$
- When $\theta = 180$ (anti-parallel): $\mathbf{a} \cdot \mathbf{b} = -\|\mathbf{a}\|\|\mathbf{b}\|$ (minimum)

In neural networks, the dot product $\mathbf{w} \cdot \mathbf{x}$ measures the alignment between weights and inputs. Large alignment (small angle) produces large activations.

1.1.3 Matrix Operations

A matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ has m rows and n columns:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \tag{1.9}$$

Matrix-vector multiplication: If $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{x} \in \mathbb{R}^n$, then $\mathbf{y} = \mathbf{A}\mathbf{x} \in \mathbb{R}^m$ where:

$$y_i = \sum_{j=1}^{n} a_{ij} x_j = \mathbf{a}_i^T \mathbf{x} \tag{1.10}$$

where $\mathbf{a}_i$ is the i-th row of $\mathbf{A}$. Notice that each output y_i is the dot product of row i with $\mathbf{x}$.

Matrix-matrix multiplication: If $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{n \times p}$, then $\mathbf{C} = \mathbf{A}\mathbf{B} \in \mathbb{R}^{m \times p}$ where:

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} = \mathbf{a}_i^T \mathbf{b}_j \tag{1.11}$$

where $\mathbf{a}_i$ is row i of $\mathbf{A}$ and $\mathbf{b}_j$ is column j of $\mathbf{B}$. Thus, the element at position (i, j) is the dot product of row i of $\mathbf{A}$ with column j of $\mathbf{B}$.

1.1.4 Important Matrix Properties

Transpose: Swapping rows and columns. If $\mathbf{A} \in \mathbb{R}^{m \times n}$, then $\mathbf{A}^T \in \mathbb{R}^{n \times m}$ with:

$$(\mathbf{A}^T)_{ij} = a_{ji} \tag{1.12}$$

Properties of transpose:

$$(\mathbf{A}\mathbf{B})^T = \mathbf{B}^T \mathbf{A}^T \quad \text{(reverse order!)} \tag{1.13}$$

Frobenius norm: For matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$:

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n} a_{ij}^2} = \sqrt{\text{trace}(\mathbf{A}^T\mathbf{A})} \tag{1.14}$$

1.2 Calculus: Foundations of Optimization

1.2.1 Derivatives and the Chain Rule

For a univariate function $f : \mathbb{R} \to \mathbb{R}$, the derivative at point x is:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{1.15}$$

Geometrically, $f'(x)$ is the slope of the tangent line to f at x.

Chain Rule: If $y = f(u)$ and $u = g(x)$, then:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx} \tag{1.16}$$

More generally, for compositions $y = f(g(h(x)))$:

$$\frac{dy}{dx} = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{dx} \tag{1.17}$$

Example: Let $y = \sin(x^2)$. Set $u = x^2$, so $y = \sin(u)$.

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx} = \cos(u) \cdot 2x = 2x\cos(x^2) \tag{1.18}$$

1.2.2 Partial Derivatives and Gradients

For a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$, the partial derivative with respect to variable x_i is:

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_i + h, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h} \tag{1.19}$$

The gradient is the vector of all partial derivatives:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} \tag{1.20}$$

Directional derivative: The rate of change of f in direction $\mathbf{d}$ (unit vector) is:

$$\nabla_{\mathbf{d}} f = \mathbf{d}^T \nabla f = \nabla f \cdot \mathbf{d} \tag{1.21}$$

The gradient ∇f points in the direction of steepest ascent. Negative gradient $-\nabla f$ points in the direction of steepest descent.

Example: Let $f(x, y) = x^2 + xy + 3y^2$.

$$\frac{\partial f}{\partial x} = 2x + y, \quad \frac{\partial f}{\partial y} = x + 6y \tag{1.22}$$

$$\nabla f = \begin{bmatrix} 2x + y \\ x + 6y \end{bmatrix} \tag{1.23}$$

At point $(x, y) = (1, 2)$:

$$\nabla f|_{(1,2)} = \begin{bmatrix} 2(1) + 2 \\ 1 + 6(2) \end{bmatrix} = \begin{bmatrix} 4 \\ 13 \end{bmatrix} \tag{1.24}$$

1.2.3 Matrix Calculus

For a scalar function $f(\mathbf{A})$ that depends on a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, the gradient (or derivative) is a matrix:

$$\frac{\partial f}{\partial \mathbf{A}} = \begin{bmatrix} \frac{\partial f}{\partial a_{11}} & \frac{\partial f}{\partial a_{12}} & \cdots \\ \frac{\partial f}{\partial a_{21}} & \frac{\partial f}{\partial a_{22}} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} \tag{1.25}$$

Key identities for matrix derivatives:

1. Linear function: $f(\mathbf{x}) = \mathbf{a}^T\mathbf{x}$, then $\frac{\partial f}{\partial \mathbf{x}} = \mathbf{a}$
2. Quadratic form: $f(\mathbf{x}) = \mathbf{x}^T\mathbf{A}\mathbf{x}$, then $\frac{\partial f}{\partial \mathbf{x}} = (\mathbf{A} + \mathbf{A}^T)\mathbf{x} = 2\mathbf{A}\mathbf{x}$ (if $\mathbf{A}$ is symmetric)
3. Matrix product: $f(\mathbf{A}) = \text{trace}(\mathbf{A}^T\mathbf{B})$, then $\frac{\partial f}{\partial \mathbf{A}} = \mathbf{B}$
4. For loss $\mathcal{L}(\mathbf{x}) = \frac{1}{2}\|\mathbf{y} - \mathbf{W}\mathbf{x}\|_2^2$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}} = (\mathbf{W}\mathbf{x} - \mathbf{y})\mathbf{x}^T \tag{1.26}$$

Notation, Dimensions, and Label Conventions

This section fixes the main symbols, tensor shapes, and label conventions used throughout the notes to avoid ambiguity across chapters.

Basic symbols and sets

- Scalars are denoted by lowercase letters (e.g., $x \in \mathbb{R}$), vectors by bold lowercase (e.g., $\mathbf{x} \in \mathbb{R}^d$), and matrices by uppercase (e.g., $A \in \mathbb{R}^{m \times n}$).
- The Euclidean norm is $\|\mathbf{v}\|_2$ and the Frobenius norm is $\|A\|_F$.
- The all-ones vector of length m is $\mathbf{1} \in \mathbb{R}^m$.

Data, mini-batches, and shapes

Let the input dimension be d, the number of classes be K, and the dataset size be N.

- Single example: $(\mathbf{x}, \mathbf{y})$ with $\mathbf{x} \in \mathbb{R}^d$.
- Mini-batch of size m (column-stacked convention):

$$X = \begin{bmatrix} \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \end{bmatrix} \in \mathbb{R}^{d \times m}.$$

  This matches the vectorized forward form used in the network chapters.

Network architecture and parameters

Consider an L-layer feedforward network with layer widths

$$n_0 = d, \quad n_1, \ldots, n_{L-1}, \quad n_L = \begin{cases} 1 & \text{binary output (sigmoid)} \\ K & K\text{-class output (softmax).} \end{cases}$$

The parameters are $\theta = \{W^{(\ell)}, \mathbf{b}^{(\ell)}\}_{\ell=1}^{L}$.

For each layer $\ell = 1, \ldots, L$:

$$W^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}, \qquad \mathbf{b}^{(\ell)} \in \mathbb{R}^{n_\ell}.$$

Forward pass (single example):

$$\mathbf{a}^{(0)} = \mathbf{x}, \qquad \mathbf{z}^{(\ell)} = W^{(\ell)}\mathbf{a}^{(\ell-1)} + \mathbf{b}^{(\ell)}, \qquad \mathbf{a}^{(\ell)} = \sigma^{(\ell)}(\mathbf{z}^{(\ell)}).$$

Here $\mathbf{z}^{(\ell)}, \mathbf{a}^{(\ell)} \in \mathbb{R}^{n_\ell}$.

Vectorized mini-batch forward pass (column-stacked):

$$A^{(0)} = X \in \mathbb{R}^{n_0 \times m}, \qquad Z^{(\ell)} = W^{(\ell)}A^{(\ell-1)} + \mathbf{b}^{(\ell)}\mathbf{1}^\top \in \mathbb{R}^{n_\ell \times m}, \qquad A^{(\ell)} = \sigma^{(\ell)}(Z^{(\ell)}).$$

This is the same convention used in the notes' fully-vectorized section.

Backpropagation symbols

The loss for one example is denoted by $L(\hat{\mathbf{y}}, \mathbf{y})$.

The key backpropagation quantity is the delta:

$$\boldsymbol{\delta}^{(\ell)} \;=\; \frac{\partial L}{\partial \mathbf{z}^{(\ell)}} \in \mathbb{R}^{n_\ell}.$$

With this convention, gradients take the outer-product form (single example):

$$\frac{\partial L}{\partial W^{(\ell)}} = \boldsymbol{\delta}^{(\ell)}\big(\mathbf{a}^{(\ell-1)}\big)^\top, \qquad \frac{\partial L}{\partial \mathbf{b}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)}.$$

These formulas match the derivations in the backpropagation chapter.

For a mini-batch of size m, define the averaged objective (empirical risk on the batch)

$$\mathcal{L}_{\text{batch}}(\theta) = \frac{1}{m}\sum_{i=1}^{m} L\big(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}\big),$$

and compute averaged gradients accordingly (as in the mini-batch gradient section).

Label conventions (important)

Binary classification appears in two common encodings:

- **Probability/BCE convention:** $y \in \{0, 1\}$, $\hat{y} \in (0, 1)$ and $\hat{y} = \sigma(z)$ (sigmoid). This is the convention used in the BCE chapter and the worked numerical example.
- **Perceptron convention:** $y \in \{-1, +1\}$ and prediction $\hat{y} = \text{sign}(w^\top x + b)$, used for the perceptron convergence theorem statement.

A simple conversion between the two binary encodings is

$$y_{\pm 1} = 2y_{01} - 1, \qquad y_{01} = \frac{y_{\pm 1} + 1}{2}.$$

Use $y \in \{-1, +1\}$ when discussing the classic perceptron theorem, and $y \in \{0, 1\}$ when using sigmoid/BCE.

Multi-class classification uses one-hot vectors:

$$\mathbf{y} \in \{0, 1\}^K, \qquad \sum_{k=1}^{K} y_k = 1, \qquad \mathbf{p} = \text{softmax}(\mathbf{z}) \in (0, 1)^K, \qquad \sum_{k=1}^{K} p_k = 1.$$

This matches the softmax + categorical cross-entropy setup and its gradient simplification.

Common nonlinearities (quick reference)

- Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$, $\sigma'(z) = \sigma(z)(1-\sigma(z))$.
- ReLU: $\mathrm{ReLU}(z) = \max(0, z)$ with subgradient $\mathrm{ReLU}'(z) = 1$ for $z > 0$, 0 for $z < 0$, and any value in $[0, 1]$ at $z = 0$.
- Softmax: $p_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$, Jacobian $\frac{\partial p_i}{\partial z_j} = p_i(\delta_{ij} - p_j)$.

Chapter 2

The Perceptron

2.1 Decision Boundary as a Hyperplane

The perceptron is a binary linear classifier. Given an input vector $\mathbf{x} \in \mathbb{R}^d$, weights $\mathbf{w} \in \mathbb{R}^d$, and bias $b \in \mathbb{R}$, define the *score*

$$z(\mathbf{x}) \;=\; \mathbf{w}^\top \mathbf{x} + b. \tag{2.1}$$

The perceptron prediction is

$$\hat{y}(\mathbf{x}) \;=\; \mathrm{step}(z(\mathbf{x})) \;=\; \begin{cases} 1 & \text{if } \mathbf{w}^\top \mathbf{x} + b \geq 0, \\ 0 & \text{if } \mathbf{w}^\top \mathbf{x} + b < 0. \end{cases} \tag{2.2}$$

(Equivalently, using labels $y \in \{-1, +1\}$ one often writes $\hat{y} = \mathrm{sign}(\mathbf{w}^\top \mathbf{x} + b)$.)

2.1.1 Hyperplane interpretation

The *decision boundary* is the set of points where the score is exactly zero:

$$\mathcal{H} \;=\; \{\mathbf{x} \in \mathbb{R}^d \,:\, \mathbf{w}^\top \mathbf{x} + b = 0\}. \tag{2.3}$$

This set $\mathcal{H}$ is a **hyperplane** in $\mathbb{R}^d$ with normal vector $\mathbf{w}$.

The hyperplane splits $\mathbb{R}^d$ into two half-spaces:

$$\mathcal{H}_+ = \{\mathbf{x} \in \mathbb{R}^d \,:\, \mathbf{w}^\top \mathbf{x} + b > 0\}, \tag{2.4}$$

$$\mathcal{H}_- = \{\mathbf{x} \in \mathbb{R}^d \,:\, \mathbf{w}^\top \mathbf{x} + b < 0\}. \tag{2.5}$$

Geometrically, $\mathbf{w}$ points toward the side classified as positive.

2.1.2 Scaling invariance

For any $\alpha > 0$, replacing $(\mathbf{w}, b)$ by $(\alpha\mathbf{w}, \alpha b)$ leaves the boundary unchanged:

$$(\alpha\mathbf{w})^\top \mathbf{x} + \alpha b = \alpha(\mathbf{w}^\top \mathbf{x} + b), \tag{2.6}$$

so $\mathbf{w}^\top \mathbf{x} + b = 0$ iff $(\alpha\mathbf{w})^\top \mathbf{x} + \alpha b = 0$. Only the *direction* of $\mathbf{w}$ and the *relative offset* b matter for classification.

2.2 Signed Distance to the Hyperplane

Let $\mathcal{H} = \{\mathbf{x} : \mathbf{w}^\top \mathbf{x} + b = 0\}$ with $\mathbf{w} \neq \mathbf{0}$. The **signed distance** from a point $\mathbf{x}$ to the hyperplane is

$$d(\mathbf{x}, \mathcal{H}) \;=\; \frac{\mathbf{w}^\top \mathbf{x} + b}{\|\mathbf{w}\|_2}. \tag{2.7}$$

2.2.1 Derivation

Pick any point $\mathbf{x}_0 \in \mathcal{H}$ so that $\mathbf{w}^\top \mathbf{x}_0 + b = 0$. The vector from $\mathbf{x}_0$ to $\mathbf{x}$ is $\mathbf{x} - \mathbf{x}_0$. Projecting this vector onto the unit normal direction $\mathbf{u} = \mathbf{w}/\|\mathbf{w}\|_2$ gives the signed distance:

$$d(\mathbf{x}, \mathcal{H}) = \mathbf{u}^\top (\mathbf{x} - \mathbf{x}_0) = \frac{\mathbf{w}^\top (\mathbf{x} - \mathbf{x}_0)}{\|\mathbf{w}\|_2} = \frac{\mathbf{w}^\top \mathbf{x} - \mathbf{w}^\top \mathbf{x}_0}{\|\mathbf{w}\|_2}. \tag{2.8}$$

Using $\mathbf{w}^\top \mathbf{x}_0 = -b$ yields (2.7). In particular:

- $d(\mathbf{x}, \mathcal{H}) > 0$ iff $\mathbf{x} \in \mathcal{H}_+$,
- $d(\mathbf{x}, \mathcal{H}) < 0$ iff $\mathbf{x} \in \mathcal{H}_-$,
- $|d(\mathbf{x}, \mathcal{H})|$ equals the Euclidean distance to the boundary.

2.3 Perceptron Learning Algorithm

2.3.1 Label convention

For the convergence theorem, it is standard to use labels $y_i \in \{-1, +1\}$. Given training data $\{(\mathbf{x}_i, y_i)\}_{i=1}^m$, define the prediction

$$\hat{y}_i = \operatorname{sign}(\mathbf{w}^\top \mathbf{x}_i + b). \tag{2.9}$$

2.3.2 Update rule

Initialize $\mathbf{w}_0 = \mathbf{0}$ and $b_0 = 0$. At iteration t, pick a misclassified example $(\mathbf{x}_i, y_i)$ such that

$$y_i(\mathbf{w}_t^\top \mathbf{x}_i + b_t) \leq 0. \tag{2.10}$$

Then apply the perceptron update:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \eta\, y_i\, \mathbf{x}_i, \tag{2.11}$$

$$b_{t+1} = b_t + \eta\, y_i, \tag{2.12}$$

where $\eta > 0$ is the learning rate. (Equivalently, one may absorb the bias into an augmented vector $\tilde{\mathbf{x}} = [\mathbf{x}^\top, 1]^\top$ and weight $\tilde{\mathbf{w}} = [\mathbf{w}^\top, b]^\top$ to write a single update.)

2.4 Perceptron Convergence Theorem (Sketch)

2.4.1 Linear separability and margin

Assume the data is linearly separable: there exists $(\mathbf{w}_\star, b_\star)$ such that

$$y_i(\mathbf{w}_\star^\top \mathbf{x}_i + b_\star) \geq 1 \quad \text{for all } i. \tag{2.13}$$

Let $R = \max_i \|\mathbf{x}_i\|_2$. Define the (geometric) margin of the separator $(\mathbf{w}_\star, b_\star)$ as

$$\gamma \;=\; \min_i \frac{y_i(\mathbf{w}_\star^\top \mathbf{x}_i + b_\star)}{\|\mathbf{w}_\star\|_2}. \tag{2.14}$$

Under (2.13), one has $\gamma \geq 1/\|\mathbf{w}_\star\|_2$.

2.4.2 Theorem

Theorem (Perceptron convergence). If the training set is linearly separable with margin $\gamma > 0$ and $\|\mathbf{x}_i\| \leq R$, then the perceptron makes at most

$$M \;\leq\; \left(\frac{R}{\gamma}\right)^2 \tag{2.15}$$

mistakes (updates), hence it terminates after finitely many updates.

2.4.3 Proof sketch

For simplicity take $\eta = 1$ and use augmented vectors to include b (the same argument works without augmentation). Let $\mathbf{w}_t$ denote the weight after t updates.

Step 1: Progress along the optimal separator. For an update using misclassified $(\mathbf{x}_i, y_i)$:

$$\mathbf{w}_{t+1}^\top \mathbf{w}_\star = (\mathbf{w}_t + y_i \mathbf{x}_i)^\top \mathbf{w}_\star = \mathbf{w}_t^\top \mathbf{w}_\star + y_i\, \mathbf{x}_i^\top \mathbf{w}_\star. \tag{2.16}$$

By separability, $y_i\, \mathbf{x}_i^\top \mathbf{w}_\star \geq \gamma \|\mathbf{w}_\star\|_2$ (up to the bias/augmentation convention), so after M mistakes:

$$\mathbf{w}_M^\top \mathbf{w}_\star \;\geq\; M\, \gamma\, \|\mathbf{w}_\star\|_2. \tag{2.17}$$

Step 2: Norm growth is controlled. The squared norm evolves as

$$\|\mathbf{w}_{t+1}\|_2^2 = \|\mathbf{w}_t + y_i \mathbf{x}_i\|_2^2 = \|\mathbf{w}_t\|_2^2 + 2 y_i\, \mathbf{w}_t^\top \mathbf{x}_i + \|\mathbf{x}_i\|_2^2. \tag{2.18}$$

Because the point is misclassified, $y_i(\mathbf{w}_t^\top \mathbf{x}_i) \leq 0$, hence

$$\|\mathbf{w}_{t+1}\|_2^2 \leq \|\mathbf{w}_t\|_2^2 + \|\mathbf{x}_i\|_2^2 \leq \|\mathbf{w}_t\|_2^2 + R^2. \tag{2.19}$$

By induction,

$$\|\mathbf{w}_M\|_2^2 \leq M R^2 \quad \Rightarrow \quad \|\mathbf{w}_M\|_2 \leq R\sqrt{M}. \tag{2.20}$$

Step 3: Combine via Cauchy–Schwarz. By Cauchy–Schwarz,

$$\mathbf{w}_M^\top \mathbf{w}_\star \le \|\mathbf{w}_M\|_2 \, \|\mathbf{w}_\star\|_2. \tag{2.21}$$

Plugging (2.17) and (2.20):

$$M\,\gamma\,\|\mathbf{w}_\star\|_2 \le \|\mathbf{w}_M\|_2 \, \|\mathbf{w}_\star\|_2 \le R\sqrt{M}\,\|\mathbf{w}_\star\|_2. \tag{2.22}$$

Cancel $\|\mathbf{w}_\star\|_2 > 0$ and rearrange:

$$M\gamma \le R\sqrt{M} \quad \Rightarrow \quad \sqrt{M} \le \frac{R}{\gamma} \quad \Rightarrow \quad M \le \left(\frac{R}{\gamma}\right)^2, \tag{2.23}$$

which proves (2.15).

2.4.4 Remarks

- If the data is *not* linearly separable, the perceptron update may cycle and never converge.
- The bound depends on R (data scale) and γ (separability margin). Larger margins imply fewer updates.

Chapter 3

Feedforward Neural Networks

3.1 From Scalar Sums to Matrix Form

A feedforward neural network generalizes the perceptron by stacking multiple affine maps and nonlinearities.

3.1.1 Single neuron (scalar form)

Let $\mathbf{x} \in \mathbb{R}^d$ be an input, weights $\mathbf{w} \in \mathbb{R}^d$, bias $b \in \mathbb{R}$. A single neuron computes

$$z = \sum_{i=1}^{d} w_i x_i + b = \mathbf{w}^\top \mathbf{x} + b, \tag{3.1}$$

then outputs an activation $a = \sigma(z)$ for some nonlinearity σ.

3.1.2 Layer of neurons (summation index notation)

Consider layer ℓ with $n_{\ell-1}$ inputs and n_ℓ neurons. Let $\mathbf{a}^{(\ell-1)} \in \mathbb{R}^{n_{\ell-1}}$ be the input activation vector to layer ℓ. For neuron $j \in \{1, \dots, n_\ell\}$, define weights $W_{jk}^{(\ell)}$ from input unit k to neuron j and bias $b_j^{(\ell)}$. Then

$$z_j^{(\ell)} = \sum_{k=1}^{n_{\ell-1}} W_{jk}^{(\ell)} \, a_k^{(\ell-1)} + b_j^{(\ell)}. \tag{3.2}$$

3.1.3 Layer of neurons (matrix form)

Collect all $z_j^{(\ell)}$ into a vector $\mathbf{z}^{(\ell)} \in \mathbb{R}^{n_\ell}$, and define

$$\mathbf{W}^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}, \quad \mathbf{b}^{(\ell)} \in \mathbb{R}^{n_\ell}, \quad \mathbf{a}^{(\ell-1)} \in \mathbb{R}^{n_{\ell-1}}. \tag{3.3}$$

Then (3.2) becomes the compact vector form:

$$\mathbf{z}^{(\ell)} = \mathbf{W}^{(\ell)} \mathbf{a}^{(\ell-1)} + \mathbf{b}^{(\ell)}. \tag{3.4}$$

Applying an activation function element-wise,

$$\mathbf{a}^{(\ell)} = \sigma\big(\mathbf{z}^{(\ell)}\big). \tag{3.5}$$

3.1.4 Mini-batch (fully vectorized) form

For a mini-batch of size m, stack inputs as a matrix

$$\mathbf{A}^{(\ell-1)} = \begin{bmatrix} \mathbf{a}_1^{(\ell-1)} & \cdots & \mathbf{a}_m^{(\ell-1)} \end{bmatrix} \in \mathbb{R}^{n_{\ell-1} \times m}. \tag{3.6}$$

Then the affine map becomes

$$\mathbf{Z}^{(\ell)} = \mathbf{W}^{(\ell)} \mathbf{A}^{(\ell-1)} + \mathbf{b}^{(\ell)} \mathbf{1}^\top \in \mathbb{R}^{n_\ell \times m}, \tag{3.7}$$

where $\mathbf{1} \in \mathbb{R}^m$ is the all-ones vector. Activations:

$$\mathbf{A}^{(\ell)} = \sigma\big(\mathbf{Z}^{(\ell)}\big). \tag{3.8}$$

This matrix form is the basis of efficient GPU computation.

3.2 Network as Function Composition

Let the input layer be $\mathbf{a}^{(0)} = \mathbf{x} \in \mathbb{R}^{n_0}$. A depth-$L$ feedforward network is defined recursively by (3.4)–(3.5) for $\ell = 1, \ldots, L$.

The network output is

$$\hat{\mathbf{y}} = f(\mathbf{x}; \theta) = \mathbf{a}^{(L)}, \tag{3.9}$$

where $\theta = \{\mathbf{W}^{(\ell)}, \mathbf{b}^{(\ell)}\}_{\ell=1}^{L}$ is the set of parameters.

Writing out the full composition:

$$f(\mathbf{x}; \theta) = \sigma^{(L)}\Big(\mathbf{W}^{(L)}\, \sigma^{(L-1)}\big(\mathbf{W}^{(L-1)} \cdots \sigma^{(1)}(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \cdots + \mathbf{b}^{(L-1)}\big) + \mathbf{b}^{(L)}\Big), \tag{3.10}$$

where $\sigma^{(\ell)}$ may differ by layer (e.g., ReLU in hidden layers and sigmoid/softmax at the output).

3.2.1 Parameter count

The number of trainable parameters is

$$\#\theta = \sum_{\ell=1}^{L} (n_\ell n_{\ell-1} + n_\ell) = \sum_{\ell=1}^{L} n_\ell (n_{\ell-1} + 1). \tag{3.11}$$

Large n_ℓ or large depth L increases capacity but also risks overfitting without regularization.

3.3 Activation Functions and Derivatives

Nonlinear activations are essential: without them, the entire network collapses to a single affine map.

3.3.1 Why nonlinearity is required

If $\sigma(z) = z$ for all layers (purely linear network), then

$$\mathbf{a}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{a}^{(\ell-1)} + \mathbf{b}^{(\ell)}. \tag{3.12}$$

By induction, the entire network becomes

$$f(\mathbf{x}) = \mathbf{W}_{\text{eff}}\mathbf{x} + \mathbf{b}_{\text{eff}}, \tag{3.13}$$

for some effective matrix/vector $(\mathbf{W}_{\text{eff}}, \mathbf{b}_{\text{eff}})$, hence depth gives no additional expressive power. Therefore, σ must be nonlinear.

3.3.2 Sigmoid

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{3.14}$$

Derivative:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)). \tag{3.15}$$

3.3.3 Hyperbolic tangent

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}. \tag{3.16}$$

Derivative:

$$\frac{d}{dz}\tanh(z) = 1 - \tanh^2(z). \tag{3.17}$$

3.3.4 ReLU

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & (z > 0), \\ 0 & (z \leq 0). \end{cases} \tag{3.18}$$

Subgradient (used in practice):

$$\text{ReLU}'(z) = \begin{cases} 1 & (z > 0), \\ 0 & (z < 0), \\ \text{any value in } [0, 1] & (z = 0). \end{cases} \tag{3.19}$$

3.3.5 Softmax

For $\mathbf{z} \in \mathbb{R}^K$,

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \quad i = 1, \ldots, K. \tag{3.20}$$

Its Jacobian is

$$\frac{\partial\, \text{softmax}(\mathbf{z})_i}{\partial z_j} = p_i(\delta_{ij} - p_j), \tag{3.21}$$

where $\mathbf{p} = \text{softmax}(\mathbf{z})$ and δ_{ij} is the Kronecker delta. Matrix form:

$$\mathbf{J} = \text{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^{\top}. \tag{3.22}$$

3.3.6 Vanishing gradients (preview)

For sigmoid, $\sigma'(z) \leq 1/4$; repeated multiplication of such terms across many layers can make gradients small:

$$\prod_{\ell=1}^{L} \sigma'\big(z^{(\ell)}\big) \quad \text{tends to 0 as } L \text{ grows (in many regimes).} \tag{3.23}$$

This motivates ReLU-family activations and normalization methods, discussed later.

3.4 A Fully Worked Tiny Example (Optional)

Consider a $2 \to 2 \to 1$ network with ReLU hidden layer and sigmoid output. Let

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \mathbf{W}^{(1)} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}, \quad \mathbf{b}^{(1)} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}. \tag{3.24}$$

Hidden pre-activations:

$$\mathbf{z}^{(1)} = \begin{bmatrix} w_{11}x_1 + w_{12}x_2 + b_1 \\ w_{21}x_1 + w_{22}x_2 + b_2 \end{bmatrix}, \tag{3.25}$$

hidden activations:

$$\mathbf{a}^{(1)} = \begin{bmatrix} \max(0, z_1^{(1)}) \\ \max(0, z_2^{(1)}) \end{bmatrix}. \tag{3.26}$$

Output layer:

$$z^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + b^{(2)} = u_1 a_1^{(1)} + u_2 a_2^{(1)} + b^{(2)}, \tag{3.27}$$

$$\hat{y} = \sigma(z^{(2)}) = \frac{1}{1 + e^{-z^{(2)}}}. \tag{3.28}$$

This concrete form makes it easy to check dimensions and confirm that each layer is an affine map followed by a nonlinearity.

Chapter 4

Loss Functions

A **loss function** quantifies how far a model prediction is from the target. Training typically minimizes the empirical risk (average loss over a dataset). Let $\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{m}$ be a dataset and let the model output be $\hat{\mathbf{y}}^{(i)} = f_\theta(\mathbf{x}^{(i)})$.

4.1 Empirical Risk Minimization

Given a per-sample loss $\ell(\hat{\mathbf{y}}, \mathbf{y})$, define the empirical risk

$$\mathcal{L}(\theta) \;=\; \frac{1}{m}\sum_{i=1}^{m} \ell\big(\hat{\mathbf{y}}^{(i)}, \mathbf{y}^{(i)}\big)\,. \tag{4.1}$$

Gradient-based learning requires computing $\nabla_\theta \mathcal{L}(\theta)$, so we will derive gradients for common losses.

4.2 Regression: Mean Squared Error

4.2.1 Definition

For regression with $\mathbf{y} \in \mathbb{R}^K$ and prediction $\hat{\mathbf{y}} \in \mathbb{R}^K$, the Mean Squared Error (MSE) loss is

$$\ell_{\text{MSE}}(\hat{\mathbf{y}}, \mathbf{y}) = \|\hat{\mathbf{y}} - \mathbf{y}\|_2^2 = \sum_{k=1}^{K} (\hat{y}_k - y_k)^2. \tag{4.2}$$

A common scaled variant uses $\frac{1}{2}\|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$ to remove a factor 2 in gradients.

Over a dataset:

$$\mathcal{L}_{\text{MSE}}(\theta) = \frac{1}{m}\sum_{i=1}^{m} \|\hat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)}\|_2^2. \tag{4.3}$$

4.2.2 Gradient w.r.t. the prediction

From (4.2), the gradient w.r.t. $\hat{\mathbf{y}}$ is

$$\nabla_{\hat{\mathbf{y}}}\, \ell_{\text{MSE}}(\hat{\mathbf{y}}, \mathbf{y}) = 2(\hat{\mathbf{y}} - \mathbf{y}). \tag{4.4}$$

For the scaled loss $\frac{1}{2}\|\hat{\mathbf{y}} - \mathbf{y}\|_2^2$, the gradient becomes $\hat{\mathbf{y}} - \mathbf{y}$.

4.3 Binary Classification: Binary Cross-Entropy

Binary classification assumes $y \in \{0, 1\}$ and model output $\hat{y} \in (0, 1)$ interpreted as $P(Y = 1 \mid \mathbf{x})$. Often $\hat{y} = \sigma(z)$ where $z \in \mathbb{R}$ is the logit and σ is the sigmoid.

4.3.1 Binary cross-entropy (negative log-likelihood)

The Binary Cross-Entropy (BCE) loss for one example is

$$\ell_{\text{BCE}}(\hat{y}, y) = -\Big[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})\Big]. \tag{4.5}$$

Over a dataset:

$$\mathcal{L}_{\text{BCE}}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \ell_{\text{BCE}}(\hat{y}^{(i)}, y^{(i)}). \tag{4.6}$$

4.3.2 Gradient w.r.t. $\hat{y}$

Differentiate (4.5):

$$\frac{\partial \ell_{\text{BCE}}}{\partial \hat{y}} = -\left(\frac{y}{\hat{y}} - \frac{1 - y}{1 - \hat{y}}\right) = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})}. \tag{4.7}$$

4.3.3 Sigmoid + BCE simplification (logit gradient)

Let $\hat{y} = \sigma(z)$ with $\sigma'(z) = \hat{y}(1 - \hat{y})$ (from Chapter 3). By the chain rule:

$$\frac{\partial \ell_{\text{BCE}}}{\partial z} = \frac{\partial \ell_{\text{BCE}}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})} \cdot \hat{y}(1 - \hat{y}) = \hat{y} - y. \tag{4.8}$$

Thus, **with sigmoid + BCE, the error signal at the logit is simply prediction minus target**.

4.4 Multi-class Classification: Softmax and Categorical Cross-Entropy

Assume K classes. Let logits be $\mathbf{z} \in \mathbb{R}^K$ and probabilities be

$$\mathbf{p} = \text{softmax}(\mathbf{z}), \qquad p_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}. \tag{4.9}$$

Let the label be one-hot $\mathbf{y} \in \{0, 1\}^K$ with $\sum_k y_k = 1$.

4.4.1 Categorical cross-entropy

The Categorical Cross-Entropy (CCE) loss for one example is

$$\ell_{\text{CCE}}(\mathbf{p}, \mathbf{y}) = -\sum_{k=1}^{K} y_k \log(p_k). \tag{4.10}$$

Over a dataset:

$$\mathcal{L}_{\text{CCE}}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \ell_{\text{CCE}}(\mathbf{p}^{(i)}, \mathbf{y}^{(i)}). \tag{4.11}$$

4.4.2 Softmax Jacobian

The derivative of softmax has the form (Jacobian):

$$\frac{\partial p_i}{\partial z_j} = p_i(\delta_{ij} - p_j), \tag{4.12}$$

where δ_{ij} is the Kronecker delta ($\delta_{ij} = 1$ if $i = j$, else 0).

Equivalently, in matrix form:

$$\frac{\partial \mathbf{p}}{\partial \mathbf{z}} = \operatorname{diag}(\mathbf{p}) - \mathbf{p}\mathbf{p}^\top. \tag{4.13}$$

4.4.3 Softmax + CCE simplification (logit gradient)

We now compute $\nabla_{\mathbf{z}} \ell_{\text{CCE}}$. First,

$$\frac{\partial \ell_{\text{CCE}}}{\partial p_i} = -\frac{y_i}{p_i}. \tag{4.14}$$

Then apply chain rule:

$$\frac{\partial \ell_{\text{CCE}}}{\partial z_j} = \sum_{i=1}^{K} \frac{\partial \ell_{\text{CCE}}}{\partial p_i} \frac{\partial p_i}{\partial z_j} = \sum_{i=1}^{K} \left(-\frac{y_i}{p_i}\right) p_i(\delta_{ij} - p_j). \tag{4.15}$$

Cancel p_i:

$$= -\sum_{i=1}^{K} y_i(\delta_{ij} - p_j) = -\sum_{i=1}^{K} y_i\delta_{ij} + \sum_{i=1}^{K} y_i p_j. \tag{4.16}$$

Since $\sum_{i=1}^{K} y_i\delta_{ij} = y_j$ and $\sum_{i=1}^{K} y_i = 1$,

$$\frac{\partial \ell_{\text{CCE}}}{\partial z_j} = -p_j + p_j \cdot 1 + (p_j - y_j) = p_j - y_j. \tag{4.17}$$

Therefore,

$$\nabla_{\mathbf{z}} \ell_{\text{CCE}}(\text{softmax}(\mathbf{z}), \mathbf{y}) = \mathbf{p} - \mathbf{y}. \tag{4.18}$$

This is the multi-class analogue of (4.8).

4.5 (Optional) Huber Loss for Robust Regression

When regression data contains outliers, MSE can be overly sensitive. The Huber loss interpolates between MSE and MAE.

For scalar $y, \hat{y} \in \mathbb{R}$ with threshold $\delta > 0$:

$$\ell_{\text{Huber}}(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & \text{if } |y - \hat{y}| \le \delta, \\ \delta\left(|y - \hat{y}| - \frac{\delta}{2}\right) & \text{if } |y - \hat{y}| > \delta. \end{cases} \tag{4.19}$$

Its derivative w.r.t. $\hat{y}$ is

$$\frac{\partial \ell_{\text{Huber}}}{\partial \hat{y}} = \begin{cases} \hat{y} - y & \text{if } |y - \hat{y}| \le \delta, \\ \delta \operatorname{sign}(\hat{y} - y) & \text{if } |y - \hat{y}| > \delta. \end{cases} \tag{4.20}$$

Chapter 5

Backpropagation Algorithm

5.1 Setup: Notation and Forward Pass

Consider an L-layer feedforward neural network. For a single example $(\mathbf{x}, \mathbf{y})$, define

$$\mathbf{a}^{(0)} = \mathbf{x}. \tag{5.1}$$

For each layer $\ell = 1, 2, \ldots, L$:

$$\mathbf{z}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{a}^{(\ell-1)} + \mathbf{b}^{(\ell)}, \tag{5.2}$$

$$\mathbf{a}^{(\ell)} = \sigma^{(\ell)}\big(\mathbf{z}^{(\ell)}\big), \tag{5.3}$$

where $\mathbf{W}^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$, $\mathbf{b}^{(\ell)} \in \mathbb{R}^{n_\ell}$, and $\sigma^{(\ell)}$ is applied element-wise unless stated otherwise.

Let the prediction be $\hat{\mathbf{y}} = \mathbf{a}^{(L)}$ and the loss be

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = \mathcal{L}(\mathbf{a}^{(L)}, \mathbf{y}). \tag{5.4}$$

The goal of backpropagation is to compute the gradients $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}}$ efficiently for all ℓ.

5.2 The Delta Terms

5.2.1 Definition

Define the **delta** (error signal) at layer ℓ by

$$\boldsymbol{\delta}^{(\ell)} \;=\; \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(\ell)}} \in \mathbb{R}^{n_\ell}. \tag{5.5}$$

Once $\boldsymbol{\delta}^{(\ell)}$ is known, the parameter gradients follow in simple outer-product form (see Section 5.3).

5.2.2 Output layer delta: general form

By the chain rule,

$$\boldsymbol{\delta}^{(L)} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(L)}} \odot \frac{\partial \mathbf{a}^{(L)}}{\partial \mathbf{z}^{(L)}} = \nabla_{\mathbf{a}^{(L)}}\mathcal{L} \;\odot\; \sigma^{(L)\prime}\big(\mathbf{z}^{(L)}\big), \tag{5.6}$$

where $\odot$ denotes element-wise multiplication.

5.2.3 Hidden layer delta

For $\ell = L-1, L-2, \ldots, 1$, backpropagate the delta:

$$\boldsymbol{\delta}^{(\ell)} = \left(\mathbf{W}^{(\ell+1)}\right)^{\top} \boldsymbol{\delta}^{(\ell+1)} \odot \sigma^{(\ell)\prime}\left(\mathbf{z}^{(\ell)}\right). \tag{5.7}$$

Interpretation:

- $\left(\mathbf{W}^{(\ell+1)}\right)^{\top} \boldsymbol{\delta}^{(\ell+1)}$ maps the error signal back to layer ℓ.
- Multiplying by $\sigma^{(\ell)\prime}$ accounts for the nonlinearity at layer ℓ.

This is the central recurrence relation of backpropagation.

5.3 Gradients for Weights and Biases

5.3.1 Single example

From (5.2), each component is

$$z_i^{(\ell)} = \sum_{j=1}^{n_{\ell-1}} W_{ij}^{(\ell)} a_j^{(\ell-1)} + b_i^{(\ell)}. \tag{5.8}$$

Hence,

$$\frac{\partial z_i^{(\ell)}}{\partial W_{ij}^{(\ell)}} = a_j^{(\ell-1)}, \quad \frac{\partial z_i^{(\ell)}}{\partial b_i^{(\ell)}} = 1, \quad \frac{\partial z_i^{(\ell)}}{\partial b_k^{(\ell)}} = 0 \ (k \neq i). \tag{5.9}$$

Using $\delta_i^{(\ell)} = \frac{\partial \mathcal{L}}{\partial z_i^{(\ell)}}$, we obtain

$$\frac{\partial \mathcal{L}}{\partial W_{ij}^{(\ell)}} = \frac{\partial \mathcal{L}}{\partial z_i^{(\ell)}} \frac{\partial z_i^{(\ell)}}{\partial W_{ij}^{(\ell)}} = \delta_i^{(\ell)} \, a_j^{(\ell-1)}. \tag{5.10}$$

In matrix form:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)} \left(\mathbf{a}^{(\ell-1)}\right)^{\top}. \tag{5.11}$$

Similarly, for the bias:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} = \boldsymbol{\delta}^{(\ell)}. \tag{5.12}$$

These match the compact formulas already appearing in the draft.

5.3.2 Mini-batch (average gradient)

For a mini-batch of size m, with deltas $\boldsymbol{\delta}^{(\ell),(i)}$ and activations $\mathbf{a}^{(\ell-1),(i)}$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}} = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{\delta}^{(\ell),(i)} \left(\mathbf{a}^{(\ell-1),(i)}\right)^{\top}, \tag{5.13}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}} = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{\delta}^{(\ell),(i)}. \tag{5.14}$$

(Implementation note: in vectorized code, one stacks examples as matrices and these become matrix multiplications.)

5.3.3 Mini-batch matrix backpropagation (vectorized deltas)

We now rewrite the delta recursion in a fully vectorized mini-batch form, consistent with the matrix forward pass (Section 3.1.4) and the numerical vectorization in Chapter 6.

Mini-batch notation. Let the mini-batch size be m and stack activations as columns:

$$A^{(\ell)} = \left[a^{(\ell),(1)}, \dots, a^{(\ell),(m)}\right] \in \mathbb{R}^{n_\ell \times m}, \qquad Z^{(\ell)} = \left[z^{(\ell),(1)}, \dots, z^{(\ell),(m)}\right] \in \mathbb{R}^{n_\ell \times m}. \tag{5.15}$$

The vectorized forward pass is

$$Z^{(\ell)} = W^{(\ell)} A^{(\ell-1)} + b^{(\ell)} \mathbf{1}^\top, \qquad A^{(\ell)} = \sigma^{(\ell)}\big(Z^{(\ell)}\big), \tag{5.16}$$

where $\mathbf{1} \in \mathbb{R}^m$ is the all-ones vector and $\sigma^{(\ell)}$ is applied element-wise.

Vectorized deltas. Define the mini-batch delta matrix

$$\Delta^{(\ell)} \;=\; \frac{\partial \mathcal{L}_{\text{batch}}}{\partial Z^{(\ell)}} \in \mathbb{R}^{n_\ell \times m}, \tag{5.17}$$

where $\mathcal{L}_{\text{batch}}$ denotes the average loss over the mini-batch.

Output layer delta (matrix form). In complete generality, for the output layer $\ell = L$,

$$\Delta^{(L)} = \left(\frac{\partial \mathcal{L}_{\text{batch}}}{\partial A^{(L)}}\right) \odot \sigma^{(L)\prime}\big(Z^{(L)}\big), \tag{5.18}$$

where $\odot$ denotes element-wise multiplication.

Hidden layer delta recursion (matrix form). For $\ell = L-1, \dots, 1$, the hidden-layer deltas satisfy the vectorized recurrence

$$\Delta^{(\ell)} = \big(W^{(\ell+1)}\big)^\top \Delta^{(\ell+1)} \odot \sigma^{(\ell)\prime}\big(Z^{(\ell)}\big). \tag{5.19}$$

This is exactly the single-example formula (5.7) applied to all m columns in parallel.

Gradients from vectorized deltas. Using (5.16) and the definition of $\Delta^{(\ell)}$, the parameter gradients become

$$\frac{\partial \mathcal{L}_{\text{batch}}}{\partial W^{(\ell)}} = \frac{1}{m} \Delta^{(\ell)} \left(A^{(\ell-1)}\right)^\top \in \mathbb{R}^{n_\ell \times n_{\ell-1}}, \tag{5.20}$$

$$\frac{\partial \mathcal{L}_{\text{batch}}}{\partial b^{(\ell)}} = \frac{1}{m} \Delta^{(\ell)} \mathbf{1} \in \mathbb{R}^{n_\ell}. \tag{5.21}$$

The bias gradient follows because $b^{(\ell)} \mathbf{1}^\top$ replicates $b^{(\ell)}$ across the m columns.

Consistency check (shapes).

$$\big(W^{(\ell+1)}\big)^\top \Delta^{(\ell+1)} \in \mathbb{R}^{n_\ell \times m}, \qquad \Delta^{(\ell)} \left(A^{(\ell-1)}\right)^\top \in \mathbb{R}^{n_\ell \times n_{\ell-1}}, \tag{5.22}$$

so all matrix multiplications are dimensionally consistent.

Two common output simplifications (mini-batch). For the standard paired choices already derived in Section 5.4:

- **Sigmoid + BCE (binary)**: if $A^{(L)} = \hat{Y} \in \mathbb{R}^{1\times m}$ and $Y \in \mathbb{R}^{1\times m}$, then
$$\Delta^{(L)} = \hat{Y} - Y. \tag{5.23}$$
- **Softmax + CCE (multi-class)**: if $A^{(L)} = P \in \mathbb{R}^{K\times m}$ and one-hot labels $Y \in \mathbb{R}^{K\times m}$, then
$$\Delta^{(L)} = P - Y. \tag{5.24}$$

These are the column-wise extensions of (5.18) and (5.21).

5.4 Two Classic "Cancellations"

Backprop becomes particularly simple for common output-layer choices.

5.4.1 Sigmoid + Binary Cross-Entropy

Assume a single output logit z and sigmoid activation
$$\hat{y} = \sigma(z) = \frac{1}{1+e^{-z}}, \qquad \sigma'(z) = \hat{y}(1-\hat{y}). \tag{5.25}$$

Binary cross-entropy loss:
$$\mathcal{L}(\hat{y}, y) = -\left[y \log(\hat{y}) + (1-y)\log(1-\hat{y})\right]. \tag{5.26}$$

First,
$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = -\left(\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}}\right) = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})}. \tag{5.27}$$

Then by chain rule:
$$\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} = \frac{\hat{y}-y}{\hat{y}(1-\hat{y})} \cdot \hat{y}(1-\hat{y}) = \hat{y} - y. \tag{5.28}$$

Thus, the output delta is simply "prediction minus target."

5.4.2 Softmax + Categorical Cross-Entropy

Let logits be $\mathbf{z} \in \mathbb{R}^K$ and softmax probabilities
$$p_k = \text{softmax}(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}. \tag{5.29}$$

With one-hot label vector $\mathbf{y} \in \{0,1\}^K$, categorical cross-entropy:
$$\mathcal{L}(\mathbf{p}, \mathbf{y}) = -\sum_{k=1}^{K} y_k \log p_k. \tag{5.30}$$

A key result (derivable from the Jacobian of softmax) is:
$$\boldsymbol{\delta}^{(L)} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \mathbf{p} - \mathbf{y}. \tag{5.31}$$

Again, the delta is "predicted probabilities minus true probabilities."

5.5 Vanishing Gradients (Why Depth Is Hard)

Equation (5.7) shows that earlier-layer deltas are products of many factors. In a deep network, schematically:

$$\boldsymbol{\delta}^{(1)} \approx \left(\prod_{\ell=2}^{L} \left(\mathbf{W}^{(\ell)}\right)^{\top} \mathrm{Diag}\left(\sigma^{(\ell)\prime}(\mathbf{z}^{(\ell)})\right) \right) \boldsymbol{\delta}^{(L)}. \tag{5.32}$$

If $\sigma^{(\ell)}$ is sigmoid, then $\sigma'(z) \leq 1/4$ for all z. Multiplying many numbers $\leq 1/4$ tends to drive the magnitude of gradients toward zero, slowing learning in early layers.

5.5.1 Mitigations (mathematical view)

- ReLU-type activations: $\mathrm{ReLU}'(z) = 1$ for $z > 0$, avoiding a ubiquitous small factor.
- Careful initialization to keep activations and gradients in a reasonable scale.
- Normalization (e.g., batch normalization) and skip connections to improve gradient flow.

These ideas motivate much of modern deep learning architecture design.

5.6 Algorithm Summary (One Iteration)

For one mini-batch:

1. Forward pass: compute $(\mathbf{z}^{(\ell)}, \mathbf{a}^{(\ell)})$ for $\ell = 1, \ldots, L$ using (5.2)–(5.3).
2. Output delta: compute $\boldsymbol{\delta}^{(L)}$ using (5.6) (or the simplified forms (5.28), (5.31) when applicable).
3. Backward recursion: compute $\boldsymbol{\delta}^{(\ell)}$ for $\ell = L-1, \ldots, 1$ using (5.7).
4. Gradients: compute $\partial\mathcal{L}/\partial\mathbf{W}^{(\ell)}$ and $\partial\mathcal{L}/\partial\mathbf{b}^{(\ell)}$ using (5.13)–(5.14).
5. Update parameters with an optimizer (SGD, momentum, Adam, etc.).

This completes one training step.

Chapter 6

Concrete Numerical Example: A Network from Scratch

This chapter constructs a tiny feedforward neural network for binary classification and derives forward propagation, loss computation, and backpropagation *numerically and symbolically*. The goal is to see every quantity (z, a, δ, gradients) explicitly.

6.1 Problem Setup

We consider a toy dataset with $m = 4$ samples, input dimension $d = 2$, and binary labels $y \in \{0, 1\}$:

$$\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{4}, \qquad \mathbf{x}^{(i)} \in \mathbb{R}^2,\ y^{(i)} \in \{0, 1\}. \tag{6.1}$$

Concretely,

i	$\mathbf{x}^{(i)}$	$y^{(i)}$
1	$[0.5,\ 0.2]^\top$	0
2	$[0.9,\ 0.8]^\top$	1
3	$[0.1,\ 0.3]^\top$	0
4	$[0.8,\ 0.9]^\top$	1

(6.2)

We use a $2 \to 3 \to 1$ network:

- Hidden layer: $n_1 = 3$ neurons with ReLU.
- Output layer: $n_2 = 1$ neuron with sigmoid producing $\hat{y} \in (0, 1)$.

6.2 Parameter Initialization

Layer 1 parameters:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \\ 0.5 & 0.6 \end{bmatrix} \in \mathbb{R}^{3\times 2}, \qquad \mathbf{b}^{(1)} = \begin{bmatrix} 0.01 \\ 0.02 \\ 0.03 \end{bmatrix} \in \mathbb{R}^3. \tag{6.3}$$

Layer 2 parameters:

$$\mathbf{W}^{(2)} = \begin{bmatrix} 0.7 & 0.8 & 0.9 \end{bmatrix} \in \mathbb{R}^{1\times 3}, \qquad b^{(2)} = 0.1. \tag{6.4}$$

6.3 Forward Propagation: General Form

For each sample $\mathbf{x}$:

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}, \tag{6.5}$$

$$\mathbf{a}^{(1)} = \mathrm{ReLU}(\mathbf{z}^{(1)}), \tag{6.6}$$

$$z^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + b^{(2)}, \tag{6.7}$$

$$\hat{y} = \sigma(z^{(2)}) = \frac{1}{1 + e^{-z^{(2)}}}. \tag{6.8}$$

We use binary cross-entropy (per-sample loss):

$$\mathcal{L}^{(i)} \;=\; -\Big(y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})\Big). \tag{6.9}$$

Batch loss:

$$\mathcal{L}_{\text{batch}} = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}^{(i)}. \tag{6.10}$$

6.4 Forward Propagation: Sample 1 (Fully Expanded)

Let $\mathbf{x}^{(1)} = [0.5, 0.2]^\top$, $y^{(1)} = 0$.

6.4.1 Layer 1 pre-activation

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x}^{(1)} + \mathbf{b}^{(1)}. \tag{6.11}$$

Compute entrywise:

$$z_1^{(1)} = 0.1 \cdot 0.5 + 0.2 \cdot 0.2 + 0.01 = 0.05 + 0.04 + 0.01 = 0.10, \tag{6.12}$$

$$z_2^{(1)} = 0.3 \cdot 0.5 + 0.4 \cdot 0.2 + 0.02 = 0.15 + 0.08 + 0.02 = 0.25, \tag{6.13}$$

$$z_3^{(1)} = 0.5 \cdot 0.5 + 0.6 \cdot 0.2 + 0.03 = 0.25 + 0.12 + 0.03 = 0.40. \tag{6.14}$$

Thus

$$\mathbf{z}^{(1)} = \begin{bmatrix} 0.10 \\ 0.25 \\ 0.40 \end{bmatrix}. \tag{6.15}$$

6.4.2 Layer 1 activation (ReLU)

$$\mathbf{a}^{(1)} = \mathrm{ReLU}(\mathbf{z}^{(1)}) = \begin{bmatrix} \max(0, 0.10) \\ \max(0, 0.25) \\ \max(0, 0.40) \end{bmatrix} = \begin{bmatrix} 0.10 \\ 0.25 \\ 0.40 \end{bmatrix}. \tag{6.16}$$

6.4.3 Layer 2 pre-activation

$$z^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + b^{(2)} \tag{6.17}$$

$$= 0.7 \cdot 0.10 + 0.8 \cdot 0.25 + 0.9 \cdot 0.40 + 0.1 \tag{6.18}$$

$$= 0.07 + 0.20 + 0.36 + 0.10 = 0.73. \tag{6.19}$$

6.4.4 Output (sigmoid)

$$\hat{y}^{(1)} = \sigma(0.73) = \frac{1}{1 + e^{-0.73}}. \tag{6.20}$$

Using $e^{-0.73} \approx 0.4819$:

$$\hat{y}^{(1)} \approx \frac{1}{1 + 0.4819} = \frac{1}{1.4819} \approx 0.6748. \tag{6.21}$$

6.4.5 Loss for sample 1

Since $y^{(1)} = 0$, the loss simplifies from (6.9) to

$$\mathcal{L}^{(1)} = -\log(1 - \hat{y}^{(1)}) = -\log(1 - 0.6748) = -\log(0.3252). \tag{6.22}$$

Numerically,

$$\mathcal{L}^{(1)} \approx 1.1223. \tag{6.23}$$

6.5 Forward Propagation: Sample 2 (Detailed)

Let $\mathbf{x}^{(2)} = [0.9, 0.8]^\top$, $y^{(2)} = 1$.

6.5.1 Layer 1

$$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x}^{(2)} + \mathbf{b}^{(1)}. \tag{6.24}$$

Entrywise:

$$z_1^{(1)} = 0.1 \cdot 0.9 + 0.2 \cdot 0.8 + 0.01 = 0.09 + 0.16 + 0.01 = 0.26, \tag{6.25}$$

$$z_2^{(1)} = 0.3 \cdot 0.9 + 0.4 \cdot 0.8 + 0.02 = 0.27 + 0.32 + 0.02 = 0.61, \tag{6.26}$$

$$z_3^{(1)} = 0.5 \cdot 0.9 + 0.6 \cdot 0.8 + 0.03 = 0.45 + 0.48 + 0.03 = 0.96. \tag{6.27}$$

Since all entries are positive,

$$\mathbf{a}^{(1)} = \mathrm{ReLU}(\mathbf{z}^{(1)}) = \begin{bmatrix} 0.26 \\ 0.61 \\ 0.96 \end{bmatrix}. \tag{6.28}$$

6.5.2 Layer 2 and output

$$z^{(2)} = 0.7 \cdot 0.26 + 0.8 \cdot 0.61 + 0.9 \cdot 0.96 + 0.1 \tag{6.29}$$

$$= 0.182 + 0.488 + 0.864 + 0.1 = 1.634, \tag{6.30}$$

$$\hat{y}^{(2)} = \sigma(1.634) = \frac{1}{1 + e^{-1.634}} \approx 0.8367. \tag{6.31}$$

Loss (since $y^{(2)} = 1$):

$$\mathcal{L}^{(2)} = -\log(\hat{y}^{(2)}) \approx -\log(0.8367) \approx 0.1779. \tag{6.32}$$

6.6 Batch Loss

Assume the remaining sample losses are (as in the current draft):

$$\mathcal{L}^{(3)} \approx 0.9502, \qquad \mathcal{L}^{(4)} \approx 0.2148. \tag{6.33}$$

Then the batch loss is

$$\mathcal{L}_{\text{batch}} = \frac{1}{4}\left(\mathcal{L}^{(1)} + \mathcal{L}^{(2)} + \mathcal{L}^{(3)} + \mathcal{L}^{(4)}\right) \tag{6.34}$$

$$= \frac{1}{4}(1.1223 + 0.1779 + 0.9502 + 0.2148) \tag{6.35}$$

$$= \frac{2.4652}{4} \approx 0.6163. \tag{6.36}$$

6.7 Backpropagation: General Form

Define the output delta (for sigmoid + binary cross-entropy) as

$$\delta^{(2)} \;=\; \frac{\partial \mathcal{L}}{\partial z^{(2)}} \;=\; \hat{y} - y. \tag{6.37}$$

Then

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}} = \delta^{(2)}(\mathbf{a}^{(1)})^{\top}, \tag{6.38}$$

$$\frac{\partial \mathcal{L}}{\partial b^{(2)}} = \delta^{(2)}. \tag{6.39}$$

For the hidden layer (ReLU):

$$\boldsymbol{\delta}^{(1)} \;=\; \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}} \;=\; (\mathbf{W}^{(2)})^{\top}\delta^{(2)} \;\odot\; \mathrm{ReLU}'(\mathbf{z}^{(1)}), \tag{6.40}$$

and

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} = \boldsymbol{\delta}^{(1)}(\mathbf{x})^{\top}, \tag{6.41}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(1)}} = \boldsymbol{\delta}^{(1)}. \tag{6.42}$$

Here

$$\mathrm{ReLU}'(z) = \begin{cases} 1 & (z > 0), \\ 0 & (z \le 0). \end{cases} \tag{6.43}$$

6.8 Backpropagation: Sample 1 (Fully Expanded)

For sample 1, $y^{(1)} = 0$ and $\hat{y}^{(1)} \approx 0.6748$.

6.8.1 Output delta

From (6.37),

$$\delta_1^{(2)} = \hat{y}^{(1)} - y^{(1)} = 0.6748 - 0 = 0.6748. \tag{6.44}$$

6.8.2 Gradients for layer 2

$$\frac{\partial \mathcal{L}^{(1)}}{\partial \mathbf{W}^{(2)}} = \delta_1^{(2)} (\mathbf{a}^{(1)})^\top \tag{6.45}$$

$$= 0.6748 \cdot [0.10,\ 0.25,\ 0.40] \tag{6.46}$$

$$= [0.06748,\ 0.16870,\ 0.26992], \tag{6.47}$$

and

$$\frac{\partial \mathcal{L}^{(1)}}{\partial b^{(2)}} = \delta_1^{(2)} = 0.6748. \tag{6.48}$$

6.8.3 Hidden delta

Using (6.40):

$$(\mathbf{W}^{(2)})^\top \delta_1^{(2)} = \begin{bmatrix} 0.7 \\ 0.8 \\ 0.9 \end{bmatrix} 0.6748 = \begin{bmatrix} 0.47236 \\ 0.53984 \\ 0.60732 \end{bmatrix}. \tag{6.49}$$

Since $\mathbf{z}^{(1)} = [0.10, 0.25, 0.40]^\top$ is strictly positive, $\mathrm{ReLU}'(\mathbf{z}^{(1)}) = [1, 1, 1]^\top$, so

$$\boldsymbol{\delta}_1^{(1)} = \begin{bmatrix} 0.47236 \\ 0.53984 \\ 0.60732 \end{bmatrix}. \tag{6.50}$$

6.8.4 Gradients for layer 1

$$\frac{\partial \mathcal{L}^{(1)}}{\partial \mathbf{W}^{(1)}} = \boldsymbol{\delta}_1^{(1)} (\mathbf{x}^{(1)})^\top \tag{6.51}$$

$$= \begin{bmatrix} 0.47236 \\ 0.53984 \\ 0.60732 \end{bmatrix} \begin{bmatrix} 0.5 & 0.2 \end{bmatrix} \tag{6.52}$$

$$= \begin{bmatrix} 0.23618 & 0.09447 \\ 0.26992 & 0.10797 \\ 0.30366 & 0.12146 \end{bmatrix}, \tag{6.53}$$

and

$$\frac{\partial \mathcal{L}^{(1)}}{\partial \mathbf{b}^{(1)}} = \boldsymbol{\delta}_1^{(1)} = \begin{bmatrix} 0.47236 \\ 0.53984 \\ 0.60732 \end{bmatrix}. \tag{6.54}$$

6.9 Mini-batch Gradients (Averaging)

For a mini-batch of size $m = 4$, define per-sample gradients $g^{(i)}$. For example, layer 2 weights:

$$\frac{\partial \mathcal{L}_{\text{batch}}}{\partial \mathbf{W}^{(2)}} = \frac{1}{4} \sum_{i=1}^{4} \frac{\partial \mathcal{L}^{(i)}}{\partial \mathbf{W}^{(2)}}. \tag{6.55}$$

In the current draft, the averaged gradient is summarized as

$$\frac{\partial \mathcal{L}_{\text{batch}}}{\partial \mathbf{W}^{(2)}} \approx [0.0289,\ 0.0425,\ 0.0568]. \tag{6.56}$$

6.10 Parameter Updates (SGD)

Using learning rate $\eta = 0.1$, the gradient descent update is

$$\theta_{\text{new}} = \theta - \eta \nabla_\theta \mathcal{L}_{\text{batch}}. \tag{6.57}$$

6.10.1 Update layer 2

$$\begin{aligned} \mathbf{W}^{(2)}_{\text{new}} &= \mathbf{W}^{(2)} - 0.1 \frac{\partial \mathcal{L}_{\text{batch}}}{\partial \mathbf{W}^{(2)}} && (6.58)\\ &= [0.7,\, 0.8,\, 0.9] - 0.1[0.0289,\, 0.0425,\, 0.0568] && (6.59)\\ &= [0.6971,\, 0.7958,\, 0.8943]. && (6.60) \end{aligned}$$

The bias update is similarly

$$b^{(2)}_{\text{new}} = b^{(2)} - 0.1 \frac{\partial \mathcal{L}_{\text{batch}}}{\partial b^{(2)}}. \tag{6.61}$$

(In the current draft, a numerical value leading to $b^{(2)}_{\text{new}} \approx 0.0831$ is reported.)

6.10.2 Update layer 1

Likewise,

$$\mathbf{W}^{(1)}_{\text{new}} = \mathbf{W}^{(1)} - 0.1 \frac{\partial \mathcal{L}_{\text{batch}}}{\partial \mathbf{W}^{(1)}}, \qquad \mathbf{b}^{(1)}_{\text{new}} = \mathbf{b}^{(1)} - 0.1 \frac{\partial \mathcal{L}_{\text{batch}}}{\partial \mathbf{b}^{(1)}}. \tag{6.62}$$

The current draft summarizes the updated parameters numerically as

$$\mathbf{W}^{(1)}_{\text{new}} \approx \begin{bmatrix} 0.0933 & 0.1974 \\ 0.2937 & 0.3981 \\ 0.4931 & 0.5971 \end{bmatrix}, \qquad \mathbf{b}^{(1)}_{\text{new}} \approx \begin{bmatrix} -0.0032 \\ 0.0094 \\ 0.0142 \end{bmatrix}. \tag{6.63}$$

6.11 Second Iteration (Forward Pass Check)

To verify that the update decreases the loss, recompute the forward pass for sample 1 using updated parameters. The current draft reports (for sample 1):

$$\mathbf{z}^{(1)}_{\text{new}} = \begin{bmatrix} 0.0872 \\ 0.2388 \\ 0.3834 \end{bmatrix}, \qquad \mathbf{a}^{(1)}_{\text{new}} = \begin{bmatrix} 0.0872 \\ 0.2388 \\ 0.3834 \end{bmatrix}, \tag{6.64}$$

$$z^{(2)}_{\text{new}} \approx 0.6772, \qquad \hat{y}^{(1)}_{\text{new}} = \sigma(0.6772) \approx 0.6636, \tag{6.65}$$

so the new loss becomes

$$\mathcal{L}^{(1)}_{\text{new}} = -\log(1 - \hat{y}^{(1)}_{\text{new}}) = -\log(1 - 0.6636) = -\log(0.3364) \approx 1.0898. \tag{6.66}$$

Thus the loss decreases from ≈ 1.1223 to ≈ 1.0898, consistent with gradient descent improving the objective.

Chapter 7

Advanced Numerical Demonstrations

This chapter extends the concrete network in Chapter 6 and demonstrates, with explicit numbers, how common optimization and regularization techniques modify updates and activations.

7.1 Optimization Dynamics

7.1.1 Effect of the learning rate

Consider a parameter vector (or weight matrix flattened) θ and a gradient estimate $g = \nabla_\theta \mathcal{L}(\theta)$. A single gradient descent step is

$$\theta_{\text{new}} = \theta - \eta g, \tag{7.1}$$

where $\eta > 0$ is the learning rate.

Concrete example (Layer 2 weights). Using the Chapter 6 batch-gradient estimate

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}} \approx [0.0289,\ 0.0425,\ 0.0568], \tag{7.2}$$

the update magnitude depends linearly on η.

- If $\eta = 0.1$:

$$\Delta \mathbf{W}^{(2)} = -0.1\,[0.0289, 0.0425, 0.0568] = [-0.00289, -0.00425, -0.00568]. \tag{7.1}$$

- If $\eta = 0.5$ (more aggressive):

$$\Delta \mathbf{W}^{(2)} = -0.5\,[0.0289, 0.0425, 0.0568] = [-0.01445, -0.02125, -0.02840]. \tag{7.2}$$

A larger η yields faster movement but increases the risk of overshooting minima or divergence.

7.1.2 Loss curve (illustrative)

Denote the batch loss after iteration t by $\mathcal{L}_t$. An example monotone decrease (as seen in the earlier draft) is:

Iteration t	$\mathcal{L}_t$
0	0.6160
1	0.5847
5	0.5172

(7.3)

7.2 Regularization Example: L2

7.2.1 Definition

L2 regularization (weight decay) augments the loss by a penalty on weight magnitudes:

$$\mathcal{L}_{\text{reg}}(\theta) = \mathcal{L}(\theta) + \lambda \sum_{\ell} \|\mathbf{W}^{(\ell)}\|_F^2, \tag{7.4}$$

where $\lambda > 0$ controls the penalty strength and

$$\|\mathbf{W}\|_F^2 = \sum_i \sum_j W_{ij}^2 \tag{7.5}$$

is the squared Frobenius norm.

7.2.2 Concrete computation

Using the Chapter 6 weights:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \\ 0.5 & 0.6 \end{bmatrix}, \qquad \mathbf{W}^{(2)} = [0.7, 0.8, 0.9]. \tag{7.6}$$

Compute squared Frobenius norms:

$$\|\mathbf{W}^{(1)}\|_F^2 = 0.1^2 + 0.2^2 + 0.3^2 + 0.4^2 + 0.5^2 + 0.6^2 = 0.01 + 0.04 + 0.09 + 0.16 + 0.25 + 0.36 = 0.91, \tag{7.4}$$

$$\|\mathbf{W}^{(2)}\|_F^2 = 0.7^2 + 0.8^2 + 0.9^2 = 0.49 + 0.64 + 0.81 = 1.94. \tag{7.5}$$

If $\lambda = 0.01$, the total penalty (weights only) becomes

$$\lambda \left(\|\mathbf{W}^{(1)}\|_F^2 + \|\mathbf{W}^{(2)}\|_F^2 \right) = 0.01(0.91 + 1.94) = 0.01(2.85) = 0.0285. \tag{7.6}$$

Hence, if $\mathcal{L} = 0.616$ then

$$\mathcal{L}_{\text{reg}} = 0.616 + 0.0285 = 0.6445. \tag{7.7}$$

7.2.3 Gradient effect (weight decay view)

Differentiating,

$$\nabla_{\mathbf{W}^{(\ell)}} \mathcal{L}_{\text{reg}} = \nabla_{\mathbf{W}^{(\ell)}} \mathcal{L} + 2\lambda \mathbf{W}^{(\ell)}. \tag{7.7}$$

Thus the update becomes

$$\mathbf{W}^{(\ell)} \leftarrow \mathbf{W}^{(\ell)} - \eta \left(\nabla_{\mathbf{W}^{(\ell)}} \mathcal{L} + 2\lambda \mathbf{W}^{(\ell)} \right), \tag{7.8}$$

which explicitly pulls weights toward zero each step.

7.3 Momentum Optimization

7.3.1 Update rule

Momentum maintains a velocity vector $\mathbf{v}_t$:

$$\mathbf{v}_{t+1} = \beta \mathbf{v}_t + \nabla_\theta \mathcal{L}(\theta_t), \tag{7.9}$$

$$\theta_{t+1} = \theta_t - \eta\, \mathbf{v}_{t+1}, \tag{7.10}$$

with momentum coefficient $\beta \in (0, 1)$ (often 0.9).

7.3.2 Two-step numerical example (Layer 2 weights)

Let $\beta = 0.9$, $\eta = 0.1$, and initialize $\mathbf{v}_0 = \mathbf{0}$. Use the gradient $g_1 = [0.0289, 0.0425, 0.0568]$.

Iteration 1.

$$\mathbf{v}_1 = 0.9\mathbf{0} + g_1 = [0.0289, 0.0425, 0.0568], \tag{7.8}$$

$$\begin{aligned}\mathbf{W}_1^{(2)} &= \mathbf{W}_0^{(2)} - 0.1\,\mathbf{v}_1 = [0.7, 0.8, 0.9] - 0.1[0.0289, 0.0425, 0.0568] \\ &= [0.6971, 0.7958, 0.8943]. \end{aligned} \tag{7.10}$$

Iteration 2. Suppose the new gradient is $g_2 = [0.0215, 0.0318, 0.0425]$.

$$\begin{aligned}\mathbf{v}_2 &= 0.9\mathbf{v}_1 + g_2 = 0.9[0.0289, 0.0425, 0.0568] + [0.0215, 0.0318, 0.0425] \\ &= [0.0260, 0.0383, 0.0511] + [0.0215, 0.0318, 0.0425] = [0.0475, 0.0701, 0.0936], \end{aligned} \tag{7.12}$$

$$\begin{aligned}\mathbf{W}_2^{(2)} &= \mathbf{W}_1^{(2)} - 0.1\mathbf{v}_2 = [0.6971, 0.7958, 0.8943] - 0.1[0.0475, 0.0701, 0.0936] \\ &= [0.6919, 0.7890, 0.8758]. \end{aligned} \tag{7.13}$$

Momentum accumulates consistent gradient directions, often accelerating convergence.

7.4 Batch Normalization (Numerical Calculation)

7.4.1 Definition

Given pre-activations $z_j^{(i)}$ for neuron j over a mini-batch of size m_B, batch normalization computes:

$$\mu_{B,j} = \frac{1}{m_B} \sum_{i=1}^{m_B} z_j^{(i)}, \tag{7.11}$$

$$\sigma_{B,j}^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} \left(z_j^{(i)} - \mu_{B,j} \right)^2, \tag{7.12}$$

$$\hat{z}_j^{(i)} = \frac{z_j^{(i)} - \mu_{B,j}}{\sqrt{\sigma_{B,j}^2 + \epsilon}}, \tag{7.13}$$

$$y_j^{(i)} = \gamma_j \hat{z}_j^{(i)} + \beta_j, \tag{7.14}$$

where γ_j, β_j are learnable parameters and $\epsilon > 0$ ensures numerical stability.

7.4.2 Concrete computation for one neuron

Take a hypothetical batch of four pre-activations for neuron $j = 1$:

$$z_1^{(1)} = 0.10, \quad z_1^{(2)} = 0.26, \quad z_1^{(3)} = 0.08, \quad z_1^{(4)} = 0.22. \tag{7.15}$$

Mean:

$$\mu_{B,1} = \frac{0.10 + 0.26 + 0.08 + 0.22}{4} = \frac{0.66}{4} = 0.165. \tag{7.16}$$

Variance:

$$\begin{aligned}\sigma_{B,1}^2 &= \frac{1}{4}\left[(0.10 - 0.165)^2 + (0.26 - 0.165)^2 + (0.08 - 0.165)^2 + (0.22 - 0.165)^2\right] \\ &= \frac{1}{4}\left[0.004225 + 0.009025 + 0.007225 + 0.003025\right] = \frac{0.02350}{4} = 0.005875. \end{aligned} \tag{7.14}$$

With $\epsilon = 10^{-5}$, normalize sample 1:

$$\hat{z}_1^{(1)} = \frac{0.10 - 0.165}{\sqrt{0.005875 + 10^{-5}}} = \frac{-0.065}{\sqrt{0.005885}} \approx \frac{-0.065}{0.0767} \approx -0.848. \tag{7.14'}$$

If $\gamma_1 = 1.0$ and $\beta_1 = 0.0$, then

$$y_1^{(1)} = \gamma_1 \hat{z}_1^{(1)} + \beta_1 = -0.848. \tag{7.15}$$

This reproduces the style of the batch-normalization calculation in the current draft.

7.5 Dropout Regularization (Numerical Calculation)

7.5.1 Training-time dropout

Let the hidden activation vector be $\mathbf{h} \in \mathbb{R}^n$. Dropout samples a mask $\mathbf{m} \in \{0, 1\}^n$ i.i.d. as

$$m_k \sim \text{Bernoulli}(1 - p), \tag{7.17}$$

and applies

$$\mathbf{h}_{\text{drop}} = \mathbf{h} \odot \mathbf{m}. \tag{7.18}$$

Concrete example (Layer 1 activation of sample 1). Using $\mathbf{h}^{(1)} = [0.10, 0.25, 0.40]^\top$ and dropout probability $p = 0.5$, suppose the sampled mask is

$$\mathbf{m} = [1, 0, 1]^\top. \tag{7.16}$$

Then

$$\mathbf{h}_{\text{drop}}^{(1)} = [0.10, 0.25, 0.40]^\top \odot [1, 0, 1]^\top = [0.10, 0, 0.40]^\top. \tag{7.17}$$

For $\mathbf{W}^{(2)} = [0.7, 0.8, 0.9]$ and $b^{(2)} = 0.1$, the new pre-activation becomes

$$z_{\text{drop}}^{(2)} = \mathbf{W}^{(2)}\mathbf{h}_{\text{drop}}^{(1)} + b^{(2)} = 0.7(0.10) + 0.8(0) + 0.9(0.40) + 0.1 = 0.07 + 0 + 0.36 + 0.1 = 0.53. \tag{7.18}$$

Dropout introduces stochasticity that discourages co-adaptation of features.

7.5.2 Test-time scaling

A common convention is to scale activations at inference by $(1-p)$ (if not using inverted dropout):

$$\mathbf{h}_{\text{test}} = (1-p)\mathbf{h}. \tag{7.19}$$

This makes the expected activation match between train and test.

7.6 Multi-sample Vectorized Processing

Vectorization replaces loops over samples with matrix operations. Stack a mini-batch of m inputs as a matrix

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \cdots & \mathbf{x}^{(m)} \end{bmatrix} \in \mathbb{R}^{d \times m}. \tag{7.20}$$

For a layer with weights $\mathbf{W} \in \mathbb{R}^{n \times d}$ and bias $\mathbf{b} \in \mathbb{R}^n$, the pre-activations for the entire batch are

$$\mathbf{Z} = \mathbf{W}\mathbf{X} + \mathbf{b}\mathbf{1}^\top, \tag{7.19}$$

where $\mathbf{1} \in \mathbb{R}^m$ is the all-ones vector. Then apply activation element-wise:

$$\mathbf{A} = \sigma(\mathbf{Z}). \tag{7.21}$$

This single matrix multiplication computes all m samples in parallel and is the core reason GPUs accelerate neural network training.

Chapter 8

Optimization Techniques

We train a neural network by minimizing an empirical risk over parameters θ. Let the dataset be $\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{N}$ and define

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \ell\big(f_\theta(\mathbf{x}^{(i)}), \mathbf{y}^{(i)}\big)\,. \tag{8.1}$$

Optimization algorithms produce a sequence $\{\theta_t\}_{t \geq 0}$ that (typically) reduces $\mathcal{L}(\theta_t)$.

8.1 Stochastic Gradient Descent (SGD)

8.1.1 Full-batch gradient descent

The basic gradient descent iteration is

$$\theta_{t+1} = \theta_t - \eta_t \nabla \mathcal{L}(\theta_t), \tag{8.2}$$

where $\eta_t > 0$ is the learning rate (possibly time-dependent).

8.1.2 Mini-batch SGD

In deep learning, $\nabla \mathcal{L}(\theta)$ is expensive when N is large. Let $\mathcal{B}_t \subset \{1, \ldots, N\}$ be a mini-batch of size B sampled at iteration t. Define the mini-batch objective

$$\mathcal{L}_{\mathcal{B}_t}(\theta) = \frac{1}{B} \sum_{i \in \mathcal{B}_t} \ell\big(f_\theta(\mathbf{x}^{(i)}), \mathbf{y}^{(i)}\big)\,, \tag{8.3}$$

and its gradient estimate

$$g_t := \nabla \mathcal{L}_{\mathcal{B}_t}(\theta_t). \tag{8.4}$$

Then SGD updates

$$\theta_{t+1} = \theta_t - \eta_t g_t. \tag{8.5}$$

Under uniform sampling (with standard independence assumptions),

$$\mathbb{E}[g_t \mid \theta_t] = \nabla \mathcal{L}(\theta_t), \tag{8.6}$$

so g_t is an unbiased estimator of the full gradient.

8.1.3 Learning-rate schedules (common choices)

A constant learning rate $\eta_t = \eta$ is often suboptimal. Typical schedules include:

- **Step decay:** $\eta_t = \eta_0 \gamma^{\lfloor t/T \rfloor}$ for some $\gamma \in (0, 1)$ and step period T.
- **Polynomial decay:** $\eta_t = \eta_0 (1 + t)^{-\alpha}$ for $\alpha \in (0, 1]$.
- **Cosine decay (common in practice):** $\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min})\,(1 + \cos(\pi t/T))$.

8.1.4 A basic descent inequality (smooth case)

Assume $\mathcal{L}$ has L-Lipschitz gradient:

$$\|\nabla \mathcal{L}(\theta) - \nabla \mathcal{L}(\theta')\|_2 \leq L\|\theta - \theta'\|_2. \tag{8.7}$$

Then a standard inequality implies

$$\mathcal{L}(\theta_{t+1}) \leq \mathcal{L}(\theta_t) - \eta_t \langle \nabla \mathcal{L}(\theta_t), g_t \rangle + \frac{L\eta_t^2}{2}\|g_t\|_2^2. \tag{8.8}$$

In the deterministic case $g_t = \nabla \mathcal{L}(\theta_t)$ and small enough η_t, the loss decreases each step.

8.2 Momentum

Momentum accelerates SGD in directions of consistent descent by accumulating a "velocity".

8.2.1 Heavy-ball momentum

Let $g_t = \nabla \mathcal{L}_{\mathcal{B}_t}(\theta_t)$. Define velocity v_t via

$$v_{t+1} = \beta v_t + g_t, \tag{8.9}$$
$$\theta_{t+1} = \theta_t - \eta_t v_{t+1}, \tag{8.10}$$

where $\beta \in [0, 1)$ is the momentum coefficient (often 0.9).

Unrolling (8.9) (with $v_0 = 0$) yields

$$v_t = \sum_{k=0}^{t-1} \beta^{t-1-k} g_k, \tag{8.11}$$

so v_t is an exponentially weighted moving average of past gradients.

8.2.2 Nesterov accelerated gradient (NAG)

A popular variant evaluates the gradient at a look-ahead point:

$$v_{t+1} = \beta v_t + \nabla \mathcal{L}_{\mathcal{B}_t}(\theta_t - \eta_t \beta v_t)\,, \tag{8.12}$$
$$\theta_{t+1} = \theta_t - \eta_t v_{t+1}. \tag{8.13}$$

This can reduce oscillations in narrow valleys compared to (8.10).

8.3 Adaptive Methods (RMSProp, Adam, AdamW)

Adaptive optimizers rescale updates coordinate-wise using second-moment statistics of gradients.

8.3.1 RMSProp (core idea)

Maintain an exponential moving average of squared gradients:

$$v_{t+1} = \beta_2 v_t + (1-\beta_2)(g_t \odot g_t), \tag{8.14}$$

and update

$$\theta_{t+1} = \theta_t - \eta_t \frac{g_t}{\sqrt{v_{t+1}} + \varepsilon}, \tag{8.15}$$

where all operations are element-wise.

8.3.2 Adam (Adaptive Moment Estimation)

Adam maintains first and second moment estimates

$$m_{t+1} = \beta_1 m_t + (1-\beta_1) g_t, \tag{8.16}$$
$$v_{t+1} = \beta_2 v_t + (1-\beta_2)(g_t \odot g_t), \tag{8.17}$$

with bias corrections

$$\hat{m}_{t+1} = \frac{m_{t+1}}{1-\beta_1^{t+1}}, \tag{8.18}$$
$$\hat{v}_{t+1} = \frac{v_{t+1}}{1-\beta_2^{t+1}}. \tag{8.19}$$

The Adam update is

$$\theta_{t+1} = \theta_t - \eta_t \frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1}} + \varepsilon}. \tag{8.20}$$

Typical defaults are $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\varepsilon = 10^{-8}$.

8.3.3 AdamW (decoupled weight decay)

With L2 regularization, one often writes $\nabla \mathcal{L}(\theta) + \lambda\theta$. However, for adaptive methods the effect of adding $\lambda\theta$ inside the gradient can differ from "decoupled" weight decay.

AdamW applies weight decay directly to parameters:

$$\theta_{t+1} = (1-\eta_t\lambda)\theta_t - \eta_t \frac{\hat{m}_{t+1}}{\sqrt{\hat{v}_{t+1}} + \varepsilon}. \tag{8.21}$$

This cleanly separates the shrinkage term from the adaptive gradient step.

8.4 Regularization as Optimization

Regularization modifies the training objective to encourage desirable parameter structure (small norm, sparsity, robustness).

8.4.1 L2 regularization (weight decay) and MAP interpretation

Add an L2 penalty on weights:

$$\mathcal{L}_{\text{reg}}(\theta) = \mathcal{L}(\theta) + \frac{\lambda}{2} \sum_{\ell} \|\mathbf{W}^{(\ell)}\|_F^2. \tag{8.22}$$

Then

$$\nabla_{\mathbf{W}^{(\ell)}} \mathcal{L}_{\text{reg}} = \nabla_{\mathbf{W}^{(\ell)}} \mathcal{L} + \lambda \mathbf{W}^{(\ell)}. \tag{8.23}$$

With SGD, the update becomes

$$\mathbf{W}^{(\ell)} \leftarrow (1 - \eta_t \lambda)\mathbf{W}^{(\ell)} - \eta_t \nabla_{\mathbf{W}^{(\ell)}} \mathcal{L}. \tag{8.24}$$

Thus weights shrink at each step by a factor $(1 - \eta_t \lambda)$.

Probabilistic view (sketch): minimizing (8.22) corresponds to MAP estimation under an (independent) Gaussian prior on weights, since $-\log p(\mathbf{W})$ is proportional to $\|\mathbf{W}\|_F^2$.

8.4.2 L1 regularization and sparsity

L1-regularized objective:

$$\mathcal{L}_{\text{L1}}(\theta) = \mathcal{L}(\theta) + \lambda \sum_{\ell} \|\mathbf{W}^{(\ell)}\|_1 = \mathcal{L}(\theta) + \lambda \sum_{\ell} \sum_{i,j} |W_{ij}^{(\ell)}|. \tag{8.25}$$

The subgradient satisfies

$$\frac{\partial}{\partial W_{ij}^{(\ell)}} |W_{ij}^{(\ell)}| = \begin{cases} \operatorname{sign}(W_{ij}^{(\ell)}) & W_{ij}^{(\ell)} \neq 0, \\ s \in [-1, 1] & W_{ij}^{(\ell)} = 0. \end{cases} \tag{8.26}$$

L1 tends to produce sparse solutions (many parameters exactly zero).

8.4.3 Dropout (inverted dropout)

Let $\mathbf{a}^{(\ell)}$ be activations at layer ℓ. Sample a mask $\mathbf{m}^{(\ell)} \in \{0, 1\}^{n_\ell}$ i.i.d. with

$$m_j^{(\ell)} \sim \text{Bernoulli}(q), \qquad q = 1 - p. \tag{8.27}$$

In inverted dropout, training-time activations are

$$\tilde{\mathbf{a}}^{(\ell)} = \frac{1}{q} \big(\mathbf{m}^{(\ell)} \odot \mathbf{a}^{(\ell)}\big). \tag{8.28}$$

Then

$$\mathbb{E}[\tilde{\mathbf{a}}^{(\ell)} \mid \mathbf{a}^{(\ell)}] = \mathbf{a}^{(\ell)}, \tag{8.29}$$

so no additional scaling is needed at inference time (contrast with the non-inverted convention).

8.4.4 Batch normalization (BN)

Given pre-activations $z_j^{(\ell),(i)}$ for neuron j over a mini-batch $i = 1, \dots, B$, BN computes

$$\mu_{B,j} = \frac{1}{B} \sum_{i=1}^{B} z_j^{(\ell),(i)}, \tag{8.30}$$

$$\sigma_{B,j}^2 = \frac{1}{B} \sum_{i=1}^{B} \left(z_j^{(\ell),(i)} - \mu_{B,j} \right)^2, \tag{8.31}$$

$$\hat{z}_j^{(\ell),(i)} = \frac{z_j^{(\ell),(i)} - \mu_{B,j}}{\sqrt{\sigma_{B,j}^2 + \varepsilon}}, \tag{8.32}$$

$$y_j^{(\ell),(i)} = \gamma_j \hat{z}_j^{(\ell),(i)} + \beta_j. \tag{8.33}$$

Here γ_j, β_j are learnable parameters.

Training vs inference. During training, BN uses mini-batch statistics $\mu_{B,j}, \sigma_{B,j}^2$. During inference, implementations typically use running averages (estimated during training) to avoid dependence on the test-time batch composition.

Why it helps. BN stabilizes the scale of intermediate representations, often enabling larger learning rates and improving gradient flow in deep networks, while also injecting mild stochasticity due to batch statistics.

8.5 Stability Tricks (practical)

8.5.1 Gradient clipping

To prevent exploding gradients, clip by global norm:

$$g_t \leftarrow g_t \cdot \min\left(1, \frac{\tau}{\|g_t\|_2}\right), \tag{8.34}$$

for threshold $\tau > 0$.

8.5.2 Mini-batch size trade-off

Small batches increase gradient noise (sometimes improving exploration and generalization), while large batches reduce variance but can require careful learning-rate scaling and warmup.

Chapter 9

Analysis and Theory

This chapter expands the theoretical part of the text. The goal is not to provide fully detailed proofs, but to state results precisely, clarify assumptions, and give *proof sketches* and intuition.

9.1 Function Approximation

9.1.1 Setting and notation

Let $K \subset \mathbb{R}^d$ be a compact set (e.g., $K = [0,1]^d$). Let $\sigma : \mathbb{R} \to \mathbb{R}$ be an activation function. A one-hidden-layer (two-layer) network with width N is

$$f_N(x) = \sum_{j=1}^{N} a_j \, \sigma(\mathbf{w}_j^\top x + b_j), \qquad x \in \mathbb{R}^d, \tag{9.1}$$

where $(a_j, \mathbf{w}_j, b_j) \in \mathbb{R} \times \mathbb{R}^d \times \mathbb{R}$ are parameters.

9.1.2 Universal Approximation Theorem (UAT)

Theorem (Universal approximation; informal). If σ is a non-polynomial activation (e.g., sigmoid, ReLU, tanh), then for any continuous $f \in C(K)$ and any $\varepsilon > 0$, there exists N and parameters such that

$$\sup_{x \in K} |f_N(x) - f(x)| < \varepsilon. \tag{9.2}$$

This asserts *existence* of an approximating network but does not give an efficient method to find it.

Proof sketch (high level). One common route uses functional analysis:

- Consider the set $\mathcal{F} = \{f_N : N \in \mathbb{N}\}$ and its closure in $(C(K), \|\cdot\|_\infty)$.
- Show that if σ is non-polynomial, then $\mathcal{F}$ is dense in $C(K)$ (often via a Hahn–Banach / Riesz representation argument: any continuous linear functional separating $\mathcal{F}$ from $C(K)$ would correspond to a signed measure μ, and one shows $\int \sigma(\mathbf{w}^\top x + b)\, d\mu(x) = 0$ for all $(\mathbf{w}, b)$ forces $\mu = 0$).
- Density implies any continuous f can be approximated uniformly on K.

Different proofs exist (e.g., Stone–Weierstrass style arguments for specific activations).

9.1.3 Approximation rates (why UAT is not enough)

UAT does not tell how large N must be as a function of ε. A useful theoretical question is: for a function class $\mathcal{G}$ (e.g., Lipschitz or Sobolev functions), what is the best achievable error

$$\inf_{f_N \in \mathcal{F}_N} \|f_N - f\|? \tag{9.3}$$

where $\mathcal{F}_N$ is the class of width-N networks of form (9.1).

Very roughly:

- **Smooth functions** can often be approximated faster as N increases.
- **High dimension** (d large) typically leads to slow worst-case rates ("curse of dimensionality") unless f has exploitable structure (sparsity, compositional form, low effective dimension).

This motivates studying *structured* targets and *deep* architectures.

9.1.4 Why depth helps (compositional structure)

A depth-L network can be viewed as a compositional function class:

$$f(x) = f^{(L)} \circ f^{(L-1)} \circ \cdots \circ f^{(1)}(x), \tag{9.4}$$

where each $f^{(\ell)}$ is an affine map plus nonlinearity.

If the target function itself has a compositional structure (e.g., it is naturally written as nested low-dimensional functions), a deep network can represent/approximate it using far fewer parameters than a shallow one.

9.2 Depth vs. Width

9.2.1 Expressivity measures

Two common notions:

- **Representation power:** Can the network represent a given function exactly?
- **Approximation power:** Can it approximate within error ε?

Depth increases expressivity because repeated composition creates many "regions" of different affine behavior (especially for piecewise-linear activations like ReLU).

9.2.2 Piecewise linear regions (ReLU intuition)

A ReLU network is piecewise linear. The input space is partitioned into regions within which the network is an affine function. Depth can increase the number of such regions dramatically, often exponentially in depth under suitable conditions.

Proof sketch (intuition). Each ReLU introduces a hyperplane boundary where a unit switches between active/inactive. Composing layers leads to new boundaries that are mapped and "folded" by previous layers, creating many linear regions. This creates a combinatorial growth in region count with depth.

9.2.3 Separation results (functions needing depth)

Theorem (informal separation). There exist families of functions that can be represented by a deep network with polynomial size (parameters/units) but require exponential width if restricted to shallow networks.

Example intuition: parity / compositional Boolean functions. A parity-like function has a strong hierarchical structure: it can be computed by composing XORs on pairs, which naturally forms a tree of depth $\log n$. Shallow representations must "memorize" many input patterns, leading to exponential size in the worst case.

9.3 Optimization Landscapes

9.3.1 Nonconvexity and critical points

Training a neural network typically solves

$$\min_{\theta} \ \mathcal{L}(\theta), \tag{9.5}$$

where $\mathcal{L}$ is nonconvex in θ due to composition and nonlinearities.

A critical point satisfies

$$\nabla \mathcal{L}(\theta) = 0. \tag{9.6}$$

Second-order behavior is characterized by the Hessian

$$H(\theta) = \nabla^2 \mathcal{L}(\theta). \tag{9.7}$$

A point can be a local minimum ($H \succeq 0$), local maximum ($H \preceq 0$), or saddle (indefinite Hessian).

9.3.2 Saddle points in high dimension (intuition)

In high-dimensional parameter spaces, saddle points are more common than strict local maxima. SGD noise and mini-batching introduce stochasticity that can help escape saddle regions, which partially explains why first-order methods work well in practice.

9.3.3 Overparameterization and benign landscapes (idea)

Modern networks are often overparameterized (more parameters than samples). Empirically, this regime often allows reaching near-zero training error. A common theoretical theme is that, with sufficient width, gradient-based methods behave almost like convex optimization in a neighborhood of initialization (related to linearization/NTK-type arguments).

Proof sketch (idea only). Linearize the network around initialization θ_0:

$$f_\theta(x) \approx f_{\theta_0}(x) + \nabla_\theta f_{\theta_0}(x)^\top (\theta - \theta_0). \tag{9.8}$$

If this linearization remains accurate during training and the induced kernel matrix is well-conditioned, then gradient descent can be analyzed similarly to kernel regression.

9.4 Generalization

9.4.1 Train vs. test

Let P be the (unknown) data distribution on (X, Y). Define the population risk

$$R(\theta) = \mathbb{E}_{(X,Y)\sim P}\big[\ell(f_\theta(X), Y)\big], \tag{9.9}$$

and empirical risk (training loss)

$$\hat{R}_n(\theta) = \frac{1}{n}\sum_{i=1}^{n} \ell(f_\theta(x_i), y_i). \tag{9.10}$$

Generalization asks how close $\hat{R}_n(\theta)$ is to $R(\theta)$, especially for $\hat{\theta}$ produced by training.

9.4.2 Bias–variance decomposition (squared loss)

For squared loss in regression with a learning algorithm producing a predictor $\hat{f}$ from data $\mathcal{D}$, one can decompose the expected test error at a point x:

$$\mathbb{E}_{\mathcal{D}}\big[(\hat{f}(x) - f^\star(x))^2\big] = \underbrace{\big(\mathbb{E}_{\mathcal{D}}[\hat{f}(x)] - f^\star(x)\big)^2}_{\text{Bias}^2} + \underbrace{\mathbb{E}_{\mathcal{D}}\big[(\hat{f}(x) - \mathbb{E}_{\mathcal{D}}[\hat{f}(x)])^2\big]}_{\text{Variance}}. \tag{9.11}$$

(An additional irreducible noise term appears when $Y = f^\star(X) + \varepsilon$ with noise.)

Interpretation. Increasing model complexity often reduces bias but increases variance, motivating regularization and early stopping.

9.4.3 Capacity control and uniform convergence (sketch)

A typical form of learning-theory result bounds the gap between population and empirical risks over a hypothesis class $\mathcal{H}$:

$$\sup_{h\in\mathcal{H}} |R(h) - \hat{R}_n(h)|. \tag{9.12}$$

This can be controlled by complexity measures such as VC dimension or Rademacher complexity (especially for bounded loss classes).

Proof sketch (outline). One uses symmetrization:

$$\mathbb{E}\left[\sup_{h\in\mathcal{H}}(R(h) - \hat{R}_n(h))\right] \le 2\,\mathbb{E}\left[\sup_{h\in\mathcal{H}} \frac{1}{n}\sum_{i=1}^{n} \sigma_i h(x_i)\right], \tag{9.13}$$

where $\sigma_i \in \{-1, +1\}$ are Rademacher variables, then bounds the right-hand side using contraction inequalities and norm constraints.

9.4.4 Implicit regularization (phenomenon)

Even without explicit penalties, gradient-based training can prefer certain solutions among many interpolating ones. For example, in linear regression, gradient descent initialized at zero converges to the minimum ℓ_2-norm solution among those that fit the data. Deep networks exhibit more complex forms of implicit bias, but the theme remains: optimization dynamics can act as a regularizer.

9.5 Interpolation and Double Descent

9.5.1 Classical U-shaped curve

In classical statistics, test error often decreases with model complexity (bias reduction) and then increases (variance increase), yielding a U-shaped curve.

9.5.2 Interpolation threshold

When a model becomes expressive enough to fit the training set perfectly, one reaches the interpolation regime:

$$\hat{R}_n(\hat{\theta}) \approx 0. \tag{9.14}$$

Classically, perfect fit suggests overfitting, but modern deep learning often operates in this regime.

9.5.3 Double descent (empirical phenomenon)

In many modern settings, the test error can decrease again as parameters increase further, producing a "double descent" curve:

- Underparameterized: decreasing test error.
- Near interpolation threshold: peak test error.
- Overparameterized: decreasing test error again.

This is an active research area and not fully explained by classical bias–variance alone.

Sketch of one explanation route. In linear models, one can analyze the minimum-norm interpolating solution explicitly and show that increasing parameters changes the geometry of interpolation and the effective complexity (e.g., via eigenvalues of the data covariance), leading to non-monotone risk. Deep networks are more complex, but similar geometric/effective-dimension ideas appear.

9.6 What theory does *not* yet explain

Even with the above tools, many practical behaviors remain only partially understood:

- Why certain architectures (e.g., residual connections, attention) train reliably at scale.
- Why SGD with specific hyperparameters generalizes well despite extreme overparameterization.
- Predicting performance from data/model/compute scaling in a principled way.

Thus, the theory is best viewed as a set of lenses: approximation, optimization, and generalization, each explaining part of the empirical success.

Chapter 10

Computational Graphs and Automatic Differentiation

The backpropagation algorithm presented in Chapter 5 is written for a specific network structure (layer-by-layer composition). This chapter generalizes it to arbitrary computational graphs, which is essential for modern frameworks (PyTorch, TensorFlow) and complex architectures (RNNs, Transformers, dynamic graphs).

10.1 Computational Graphs: Formal Definition

10.1.1 Directed acyclic graphs (DAGs)

A **computational graph** is a directed acyclic graph (DAG) where:

- Each **node** v represents a variable or operation.
- Each **edge** (u, v) represents data flow: output of u is input to v.
- **Leaf nodes** (sources) hold input data $\mathbf{x}$ or parameters θ.
- **Root nodes** (sinks) represent the loss or output $\mathcal{L}$.

An acyclic structure ensures that a topological ordering exists: we can label nodes $v_1, \ldots, v_n$ such that if (v_i, v_j) is an edge, then $i < j$.

10.1.2 Example: Simple expression graph

Consider computing $y = (x_1 + x_2) \cdot x_1$ where inputs are $x_1, x_2 \in \mathbb{R}$.

Nodes:

- $v_1 = x_1$ (leaf)
- $v_2 = x_2$ (leaf)
- $v_3 = v_1 + v_2$ (addition)
- $v_4 = v_3 \cdot v_1$ (multiplication)
- $v_5 = y = v_4$ (output/root)

Edges: (v_1, v_3), (v_2, v_3), (v_3, v_4), (v_1, v_4), (v_4, v_5).

The topological order is v_1, v_2, v_3, v_4, v_5. Forward evaluation follows this order; backward propagation (differentiation) reverses it.

10.1.3 Forward evaluation

For each node v_j, let $\text{in}(v_j)$ denote the set of immediate predecessors (parents). If v_j is an operation with inputs from parents, compute

$$v_j = \text{op}_j(\{v_i : i \in \text{parents}(j)\}). \tag{10.1}$$

By topological ordering, all parents of v_j have already been computed.

10.2 Automatic Differentiation: Backpropagation in DAGs

10.2.1 Generalized chain rule

For any node v_j in the graph, the gradient w.r.t. parameter (or leaf) v_i is decomposed as a sum over all paths from v_i to the root (loss $\mathcal{L}$):

$$\frac{\partial \mathcal{L}}{\partial v_i} = \sum_{\text{paths } i \to \text{root}} \prod_{\text{edges in path}} \frac{\partial v_{\text{dest}}}{\partial v_{\text{src}}}. \tag{10.2}$$

Equivalently, using dynamic programming on the DAG: define $\bar{v}_j = \frac{\partial \mathcal{L}}{\partial v_j}$ (the adjoint or backprop error). For the root, $\bar{v}_{\text{root}} = 1$ (or $\nabla \mathcal{L}$ if loss is vectorial).

For each non-root node v_j, the adjoint is computed as

$$\bar{v}_j = \sum_{k \in \text{children}(j)} \bar{v}_k \cdot \frac{\partial v_k}{\partial v_j}. \tag{10.3}$$

This recurrence is applied in reverse topological order (from root to leaves).

10.2.2 Example: Computing adjoints for $y = (x_1 + x_2) \cdot x_1$

Continue the graph from Section 10.1.

Forward pass (already computed): Suppose $x_1 = 2, x_2 = 3$. Then $v_3 = 2 + 3 = 5$, $v_4 = 5 \cdot 2 = 10$, $y = 10$.

Backward pass (adjoints): Assume loss is $\mathcal{L} = y^2$, so $\frac{\partial \mathcal{L}}{\partial y} = 2y = 20$.

1. Initialize $\bar{v}_5 = 20$ (root adjoint).
2. $v_4 \to v_5$: edge with $\frac{\partial v_5}{\partial v_4} = 1$, so

$$\bar{v}_4 = \bar{v}_5 \cdot 1 = 20. \tag{10.4}$$

3. $v_3 \to v_4$ and $v_1 \to v_4$: edges with $\frac{\partial v_4}{\partial v_3} = v_1 = 2$ and $\frac{\partial v_4}{\partial v_1} = v_3 = 5$, so

$$\bar{v}_3 = \bar{v}_4 \cdot 2 = 40, \quad \bar{v}_1 \mathrel{+}= \bar{v}_4 \cdot 5 = 100. \tag{10.5}$$

(Note: $\bar{v}_1$ accumulates because it has multiple children.)

4. $v_1 \to v_3$ and $v_2 \to v_3$: edges with $\frac{\partial v_3}{\partial v_1} = 1$ and $\frac{\partial v_3}{\partial v_2} = 1$, so

$$\bar{v}_1 \mathrel{+}= \bar{v}_3 \cdot 1 = 40 \quad \Rightarrow \quad \bar{v}_1 = 100 + 40 = 140, \tag{10.6}$$

$$\bar{v}_2 = \bar{v}_3 \cdot 1 = 40. \tag{10.7}$$

Result: $\frac{\partial \mathcal{L}}{\partial x_1} = 140$, $\frac{\partial \mathcal{L}}{\partial x_2} = 40$.

This can be verified by hand: $\frac{\partial \mathcal{L}}{\partial x_1} = \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial y}{\partial x_1} = 20 \cdot (2x_1 + x_2) = 20(4 + 3) = 140$.

10.3 Forward Mode vs. Reverse Mode Differentiation

10.3.1 Reverse mode (backpropagation)

As described above, reverse mode (backprop) computes all adjoints via a single backward pass.

Complexity: Each edge is traversed once, and each adjoint accumulation is $O(1)$. Total cost: $O(|V| + |E|)$ where $|V|$ is node count and $|E|$ is edge count. For a feedforward network with L layers of n neurons each, this is roughly $O(L \cdot n)$.

Memory: Must store intermediate activations v_j for all nodes (for use in gradients during backward pass). For deep networks, this can be prohibitive, motivating strategies like gradient checkpointing.

10.3.2 Forward mode (tangent linear)

Forward mode traces gradients *forward* through the graph. For each leaf input x_i, compute the tangent vector $\dot{v}_j = \frac{\partial v_j}{\partial x_i}$ via a forward pass.

Recurrence: Initialize $\dot{x}_i = 1$, $\dot{x}_{i'} = 0$ for $i' \neq i$. For each node in topological order,

$$\dot{v}_j = \sum_{k \in \text{parents}(j)} \frac{\partial v_j}{\partial v_k} \dot{v}_k. \tag{10.8}$$

Complexity: One forward tangent pass computes $\frac{\partial v_j}{\partial x_i}$ for all j and *one* input i. To get all d inputs: requires d forward passes, each costing $O(|V|+|E|)$. Total: $O(d \cdot (|V|+|E|))$.

For $d \gg 1$ outputs and $\ll d$ inputs (as in supervised learning), reverse mode is vastly more efficient.

10.3.3 Comparison table

	Reverse (Backprop)	**Forward (Tangent)**
One gradient pass cost	$O(\|V\| + \|E\|)$	$O(\|V\| + \|E\|)$
For m outputs, n inputs	$O(m \cdot (\|V\| + \|E\|))$	$O(n \cdot (\|V\| + \|E\|))$
Best for	$n \gg m$ (typical learning)	$m \gg n$ (rare in learning)
Memory	$O(n_{\max}^{(\ell)})$ during backward	$O(n_{\max}^{(\ell)})$ during forward

In neural network training, $m = 1$ (scalar loss) and n is number of parameters (millions to billions), so reverse mode is standard.

10.4 Chain Rule in Multivariate Form

For nodes with vector/matrix values, the chain rule uses careful indexing.

10.4.1 Jacobian-vector products

If $v_k : \mathbb{R}^a \to \mathbb{R}^b$ (a node taking a-dimensional input, producing b-dimensional output), and loss is scalar, then

$$\frac{\partial \mathcal{L}}{\partial v_{k,i}} = \sum_{j=1}^{b} \frac{\partial \mathcal{L}}{\partial v_{k,j}^{\text{out}}} \cdot \frac{\partial v_{k,j}^{\text{out}}}{\partial v_{k,i}}. \tag{10.9}$$

In matrix notation, if $v_k^{\text{in}} \in \mathbb{R}^a$, $v_k^{\text{out}} \in \mathbb{R}^b$, and $J_k \in \mathbb{R}^{b \times a}$ is the Jacobian, then

$$\overline{v_k^{\text{in}}} = J_k^\top \, \overline{v_k^{\text{out}}}. \tag{10.10}$$

10.4.2 Example: Softmax backward

Softmax node: input $z \in \mathbb{R}^K$, output $p \in \mathbb{R}^K$ with $p_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$.

The Jacobian (from Chapter 4) is

$$J_{ij} = \frac{\partial p_i}{\partial z_j} = p_i(\delta_{ij} - p_j). \tag{10.11}$$

If the upstream loss adjoint is $\bar{p} \in \mathbb{R}^K$, then

$$\bar{z} = J^\top \bar{p} = \begin{bmatrix} p_1(\bar{p}_1 - \bar{p}^\top p) \\ \vdots \\ p_K(\bar{p}_K - \bar{p}^\top p) \end{bmatrix}. \tag{10.12}$$

In the special case of cross-entropy loss where $\bar{p}_i = p_i - y_i$ (from Chapter 5), we get $\bar{z}_i = p_i - y_i$, matching our earlier result.

Chapter 11

Numerical Stability and Precision

Neural networks require careful numerical handling, especially in loss computation and gradient flow.

11.1 Softmax and the Log-Sum-Exp Trick

11.1.1 Naive softmax (numerically unstable)

Computing $\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$ directly can overflow: if z_i is large (e.g., $z_i = 1000$), then e^{z_i} exceeds floating-point range.

11.1.2 Stable variant (log-sum-exp)

Subtract the max before exponentiating:

$$z_i' = z_i - \max_k z_k, \tag{11.1}$$

then compute

$$p_i = \frac{e^{z_i'}}{\sum_j e^{z_j'}}. \tag{11.2}$$

Now $z_i' \leq 0$ for all i, so $e^{z_i'} \in (0, 1]$, avoiding overflow.

Mathematically:

$$\frac{e^{z_i}}{\sum_j e^{z_j}} = \frac{e^{z_i - \max z}}{\sum_j e^{z_j - \max z}}. \tag{11.3}$$

11.1.3 Log-domain computation

For numerical stability in probability computations, work in log space:

$$\log p_i = z_i' - \log\left(\sum_j e^{z_j'}\right) = z_i' - \text{logsumexp}(z'). \tag{11.4}$$

This is especially useful when computing cross-entropy:

$$\text{CCE} = -\sum_k y_k \log p_k = -\sum_k y_k (z_k' - \text{logsumexp}(z')). \tag{11.5}$$

11.2 Underflow and Overflow in Deep Networks

11.2.1 Activation norms

In very deep networks, activations can grow or shrink exponentially layer by layer. If $|z^{(\ell)}| \to 0$ (underflow), gradients vanish. If $|z^{(\ell)}| \to \infty$ (overflow), parameters become NaN.

11.2.2 Initialization and gradient norms

Careful initialization (e.g., He initialization for ReLU, Xavier for sigmoid) helps maintain reasonable activation magnitudes:

$$\mathbf{W}_{ij}^{(\ell)} \sim \mathcal{N}\left(0, \frac{2}{n_{\ell-1}}\right) \quad \text{(He)} \tag{11.6}$$

ensures that $\mathbb{E}[|z^{(\ell)}|^2] \approx \mathbb{E}[|z^{(\ell-1)}|^2]$.

11.2.3 Gradient clipping

To prevent exploding gradients, clip by norm:

$$g \leftarrow g \cdot \min\left(1, \frac{C}{\|g\|_2}\right), \tag{11.7}$$

where C is a threshold (e.g., $C = 1$). This keeps gradients bounded during backprop, especially important for RNNs.

11.3 Mixed Precision Training

Modern hardware (GPUs, TPUs) supports low-precision floating point (e.g., FP16: 16-bit) at much higher speed than full precision (FP32: 32-bit).

11.3.1 Strategy

1. Perform forward pass in FP16 (fast).
2. Compute loss in FP16 (or reduced precision).
3. *Scale* the loss by a large factor L_{scale} (e.g., 2^{15}):
$$\hat{\mathcal{L}} = L_{\text{scale}} \cdot \mathcal{L}. \tag{11.8}$$
4. Backprop through scaled loss (gradients are also scaled up, reducing underflow risk).
5. Perform weight update in FP32, with gradients scaled down by L_{scale}.

11.3.2 Why scaling helps

FP16 has range roughly $[6 \times 10^{-5}, 6 \times 10^{4}]$. Typical gradients are small ($\sim 10^{-3}$ to 10^{-5}); without scaling, they underflow to zero in FP16. Scaling before backprop keeps gradients in the representable range; then downscaling recovers the true gradient for the update.

Chapter 12

Tensor Operations and Notation

Modern neural networks, especially CNNs and Transformers, manipulate high-dimensional arrays (tensors). This chapter formalizes tensor operations and notation.

12.1 Tensors and Index Notation

12.1.1 Definition

An n-th order tensor $T \in \mathbb{R}^{d_1 \times \cdots \times d_n}$ is a multi-dimensional array.

- Order 0: scalar.
- Order 1: vector.
- Order 2: matrix.
- Order 3+: higher-order tensors.

Element indexing: $T[i_1, i_2, \ldots, i_n]$ or $T_{i_1 i_2 \cdots i_n}$.

12.1.2 Einstein notation (summation convention)

In Einstein notation, repeated indices imply summation:

$$C_{ij} = \sum_k A_{ik} B_{kj} \quad \text{is written as} \quad C_{ij} = A_{ik} B_{kj}. \tag{12.1}$$

Implicit indices (not repeated) are free indices; repeated indices are contracted (summed over).

Example: Matrix-vector product

$$y_i = \sum_j W_{ij} x_j \quad \Rightarrow \quad y_i = W_{ij} x_j. \tag{12.2}$$

Example: Convolution (1D, single sample)

Input sequence x_t (length T), filter w_s (length S), stride 1:

$$y_t = \sum_{s=0}^{S-1} w_s x_{t+s} \quad \Rightarrow \quad y_t = w_s x_{t+s}. \tag{12.3}$$

Here t is the free (output) index, s is contracted.

Example: Batched matrix multiplication

Batch size B, $A \in \mathbb{R}^{B\times M\times K}$, $B \in \mathbb{R}^{B\times K\times N}$:

$$C_{b,m,n} = \sum_k A_{b,m,k} B_{b,k,n} \quad \Rightarrow \quad C_{bmn} = A_{bmk} B_{bkn}. \tag{12.4}$$

12.2 Broadcasting and Element-wise Operations

12.2.1 Broadcasting rules (NumPy/PyTorch convention)

When operating on tensors of different shapes, dimensions are aligned from the right. Missing dimensions are inserted on the left.

Example 1: Shape (M, N) and shape $(N,)$: expand $(N,)$ to $(1, N)$, then broadcast to (M, N).

Example 2: Shape (B, M, N) and shape $(N,)$: expand to $(1, 1, N)$, broadcast to (B, M, N).

Element-wise scaling:

$$Y_{b,m,n} = X_{b,m,n} \cdot \gamma_n \tag{12.5}$$

where γ is shape $(N,)$. This is a common pattern in layer normalization and bias addition.

12.3 Reshape and Transpose

12.3.1 Reshape (view)

Changing shape without reordering data:

$$X \in \mathbb{R}^{B\times C\times H\times W} \to X' \in \mathbb{R}^{BC\times HW}. \tag{12.6}$$

Data layout matters: reshape assumes row-major (C-contiguous) or column-major (Fortran-contiguous) memory order.

12.3.2 Transpose (permutation)

Reorder dimensions:

$$Y_{i,j,k,\ell} = X_{k,i,\ell,j} \quad \text{corresponds to} \quad \text{permute}((0,1,2,3) \to (2,0,3,1)). \tag{12.7}$$

In Einstein notation:

$$Y_{ijkl} = X_{kilj}. \tag{12.8}$$

12.3.3 Flattening (vectorization)

Combining batch and spatial dimensions:

$$X \in \mathbb{R}^{B\times C\times H\times W} \to \vec{X} \in \mathbb{R}^{B\cdot C\cdot H\cdot W}. \tag{12.9}$$

Useful for fully connected layers following convolutional layers.

Chapter 13

Hyperparameter Tuning and Learning Rate Schedules

Training hyperparameters (learning rate, momentum, batch size, etc.) dramatically affect convergence and generalization. This chapter covers principled tuning strategies.

13.1 Learning Rate Selection

13.1.1 Learning rate finder (LRFinder)

A practical heuristic (Fastai, PyTorch Lightning):

1. Start with a small learning rate $\eta_{\min}$ (e.g., 10^{-5}).
2. Train for one epoch, exponentially increasing η at each batch:

$$\eta_t = \eta_{\min} \cdot \left(\frac{\eta_{\max}}{\eta_{\min}} \right)^{t/T}, \tag{13.1}$$

   where T is total batches, $\eta_{\max}$ is max rate.
3. Track loss vs. η.
4. Select η where loss is still decreasing steeply but not yet diverging.

Why it works: Identifies the "sweet spot" where the loss landscape has largest gradient (learning is efficient) without being at risk of divergence.

13.1.2 Learning rate schedules

After choosing a base learning rate, reduce it over time:

Step decay:

$$\eta_t = \eta_0 \cdot \gamma^{\lfloor t/S \rfloor}, \tag{13.2}$$

where $\gamma < 1$ (e.g., 0.1) and S is step size (epochs between drops).

Exponential decay:

$$\eta_t = \eta_0 \cdot e^{-\lambda t}, \tag{13.3}$$

where $\lambda > 0$ is decay rate.

Cosine annealing:

$$\eta_t = \eta_{\min} + \frac{\eta_0 - \eta_{\min}}{2}\left(1 + \cos\frac{\pi t}{T}\right), \tag{13.4}$$

where T is total iterations. Smoothly decreases from η_0 to $\eta_{\min}$.

Cosine with warm restarts (SGDR): Reset the cosine schedule multiple times, with decreasing max learning rate:

$$\eta_t^{(i)} = \eta_{\min} + \frac{\eta_0 \cdot \gamma^i - \eta_{\min}}{2}\left(1 + \cos\frac{\pi(t - t_i)}{T_i}\right), \tag{13.5}$$

where t_i marks the start of restart i, T_i is its period.

13.2 Warmup

13.2.1 Linear warmup

Start with very small learning rate, linearly increase to target:

$$\eta_t = \eta_{\min} + \frac{\eta_0 - \eta_{\min}}{T_{\text{warm}}} \cdot t, \quad t \in [0, T_{\text{warm}}]. \tag{13.6}$$

After T_{warm} iterations, switch to standard schedule.

Why: Prevents extreme parameter updates early in training when initialization is random and gradients are unreliable. Especially important for Transformers and other large models.

13.2.2 Gradient accumulation + warmup

With gradient accumulation (computing loss on mini-batches, accumulating gradients, updating after k mini-batches), warmup typically spans the accumulated steps:

$$\eta_t = \eta_{\min} + \frac{\eta_0 - \eta_{\min}}{k \cdot T_{\text{warm}}} \cdot t. \tag{13.7}$$

13.3 Hyperparameter Search Methods

13.3.1 Grid search

Enumerate all combinations of discrete values:

$$(\eta, \beta, \lambda) \in \{\eta_1, \ldots, \eta_m\} \times \{\beta_1, \ldots, \beta_n\} \times \{\lambda_1, \ldots, \lambda_p\}. \tag{13.8}$$

Train $m \cdot n \cdot p$ models, select the best.

Disadvantage: Combinatorial explosion; many hyperparameters become infeasible.

13.3.2 Random search

Sample hyperparameters uniformly (or from a prior) for N trials:

$$(\eta^{(i)}, \beta^{(i)}, \lambda^{(i)}) \sim p(\eta, \beta, \lambda), \quad i = 1, \ldots, N. \tag{13.9}$$

Train N models, select best.

Advantage: More efficient than grid search in high dimensions; discovers good regions faster.

13.3.3 Bayesian optimization

Model the objective (e.g., validation loss) as a Gaussian process:

$$f(\mathbf{h}) \sim \mathcal{GP}(\mu(\mathbf{h}), k(\mathbf{h}, \mathbf{h}')) \tag{13.10}$$

where $\mathbf{h}$ is the hyperparameter vector.

At each iteration:

1. Fit GP to observed trials.
2. Define an acquisition function (e.g., Expected Improvement) balancing exploration and exploitation.
3. Select next hyperparameters to maximize acquisition.
4. Train and observe outcome.
5. Repeat.

Complexity: Higher computational cost per iteration (fitting GP) but fewer total trials needed.

Chapter 14

Data Preprocessing and Normalization

Before training, data and intermediate activations should be normalized for stability and convergence.

14.1 Input Normalization

14.1.1 Standardization (Z-score)

Center and scale each feature:

$$x_i' = \frac{x_i - \mu_i}{\sigma_i}, \tag{14.1}$$

where $\mu_i = \frac{1}{n}\sum_{j=1}^{n} x_{ij}$, $\sigma_i = \sqrt{\frac{1}{n}\sum_{j=1}^{n}(x_{ij} - \mu_i)^2}$.

Assumption: Features are roughly normally distributed.

14.1.2 Min-Max scaling

Scale to fixed range (e.g., $[0, 1]$):

$$x_i' = \frac{x_i - \min_j x_{ij}}{\max_j x_{ij} - \min_j x_{ij}}. \tag{14.2}$$

Use: When you want bounded values; sensitive to outliers.

14.1.3 Data statistics (train vs. test)

Compute μ, σ (or min, max) on training data. Apply the same transformation to test data. **Never** compute statistics on test data; this leaks test information into the model.

14.2 Batch Normalization (Revisited)

From Chapter 7/8, batch normalization normalizes layer inputs within mini-batches:

$$\hat{z}_i^{(\ell)} = \frac{z_i^{(\ell)} - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}}, \quad y_i^{(\ell)} = \gamma \hat{z}_i^{(\ell)} + \beta. \tag{14.3}$$

14.2.1 Running mean and variance (inference)

During training, use batch statistics. During inference, use a running average computed across training:

$$\mu_{\text{run}} \leftarrow \alpha \mu_{\text{run}} + (1-\alpha)\mu_B, \quad \sigma^2_{\text{run}} \leftarrow \alpha \sigma^2_{\text{run}} + (1-\alpha)\sigma_B^2, \tag{14.4}$$

where $\alpha \approx 0.9$ or 0.99 (momentum).

14.3 Layer Normalization

Layer normalization (LayerNorm) computes statistics per sample and layer, not per batch. For a layer input $z^{(\ell)} \in \mathbb{R}^{n_\ell}$ (single sample):

$$\mu^{(\ell)} = \frac{1}{n_\ell} \sum_{i=1}^{n_\ell} z_i^{(\ell)}, \quad \sigma^{(\ell),2} = \frac{1}{n_\ell} \sum_{i=1}^{n_\ell} (z_i^{(\ell)} - \mu^{(\ell)})^2, \tag{14.5}$$

$$\hat{z}_i^{(\ell)} = \frac{z_i^{(\ell)} - \mu^{(\ell)}}{\sqrt{\sigma^{(\ell),2} + \varepsilon}}, \quad y_i^{(\ell)} = \gamma_i \hat{z}_i^{(\ell)} + \beta_i. \tag{14.6}$$

Advantage: Not dependent on batch statistics; works well with small batches, RNNs, and Transformers. Learnable scale γ and shift β are usually per-feature (size n_ℓ).

14.4 Group Normalization and Instance Normalization

14.4.1 Group normalization

Divide channels into G groups, normalize within each group:

$$\mu^{(g)} = \frac{1}{S/G} \sum_{s \in \text{group } g} z_s, \quad \sigma^{(g),2} = \frac{1}{S/G} \sum_{s \in \text{group } g} (z_s - \mu^{(g)})^2, \tag{14.7}$$

where $S = C \cdot H \cdot W$ (total features per sample).

Use: Works well when batch size is small (e.g., 1–4).

14.4.2 Instance normalization

Normalize each feature map (channel) independently:

$$\mu^{(c)} = \frac{1}{H \cdot W} \sum_{h,w} z_{c,h,w}, \quad \sigma^{(c),2} = \frac{1}{H \cdot W} \sum_{h,w} (z_{c,h,w} - \mu^{(c)})^2. \tag{14.8}$$

Use: Style transfer, image generation. Normalizes per-instance statistics, removing instance-specific information.

14.5 Comparison of Normalization Methods

Chapter 15

Recurrent Neural Networks (RNNs)

15.1 Sequence Data and Mathematical Formulation

15.1.1 Temporal Data Representation

For sequence data, we denote:

- Input sequence: $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(T)}$ where each $\mathbf{x}^{(t)} \in \mathbb{R}^{d_x}$
- Target sequence: $\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \ldots, \mathbf{y}^{(T)}$ (for many-to-many tasks)
- Sequence length T may vary across examples

Unlike feedforward networks, RNNs process sequences one timestep at a time, maintaining an internal state.

15.1.2 Notation and Convention

Let:

- $\mathbf{h}^{(t)} \in \mathbb{R}^{d_h}$ be the hidden state at time t
- d_h be the hidden dimension
- $\mathbf{h}^{(0)} = \mathbf{0}$ (zero initialization)

For mini-batch processing with m sequences stacked as columns:

$$\mathbf{H}^{(t)} = [\mathbf{h}^{(t,1)}, \mathbf{h}^{(t,2)}, \ldots, \mathbf{h}^{(t,m)}] \in \mathbb{R}^{d_h \times m} \tag{15.1}$$

15.1.3 Common Task Architectures

Many-to-one (e.g., sentiment classification):

- Input: $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T)}$
- Output: single $\hat{\mathbf{y}}$ from final hidden state

One-to-many (e.g., image captioning):

- Input: single $\mathbf{x}$

- Output: $\hat{\mathbf{y}}^{(1)}, \ldots, \hat{\mathbf{y}}^{(T')}$

Many-to-many (e.g., machine translation):

- Input sequence: $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(T_{\text{in}})}$
- Output sequence: $\hat{\mathbf{y}}^{(1)}, \ldots, \hat{\mathbf{y}}^{(T_{\text{out}})}$

15.2 Vanilla RNN Definition and Forward Propagation

15.2.1 Recurrent Computation

At each timestep $t = 1, 2, \ldots, T$, the Vanilla RNN computes:

$$\mathbf{z}_h^{(t)} = \mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \mathbf{W}_{xh}\mathbf{x}^{(t)} + \mathbf{b}_h \tag{15.2}$$

$$\mathbf{h}^{(t)} = \sigma_h(\mathbf{z}_h^{(t)}) \tag{15.3}$$

$$\mathbf{z}_y^{(t)} = \mathbf{W}_{hy}\mathbf{h}^{(t)} + \mathbf{b}_y \tag{15.4}$$

$$\hat{\mathbf{y}}^{(t)} = \sigma_y(\mathbf{z}_y^{(t)}) \tag{15.5}$$

where:

- $\mathbf{W}_{hh} \in \mathbb{R}^{d_h \times d_h}$ (hidden-to-hidden weights)
- $\mathbf{W}_{xh} \in \mathbb{R}^{d_h \times d_x}$ (input-to-hidden weights)
- $\mathbf{W}_{hy} \in \mathbb{R}^{d_y \times d_h}$ (hidden-to-output weights)
- σ_h is typically tanh or ReLU
- σ_y depends on the task (softmax for classification, sigmoid for binary, linear for regression)

15.2.2 Parameter Sharing Across Time

The key insight of RNNs is **parameter sharing**: the same parameters ($\mathbf{W}_{hh}, \mathbf{W}_{xh}, \mathbf{W}_{hy}, \mathbf{b}_h, \mathbf{b}_y$) are used at every timestep. This is why the model can handle variable-length sequences.

15.2.3 Vectorized Mini-batch Forward Pass

For a mini-batch, stack hidden states and inputs as matrices:

$$\mathbf{Z}_h^{(t)} = \mathbf{W}_{hh}\mathbf{H}^{(t-1)} + \mathbf{W}_{xh}\mathbf{X}^{(t)} + \mathbf{b}_h\mathbf{1}^\top \tag{15.6}$$

$$\mathbf{H}^{(t)} = \sigma_h(\mathbf{Z}_h^{(t)}) \tag{15.7}$$

$$\mathbf{Z}_y^{(t)} = \mathbf{W}_{hy}\mathbf{H}^{(t)} + \mathbf{b}_y\mathbf{1}^\top \tag{15.8}$$

$$\hat{\mathbf{Y}}^{(t)} = \sigma_y(\mathbf{Z}_y^{(t)}) \tag{15.9}$$

where $\mathbf{1} \in \mathbb{R}^m$ is the all-ones vector.

15.3 Computational Graph Unrolling in Time

15.3.1 Unrolled Graph Representation

When we unfold the RNN across T timesteps, we obtain a computational graph that is a directed acyclic graph (DAG):

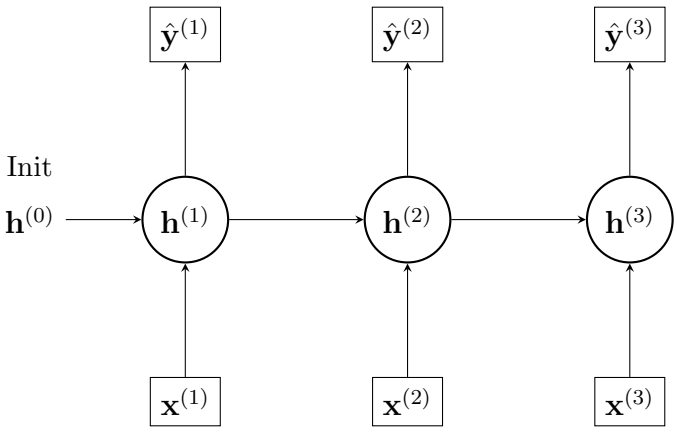

Figure 15.1: Unrolled Recurrent Neural Network. The hidden state $\mathbf{h}^{(t)}$ passes information to the next timestep. (Adapted from Goodfellow et al., 2016)

Each "RNN cell" at time t depends on:

1. Current input $\mathbf{x}^{(t)}$
2. Previous hidden state $\mathbf{h}^{(t-1)}$

15.3.2 Temporal Dependencies

The hidden state $\mathbf{h}^{(t)}$ depends on all previous inputs:

$$\mathbf{h}^{(t)} = f_t(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(t)}) \tag{15.10}$$

This creates a long chain of dependencies, which will be crucial for understanding gradient flow.

15.4 Backpropagation Through Time (BPTT)

15.4.1 Loss Function and Objective

For a sequence, the total loss is:

$$L = \sum_{t=1}^{T} L^{(t)} \tag{15.11}$$

where $L^{(t)} = \ell(\hat{\mathbf{y}}^{(t)}, \mathbf{y}^{(t)})$ is the loss at timestep t (e.g., cross-entropy for classification).

15.4.2 Backpropagation Through Time Algorithm

The key insight is that the gradient at each timestep comes from two sources:

$$\frac{\partial L}{\partial \mathbf{h}^{(t)}} = \frac{\partial L^{(t)}}{\partial \mathbf{h}^{(t)}} + \frac{\partial L^{(t+1:T)}}{\partial \mathbf{h}^{(t)}} \tag{15.12}$$

Derivation:
By the chain rule and the flow of gradients from the computational graph:

$$\frac{\partial L^{(t+1:T)}}{\partial \mathbf{h}^{(t)}} = \frac{\partial L^{(t+1:T)}}{\partial \mathbf{h}^{(t+1)}} \frac{\partial \mathbf{h}^{(t+1)}}{\partial \mathbf{h}^{(t)}} \tag{15.13}$$

Let $\boldsymbol{\delta}_h^{(t)} = \frac{\partial L}{\partial \mathbf{z}_h^{(t)}}$ be the delta (error signal) at the hidden layer pre-activation.
From the output layer:

$$\frac{\partial L^{(t)}}{\partial \mathbf{h}^{(t)}} = \mathbf{W}_{hy}^{\top} \frac{\partial L^{(t)}}{\partial \mathbf{z}_y^{(t)}} \tag{15.14}$$

From the next timestep (via the recurrent connection):

$$\frac{\partial L^{(t+1:T)}}{\partial \mathbf{h}^{(t)}} = \mathbf{W}_{hh}^{\top} \boldsymbol{\delta}_h^{(t+1)} \tag{15.15}$$

Therefore:

$$\boldsymbol{\delta}_h^{(t)} = \left(\mathbf{W}_{hy}^{\top} \boldsymbol{\delta}_y^{(t)} + \mathbf{W}_{hh}^{\top} \boldsymbol{\delta}_h^{(t+1)} \right) \odot \sigma_h'(\mathbf{z}_h^{(t)}) \tag{15.16}$$

where $\boldsymbol{\delta}_y^{(t)} = \frac{\partial L^{(t)}}{\partial \mathbf{z}_y^{(t)}}$ is the output layer delta.

15.4.3 Parameter Gradients

The gradients for the weight matrices are computed by summing contributions from all timesteps:

$$\frac{\partial L}{\partial \mathbf{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial L}{\partial \mathbf{z}_h^{(t)}} \frac{\partial \mathbf{z}_h^{(t)}}{\partial \mathbf{W}_{hh}} \tag{15.17}$$

$$= \sum_{t=1}^{T} \boldsymbol{\delta}_h^{(t)} (\mathbf{h}^{(t-1)})^{\top} \tag{15.18}$$

Similarly:

$$\frac{\partial L}{\partial \mathbf{W}_{xh}} = \sum_{t=1}^{T} \boldsymbol{\delta}_h^{(t)} (\mathbf{x}^{(t)})^\top \tag{15.19}$$

$$\frac{\partial L}{\partial \mathbf{W}_{hy}} = \sum_{t=1}^{T} \boldsymbol{\delta}_y^{(t)} (\mathbf{h}^{(t)})^\top \tag{15.20}$$

For biases:

$$\frac{\partial L}{\partial \mathbf{b}_h} = \sum_{t=1}^{T} \boldsymbol{\delta}_h^{(t)} \tag{15.21}$$

$$\frac{\partial L}{\partial \mathbf{b}_y} = \sum_{t=1}^{T} \boldsymbol{\delta}_y^{(t)} \tag{15.22}$$

15.5 Vanishing and Exploding Gradients: Mathematical Analysis

15.5.1 Gradient Flow Through Hidden States

The gradient of the loss with respect to a distant hidden state $\mathbf{h}^{(k)}$ (where $k < t$) involves a product of Jacobians:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \prod_{j=k+1}^{t} \frac{\partial \mathbf{h}^{(j)}}{\partial \mathbf{h}^{(j-1)}} \tag{15.23}$$

Derivation:
By the chain rule:

$$\frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(k)}} = \frac{\partial \mathbf{h}^{(t)}}{\partial \mathbf{h}^{(t-1)}} \frac{\partial \mathbf{h}^{(t-1)}}{\partial \mathbf{h}^{(t-2)}} \cdots \frac{\partial \mathbf{h}^{(k+1)}}{\partial \mathbf{h}^{(k)}} \tag{15.24}$$

Each Jacobian $\frac{\partial \mathbf{h}^{(j)}}{\partial \mathbf{h}^{(j-1)}}$ has the form:

$$\frac{\partial \mathbf{h}^{(j)}}{\partial \mathbf{h}^{(j-1)}} = \frac{\partial \sigma_h(\mathbf{z}_h^{(j)})}{\partial \mathbf{z}_h^{(j)}} \frac{\partial \mathbf{z}_h^{(j)}}{\partial \mathbf{h}^{(j-1)}} \tag{15.25}$$

$$= \operatorname{diag}(\sigma_h'(\mathbf{z}_h^{(j)})) \mathbf{W}_{hh} \tag{15.26}$$

15.5.2 Spectral Analysis

For stability, we analyze the spectral properties. The product of Jacobians can be approximated by:

$$\left\| \prod_{j=k+1}^{t} \frac{\partial \mathbf{h}^{(j)}}{\partial \mathbf{h}^{(j-1)}} \right\| \lesssim \|\sigma_h'\|_\infty^{t-k} \|\mathbf{W}_{hh}\|^{t-k} \tag{15.27}$$

For tanh activation, $|\sigma_h'(z)| \leq 1$ for all z, and the maximum is $1/4$ at $z = 0$.
Let $\rho = \lambda_{\max}(\mathbf{W}_{hh})$ be the spectral radius (largest eigenvalue magnitude).

- **Vanishing gradients:** If $\rho < 1$, then $\|\mathbf{W}_{hh}\|^{t-k} \to 0$ as $t - k \to \infty$.

  - Consequence: Gradients for distant timesteps become negligible.
  - Learning long-term dependencies becomes slow.

- **Exploding gradients:** If $\rho > 1$, then $\|\mathbf{W}_{hh}\|^{t-k} \to \infty$ as $t - k \to \infty$.
  - Consequence: Gradients become unbounded; training becomes unstable.
  - Parameter updates may be very large, causing divergence.

15.5.3 Mathematical Condition for Stability

Define the **temporal condition number**:

$$\kappa_T = \rho^T \tag{15.28}$$

For long sequences (T large):

- If $\rho < 1$: exponential decay of $\kappa_T \to 0$ (vanishing)
- If $\rho > 1$: exponential growth of $\kappa_T \to \infty$ (exploding)
- If $\rho = 1$: $\kappa_T = 1$ (critically balanced, unstable in practice)

15.6 Common Questions (RNNs)

15.6.1 Q1: Why do we share $\mathbf{W}_{hh}$?

A: Parameter sharing allows the RNN to apply the same "rule" regardless of the sequence length.

Example:

- **Without sharing**: A sequence of length 100 and length 1000 would require different networks (different parameters).
- **With sharing**: The same $\mathbf{W}_{hh}$ is used from time 1 to 100, and from 100 to 1000.

Mathematical view:

$$\mathbf{h}^{(T)} = \sigma(\mathbf{W}_{hh} \cdots \sigma(\mathbf{W}_{hh}\mathbf{h}^{(1)})) \tag{15.29}$$

By applying $\mathbf{W}_{hh}$ repeatedly, the model can handle **variable-length sequences**.

Trade-off: Gradients become a long product of $\mathbf{W}_{hh}$, making them prone to vanishing/exploding.

15.6.2 Q2: What exactly is "vanishing gradient"?

A: It is a state where parameter updates become nearly zero, stopping the model from learning.

Numerical example:

- Processing a 100-step sentence with Vanilla RNN.
- $|\sigma'| \approx 0.5$ at each step (moderate gradient for tanh).

- Gradient magnitude: $0.5^{100} \approx 10^{-30}$ (nearly zero!)

Practical impact:

- The influence of the 1st word on the 100th output becomes unmeasurable.
- The model relies only on "recent words" and cannot learn long-term dependencies.

Example: Translation task

```
``The quick brown fox jumps over the lazy dogs are \_\_\_''
        ^ Subject (long distance)          ^ Predicate (end)
```

The RNN tries to complete "dogs are" using only local context, missing the singular/-plural agreement with "fox".

15.6.3 Q3: What is the computational cost of BPTT?

A: Full BPTT requires storing states for all timesteps, which is memory-inefficient.
Memory usage:

- Sequence length $T = 1000$, Hidden dim $d_h = 512$, Float32 (4 bytes).
- **Memory required**: $1000 \times 512 \times 4 = 2.048$ MB (per sequence).
- **Batch size 32**: 65.5 MB.

Since data must be kept for gradient computation across all steps, it is memory-heavy.
Solution: Truncated BPTT ($\tau \approx 50$), considering only the past 50 steps.

15.6.4 Q4: Why can't RNNs be parallelized?

A: Because $\mathbf{h}^{(t)}$ depends on $\mathbf{h}^{(t-1)}$, calculations must respect chronological order.
Dependency graph:

```
h(1) -> h(2) -> h(3) -> h(4)
 |       |       |       |
x(1)    x(2)    x(3)    x(4)
```

Computation time:

- RNN: $O(T)$ (Sequential).
- Transformer: $O(\log T)$ or $O(1)$ (Parallelizable).

In LLMs, parallel efficiency dictates training speed, giving Transformers a huge advantage.

15.7 Common Questions (Gradient Problems)

15.7.1 Q5: What is the danger of exploding gradients?

A: Updates become massive, causing parameters to overshoot the optimal solution.

Numerical example:

```
# Exploding gradient
gradient = 1e8
learning_rate = 0.001
weight_update = 100000  # Massive update!

# Normal
gradient = 1.0
weight_update = 0.001   # Stable
```

Impact on Loss Curve: Instead of converging, the loss oscillates wildly or diverges to NaN.

Solution:

1. **Gradient Clipping**: $\mathbf{g} \leftarrow \theta \frac{\mathbf{g}}{\|\mathbf{g}\|}$ if $\|\mathbf{g}\| > \theta$.
2. **Weight Initialization**: Keep $\|\mathbf{W}_{hh}\|$ small.

15.7.2 Q6: Why is the spectral radius important?

A: The largest eigenvalue ρ of $\mathbf{W}_{hh}$ determines the rate of gradient growth/decay.

Intuition:

- $\rho = 0.9$: Gradient $\approx 0.9^{100} \approx 0$ (Vanishing).
- $\rho = 1.0$: Gradient ≈ 1 (Critical boundary).
- $\rho = 1.1$: Gradient $\approx 1.1^{100} \approx 14000$ (Exploding).

Implication: Initialize weights such that the spectral radius is reasonable (e.g., Xavier initialization, Orthogonal initialization).

Chapter 16

Long Short-Term Memory (LSTM)

16.1 Motivation and Design Principles

16.1.1 The Problem with Vanilla RNNs

The core issue is that gradients must flow through a chain of matrix multiplications:

$$\frac{\partial L}{\partial \mathbf{h}^{(1)}} \propto \prod_{t=2}^{T} \mathbf{W}_{hh}^{\top} \mathrm{diag}(\sigma_h') \tag{16.1}$$

With $\sigma_h = \tanh$:

- At the peak, $|\sigma_h'(0)| = 1$
- Away from the peak, $|\sigma_h'(z)| < 1$, often ≈ 0.1 to 0.01

For a 100-step sequence, even with $|\sigma_h'| \approx 0.9$ at each step:

$$0.9^{100} \approx 2.7 \times 10^{-5} \tag{16.2}$$

The gradient vanishes severely.

16.1.2 The LSTM Solution: Additive State Update

Instead of a multiplicative update $\mathbf{h}^{(t)} = \sigma_h(\mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \ldots)$, LSTM uses an **additive state update**:

$$\mathbf{c}^{(t)} = \mathbf{c}^{(t-1)} + (\text{something new}) \tag{16.3}$$

where $\mathbf{c}^{(t)}$ is the **cell state** (internal memory).
Key insight: The gradient flowing through the cell state is:

$$\frac{\partial \mathbf{c}^{(t)}}{\partial \mathbf{c}^{(t-1)}} = \mathbf{f}^{(t)} \tag{16.4}$$

where $\mathbf{f}^{(t)}$ is the **forget gate**. If $\mathbf{f}^{(t)} \approx 1$, then:

$$\prod_{j=k}^{t} \frac{\partial \mathbf{c}^{(j)}}{\partial \mathbf{c}^{(j-1)}} = \prod_{j=k}^{t} \mathbf{f}^{(j)} \approx 1 \tag{16.5}$$

The product remains stable (close to 1), preventing vanishing gradients!

16.2 Complete LSTM Cell Definition

16.2.1 Gate Computations

Forget Gate:

$$\mathbf{f}^{(t)} = \sigma(\mathbf{W}_{xf}\mathbf{x}^{(t)} + \mathbf{W}_{hf}\mathbf{h}^{(t-1)} + \mathbf{b}_f) \tag{16.6}$$

where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid (output in $[0, 1]$).

Input Gate:

$$\mathbf{i}^{(t)} = \sigma(\mathbf{W}_{xi}\mathbf{x}^{(t)} + \mathbf{W}_{hi}\mathbf{h}^{(t-1)} + \mathbf{b}_i) \tag{16.7}$$

Output Gate:

$$\mathbf{o}^{(t)} = \sigma(\mathbf{W}_{xo}\mathbf{x}^{(t)} + \mathbf{W}_{ho}\mathbf{h}^{(t-1)} + \mathbf{b}_o) \tag{16.8}$$

Cell State Candidate (Tanh pre-activation):

$$\tilde{\mathbf{c}}^{(t)} = \tanh(\mathbf{W}_{xc}\mathbf{x}^{(t)} + \mathbf{W}_{hc}\mathbf{h}^{(t-1)} + \mathbf{b}_c) \tag{16.9}$$

16.2.2 State and Hidden State Updates

Cell State Update (additive, the crucial part):

$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \odot \tilde{\mathbf{c}}^{(t)} \tag{16.10}$$

where $\odot$ denotes element-wise multiplication.

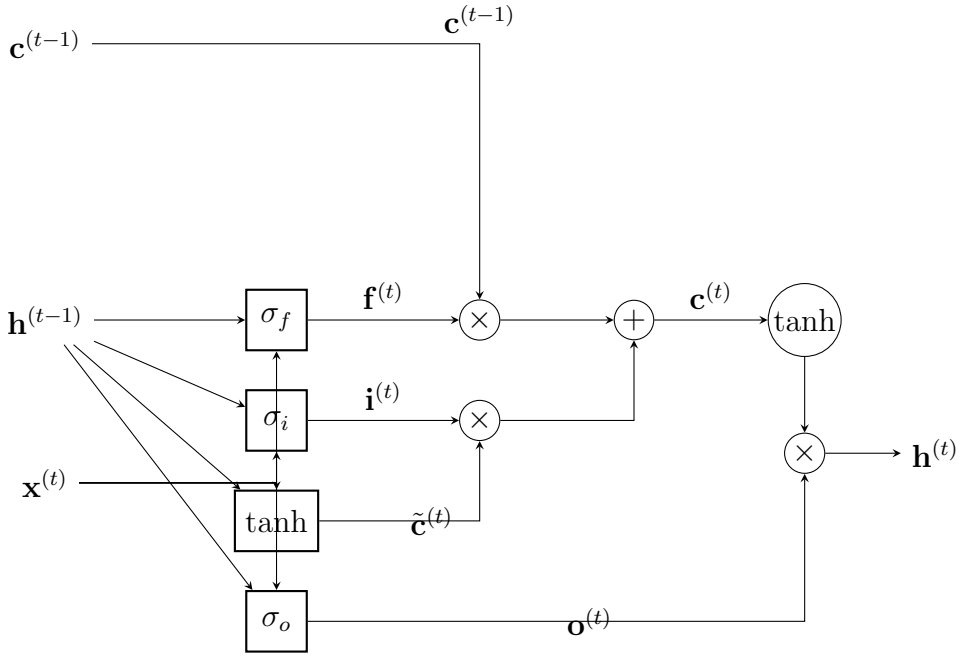

Figure 16.1: Structure of the LSTM Cell. The cell state $\mathbf{c}$ runs along the top "highway," interacting linearly via the forget and input gates. (Adapted from Olah, 2015)

Hidden State Output:

$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \odot \tanh(\mathbf{c}^{(t)}) \tag{16.11}$$

16.3 Conceptual Intuition: The Notebook Analogy

The roles of the LSTM gates (c, f, i, o) can be naturally understood by thinking of them as **three operations on a notebook** (the cell state).

1. Think of c as the "Internal Notebook"

- $\mathbf{c}^{(t)}$ is the **Cell State**: A notebook that maintains important information over a long period.
- The gates (f, i, o) are strictly **verbs (operations)** that control this notebook.

2. The Three Operations (Verbs)

Memorize the order: **Delete** → **Write** → **Show**.

1. **Forget Gate (f_t): The Red Pen (Editor).**
   - Decides what to delete from the previous notebook ($\mathbf{c}^{(t-1)}$).
   - Example: "The subject has changed." → Cross out the old subject.
2. **Input Gate (i_t): The Recruitment Officer.**
   - Decides how much of the new draft ($\tilde{\mathbf{c}}^{(t)}$) to "hire" or write into the notebook.
   - Example: "This is a new subject." → Write it down strongly.
3. **Output Gate (o_t): The PR Officer (Spokesperson).**
   - The notebook $\mathbf{c}^{(t)}$ contains everything (possibly raw or messy).
   - We format it (tanh) and decide what to reveal to the outside world ($\mathbf{h}^{(t)}$).
   - Example: "The verb needs to agree with the subject." → Output the singular/plural flag.

3. Synthesis Rule: "Weighted Sum"

The core update rule is:

$$\mathbf{c}^{(t)} = \underbrace{\mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)}}_{\text{Remaining old notes}} + \underbrace{\mathbf{i}^{(t)} \odot \tilde{\mathbf{c}}^{(t)}}_{\text{Accepted new notes}} \tag{16.12}$$

Why use multiplication ($\odot$) and addition?

- **Multiplication ($\odot$)** acts as a **valve**. 0.0 blocks flow (delete), 1.0 lets it pass (keep).
- **Addition ($+$)** naturally **superimposes** the remaining history and the new information.

4. Sigmoid vs Tanh

How to remember which activation to use?

- **Sigmoid (σ)**: Outputs 0 to 1. Use this for **probabilities or ratios** (Knobs/Gates).
- **Tanh**: Outputs -1 to 1. Use this for **content or features** (Drafts/States).

16.4 Common Questions (LSTM)

16.4.1 Q1: Why doesn't the cell state gradient vanish?

A: Due to the additive connection ($\mathbf{c}^{(t)} = \mathbf{c}^{(t-1)} + \ldots$), gradients flow through an **addition path** rather than a multiplication chain.

Comparison:

- **Vanilla RNN**: $\partial L/\partial h^{(1)} \propto \mathbf{W}_{hh}^{100}$ (Exponential decay).
- **LSTM**: $\partial L/\partial c^{(1)} \propto \sum f^{(t)}$ (Summation).

If the forget gate $f^{(t)} \approx 1$, the gradient is multiplied by 1 across time, preventing vanishing.

16.4.2 Q2: Are 4 gates really necessary?

A: Theoretically minimal models exist (e.g., GRU with 2-3 gates), but 4 gates provide stability and clear roles.

- **Forget**: Controls forgetting past (≈ 1 to remember).
- **Input**: Controls writing new info.
- **Output**: Controls reading info.
- **Candidate**: The new content itself.

GRU comparison: GRU merges Input/Forget into an Update gate, and Output into a Reset gate, offering better efficiency but slightly less interpretability.

16.4.3 Q3: Why initialize the Forget gate to 1?

A: To ensure the model **preserves past information by default** at the start of training.

Bias initialization:

$$b_f = 1.0 \implies \sigma(1.0) \approx 0.73 \tag{16.13}$$

With $b_f = 0$, the forget rate starts at 50%, leading to vanishing gradients early in training. Initializing to 1 allows gradients to flow through time immediately.

16.4.4 Q4: Difference between Cell State and Hidden State?

A:

- **Cell State $\mathbf{c}^{(t)}$**: Long-term memory, stable, updated additively. Ideally changes slowly.
- **Hidden State $\mathbf{h}^{(t)}$**: Short-term working memory, output to next layer, highly variable.

Example: "The dogs are running..."

- $\mathbf{c}^{(t)}$ maintains "plural subject" feature.
- $\mathbf{h}^{(t)}$ indicates specific current word "are".

Chapter 17

Sequence-to-Sequence Models and Attention

17.1 Encoder-Decoder Architecture

17.1.1 Motivation

Standard RNNs map an input sequence to an output sequence of the same length (or a single output). Many tasks, like machine translation, map an input sequence $\mathbf{x}^{(1:T)}$ to an output sequence $\mathbf{y}^{(1:T')}$ of a *different* length T'.

The **Encoder-Decoder** architecture solves this by:

1. **Encoder**: Processes the input sequence into a fixed-length "context vector" $\mathbf{c}$.
2. **Decoder**: Generates the output sequence conditioned on $\mathbf{c}$.

17.1.2 Fixed-Length Context Vector

Usually, the final hidden state of the encoder RNN is used as the context:

$$\mathbf{c} = \mathbf{h}_{\text{enc}}^{(T)} \tag{17.1}$$

The decoder is then initialized with $\mathbf{s}^{(0)} = \mathbf{c}$ and generates tokens auto-regressively.

17.2 Attention Mechanism

17.2.1 The Bottleneck Problem

Encoding a long sentence into a single vector $\mathbf{c}$ creates an **information bottleneck**. The decoder must reconstruct the entire meaning from just $\mathbf{c}$.

17.2.2 Attention Weights

Attention allows the decoder to "look back" at the encoder states at every step. For decoder step t and encoder states $\mathbf{h}_s$ ($s = 1 \dots T$):

$$\alpha_{t,s} = \frac{\exp(\text{score}(\mathbf{s}_t, \mathbf{h}_s))}{\sum_{k=1}^{T} \exp(\text{score}(\mathbf{s}_t, \mathbf{h}_k))} \tag{17.2}$$

The context vector becomes dynamic:

$$\mathbf{c}_t = \sum_{s=1}^{T} \alpha_{t,s} \mathbf{h}_s \tag{17.3}$$

17.3 Common Questions (Seq2Seq & Attention)

17.3.1 Q1: Limitations of Fixed Context Vectors?

A: Compressing a long sentence into a small vector is **information-theoretically hard**.

Bottleneck Theory: Input (high info) → Context Vector (e.g., 512 dim) → Output. Compressing 30 words (∼9000 dimensions) into 512 dimensions loses nuance. Attention solves this by assessing source states directly.

17.3.2 Q2: What do Attention weights mean?

A: They represent a probability distribution of "where to look" at time t.

Example: Translating "The cat sat".

- Generating "Le" via Attention → 'The' (0.9), 'cat' (0.1).
- Generating "chat" via Attention → 'cat' (0.8).

17.3.3 Q3: Which Attention score is best?

A: Scaled Dot-Product is the standard winner.

- **Dot**: $\mathbf{s}^T\mathbf{h}$ (Fast).
- **Additive**: $\mathbf{v}^T \tanh(...)$ (Flexible).
- **Scaled Dot**: $\frac{\mathbf{s}^T\mathbf{h}}{\sqrt{d}}$ (Fast + Stable).

17.3.4 Q4: Why Multi-Head Attention?

A: To capture multiple types of relationships simultaneously.

Single Head: Can only focus on one dominant pattern (e.g., "subject-verb"). **Multi-Head:**

- Head 1: Focuses on long-range dependencies.
- Head 2: Focuses on immediate neighbors.
- Head 3: Focuses on syntactic roles.

Parallel processing allows learning all these "views" at once.

Chapter 18

Transformer Architecture

18.1 Self-Attention Mechanism

18.1.1 Query, Key, Value Projection

Given a sequence of embeddings $\mathbf{X} \in \mathbb{R}^{T \times d_{\text{model}}}$:

Project to Query, Key, Value:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \quad \mathbf{Q} \in \mathbb{R}^{T \times d_k} \tag{18.1}$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}_K, \quad \mathbf{K} \in \mathbb{R}^{T \times d_k} \tag{18.2}$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}_V, \quad \mathbf{V} \in \mathbb{R}^{T \times d_v} \tag{18.3}$$

where:

- $\mathbf{W}_Q \in \mathbb{R}^{d_{\text{model}} \times d_k}$ (query projection)
- $\mathbf{W}_K \in \mathbb{R}^{d_{\text{model}} \times d_k}$ (key projection)
- $\mathbf{W}_V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ (value projection)

Typically, $d_k = d_v = d_{\text{model}}/h$ where h is the number of attention heads.

18.1.2 Scaled Dot-Product Attention

Attention scores (unnormalized similarities):

$$\mathbf{E} = \mathbf{Q}\mathbf{K}^\top, \quad \mathbf{E} \in \mathbb{R}^{T \times T} \tag{18.4}$$

Scale by $\sqrt{d_k}$:

$$\mathbf{E}_{\text{scaled}} = \frac{\mathbf{E}}{\sqrt{d_k}} \tag{18.5}$$

Softmax normalization:

$$\mathbf{A} = \text{softmax}(\mathbf{E}_{\text{scaled}}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right) \tag{18.6}$$

Attention output:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathbf{AV} \tag{18.7}$$

18.1.3 Why Scale by $\sqrt{d_k}$?

For random vectors $\mathbf{q}, \mathbf{k} \sim \mathcal{N}(0, 1)^{d_k}$:

$$\text{Var}[\mathbf{q}^\top \mathbf{k}] = d_k \tag{18.8}$$

For large d_k, the dot products have large variance. Scaling by $\sqrt{d_k}$ maintains diffuse attention distributions and prevents gradient saturation.

18.2 Multi-Head Attention

18.2.1 Multiple Attention Heads

For h attention heads (typically $h = 8$ or $h = 12$):

Compute attention for each head i:

$$\text{head}_i = \text{Attention}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i) = \text{softmax}\left(\frac{\mathbf{Q}_i \mathbf{K}_i^\top}{\sqrt{d_k}}\right)\mathbf{V}_i \tag{18.9}$$

Concatenate heads:

$$\text{MultiHead}(\mathbf{X}) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)\mathbf{W}_O \tag{18.10}$$

where $\mathbf{W}_O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$ is the output projection.

18.3 Positional Encoding

18.3.1 Sinusoidal Positional Encoding

For position pos and embedding dimension i:

$$PE(\text{pos}, 2i) = \sin\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right) \tag{18.11}$$

$$PE(\text{pos}, 2i+1) = \cos\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right) \tag{18.12}$$

Properties:

- All values bounded in $[-1, 1]$
- Relative position information encoded via angle addition formulas
- Hierarchical wavelength structure allows learning distances

18.4 Transformer Encoder Block

18.4.1 Block Structure

1. **Multi-Head Self-Attention**
2. **Add & Normalize (residual + layer normalization)**
3. **Position-wise Feed-Forward Network**
4. **Add & Normalize**

18.4.2 Feed-Forward Network

$$\mathbf{Z}_{\text{ffn}} = \max(0, \mathbf{Y}_{\text{attn}}\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2 \tag{18.13}$$

where typically $d_{\text{ffn}} = 4 \times d_{\text{model}}$.

18.4.3 Layer Normalization

$$\text{LayerNorm}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \mathbb{E}[\mathbf{x}]}{\sqrt{\text{Var}[\mathbf{x}] + \epsilon}} + \beta \tag{18.14}$$

Applied row-wise, independent of batch size.

18.5 Transformer Decoder Block

18.5.1 Three Sub-layers

1. **Masked Multi-Head Self-Attention** (target can only attend to positions $\leq$ current)
2. **Multi-Head Cross-Attention** (target attends to encoder output)
3. **Position-wise Feed-Forward Network**

18.5.2 Masked Attention

18.5.3 Masked Attention

Before softmax in self-attention:

$$\mathbf{E}_{\text{masked}} = \mathbf{E} + \mathbf{M} \tag{18.15}$$

where:

$$M_{i,j} = \begin{cases} 0 & \text{if } j \leq i \\ -\infty & \text{if } j > i \end{cases} \tag{18.16}$$

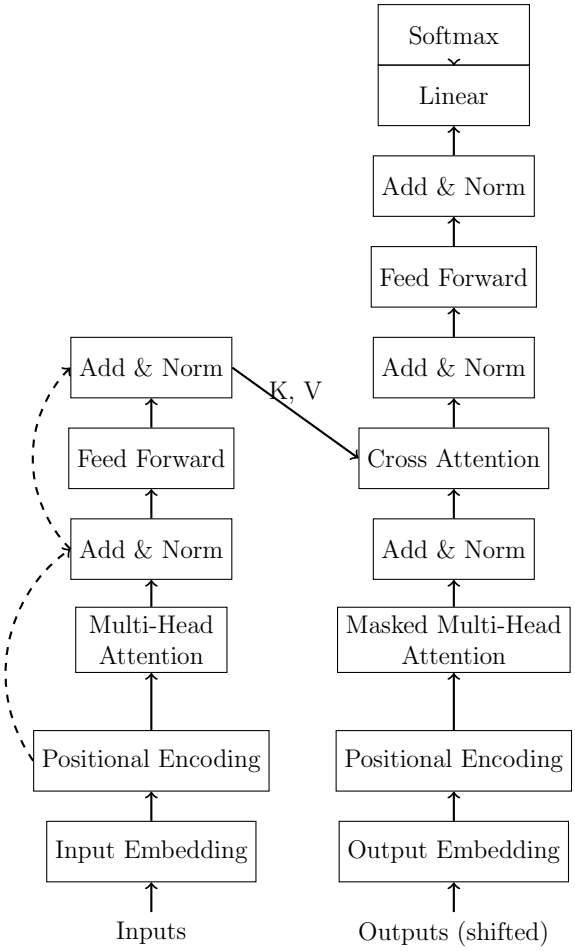

Figure 18.1: The Transformer Architecture. Left: Encoder block. Right: Decoder block. (Based on Vaswani et al., 2017)

18.6 Conceptual Intuition: The Conference Room Analogy

The Transformer (Encoder-Decoder) is best imagined as a **"Conference + Scribe" system** where all participants sit in the same room, referencing each other's statements to build individual notes.

18.6.1 1. Inputs: Participants and Seating

- **Tokens**: The conference participants (words). The sequence length T is the number of seats.
- **Embedding**: The "Name Tag" for each participant, converting their identity (ID) into a meaning vector.
- **Positional Encoding**: The "Seating Chart". Since everyone speaks at once (parallelism), we must explicitly add "I am sitting in seat #1" to the name tag so the model knows the order.

18.6.2 2. Self-Attention: The Q-K-V Mechanism

Imagine each participant (Token) holds three items:

1. **Query (Q): "What I want to know."** (A questionnaire to other participants).
2. **Key (K): "What I can offer."** (An index card summarizing their topic).
3. **Value (V): "My actual content."** (The detailed information to be shared).

The Process:

- **Step 1 (Matching)**: Participant A broadcasts their Query (Q_A). It is compared against everyone's Keys (K).
- **Step 2 (Weighing)**: If Q_A matches K_B well (Dot Product), A pays 90% attention to B. If it matches K_C poorly, A pays 1% attention to C.
- **Step 3 (Gathering)**: A compiles a "Summary Note" by taking a weighted sum of everyone's Values (V) based on these match scores.

18.6.3 3. Multi-Head and Masking

- **Multi-Head Attention**: Conducting the meeting with different "Mindsets" simultaneously. Head 1 checks grammar (Subject-Verb), Head 2 checks context (Pronoun-resolution). They run in parallel.
- **Masked Attention (Decoder)**: A **"Blindfold"** that prevents the Scribe from looking at future speakers. When writing the minutes for time t, they cannot cheat by looking at what speaker $t+1$ will say.

18.6.4 4. The Building Blocks

- **Feed-Forward Network (FFN)**: "Individual Digestion". After gathering info from others (Attention), each participant writes their own interpretation in their notebook independently.
- **Residual Connection**: The "Shortcut". You take your previous notes and **add** the new insights. This prevents you from forgetting what you originally knew.
- **Layer Norm**: The "Organizer". It standardizes the scale of the notes to keep the learning process stable.

18.7 Common Questions (Transformer)

18.7.1 Q1: Why does Self-Attention solve the RNN problem?

A: Because of **direct connections**, the path length between any two tokens is reduced to 1.

Path length comparison:

- **RNN**: Position $1 \to 2 \to \cdots \to 100$. Path length $= 99$.

- **Self-Attention**: Position 1 $\leftrightarrow$ Position 100 (Direct). Path length = 1.

Computational complexity:

- RNN: $O(T \cdot d^2)$ (Sequential).
- Transformer: $O(T^2 \cdot d + T \cdot d^2)$ (Parallel).

18.7.2 Q2: What is the difference between Query, Key, and Value?

A: Think of a library search system.

- **Query**: "What books are about neural networks?" (Search intent).
- **Key**: Book titles/categories (Features to match against).
- **Value**: The actual book content (Content to retrieve).

Mechanism: Match Query with Key $\rightarrow$ Retrieve Value weighted by match strength.

18.7.3 Q3: Why scale by $\sqrt{d_k}$?

A: Without scaling, dot products grow with dimensionality, pushing softmax into regions with vanishing gradients.

Simulation ($d_k = 512$):

- **No scaling**: Dot product variance ≈ 512. Values range ± 30. Softmax becomes one-hot (saturated).
- **With scaling**: Dot$/\sqrt{512}$. Variance ≈ 1. Values range ± 3. Softmax is well-behaved.

18.7.4 Q4: Sinusoidal vs. Learnable Positional Encoding?

A:

- **Sinusoidal**: Can extrapolate to sequence lengths not seen during training. (Used in original Transformer).
- **Learnable**: Cannot handle positions beyond training limit. (Used in BERT, GPT).

If variable/extrapolatable context length is needed, Sinusoidal is preferred.

18.7.5 Q5: Layer Norm vs. Batch Norm?

A: Layer Norm is better for variable-length sequences.

- **Batch Norm**: Normalizes across the batch dimension. Dependence on batch stats is problematic for NLP.
- **Layer Norm**: Normalizes across the feature dimension for each token independently. Stable for RNNs/Transformers.

18.7.6 Q6: What is Masked Attention?

A: In the decoder, it prevents the model from "cheating" by seeing future tokens during training (Teacher Forcing). It enforces causality by setting attention scores to $-\infty$ for future positions.

18.7.7 Q7: Is Transformer really faster than RNN?

A: It depends on whether you mean **Training** or **Inference**. The answer lies in **parallelizability** and the scaling of matrix operations with sequence length T.

1. Training (Parallel): Transformer is Faster

- **Transformer**: Complexity is $O(T^2 \cdot d)$ per layer. Crucially, the attention mechanism computes interactions for all T tokens **simultaneously** as a single large matrix multiplication. There is no sequential dependency in the time dimension, allowing massive parallelization on GPUs.
- **RNN**: Complexity is $O(T \cdot d^2)$. Although it looks linear in T, the hidden state $h_t = \phi(Wh_{t-1} + \dots)$ depends on h_{t-1}. This enforces a **sequential** calculation (Wall-clock time $\propto T$), preventing parallelization across time.

2. Inference (Auto-regressive): Transformer can be Slower Generation is inherently sequential ($t = 1 \to T$).

- **RNN**: $O(T \cdot d^2)$. To generate 1 token, we update the fixed-size state h_t (constant cost). Total cost for length T scales linearly: $O(T)$.
- **Transformer (Naive)**: $O(T^3 \cdot d)$. Re-computing attention for the growing prefix at every step leads to quadratic cost per step, summing to cubic total cost.
- **Transformer (KV Cache)**: $O(T^2 \cdot d)$. By caching previous Keys and Values, each step only attends to the past ($O(t \cdot d)$). Total cost sums to quadratic: $\sum_{t=1}^{T} O(t \cdot d) \approx O(T^2 \cdot d)$.

Conclusion: Transformer is optimized for **fast parallel training** on massive data, even if inference requires tricks (like KV-caching) to remain efficient.

Chapter 19

Scaling Laws and Foundation Models

19.1 Scaling Laws

19.1.1 Empirical Observations

Performance of language models (loss L) scales as a power-law with respect to compute (C), dataset size (N), and parameters (P):

$$L(N) \propto N^{-\alpha_N}, \quad L(P) \propto P^{-\alpha_P}, \quad L(C) \propto C^{-\alpha_C} \tag{19.1}$$

Typically $\alpha \approx 0.05$ to 0.1. This implies that simply scaling up resources predictably improves performance, driving the race for larger models ("Foundation Models").

19.2 In-Context Learning

Large models exhibit an ability to learn from examples in the prompt without parameter updates.

Zero-shot:

```
Translate to French: ‘‘Hello’’ ->
```

Few-shot (In-Context):

```
Translate to French:
‘‘Cat’’ -> ‘‘Chat’’
‘‘Dog’’ -> ‘‘Chien’’
‘‘Hello’’ ->
```

The model infers the task rule from the context.

19.3 Common Questions (Foundation Models)

19.3.1 Q1: Do Scaling Laws hold forever?

A: Currently validated up to $\sim 10^{24}$ FLOPs, but limits exist.

1. **Data Saturation**: High-quality text on the internet is finite.

2. **Compute Costs**: Exponential cost growth.
3. **Irreducible Error**: There is a limit to how well language can be predicted (entropy).

19.3.2 Q2: Why do Emergent Abilities occur?

A: Several hypotheses exist:

- **Phase Transition**: A critical mass of parameters allows complex heuristics to form suddenly.
- **Structured Latent Space**: Larger models learn hierarchically stable representations that enable transfer.
- **Learning to Learn**: The model learns to perform gradient descent-like adaptation purely via attention dynamics during inference (In-Context Learning).

19.3.3 Q3: How does In-Context Learning work?

A: It is essentially **Meta-Learning**. During pre-training, the model sees document structures like "Title -¿ Body" or "Question -¿ Answer". It learns to recognize "Task Pattern" $\rightarrow$ "Output". Mechanistically, Induction Heads (special attention heads) copy patterns from previous context.

19.3.4 Q4: What makes a "Foundation" Model?

A: A single model trained on broad data that can be adapted (fine-tuned) to many downstream tasks.

- **Traditional**: Train one model per task (1 for translation, 1 for sentiment).
- **Foundation**: Pre-train one huge model $\rightarrow$ Adapt to translation, sentiment, coding, etc.

It serves as the "foundation" for a wide array of applications.

19.4 Summary Table: Architecture Comparison

Feature	RNN	LSTM	Transformer
Parallelization	No (Sequential)	No (Sequential)	**Yes (Full)**
Gradient Vanishing	Severe	Mild	None
Memory	Low	Medium	High ($O(T^2)$)
Inference Speed (Long T)	**Fast**	**Fast**	Slow
Training Speed	Slow	Slow	**Fast**
Interpretability	Medium	High	Low (Attention)
Implementation	Easy	Medium	Hard

Chapter 20

Transformer Variants and Modern Architectures

20.1 Overview of Transformer Evolution

20.1.1 Timeline and Motivation

The standard Transformer has spawned numerous variants addressing specific limitations:

Year	Architecture	Key Innovation	Motivation
2017	Transformer	Self-attention + parallel	Baseline
2018	BERT	Bidirectional + MLM	Better understanding
2018	GPT	Decoder-only + scale	Better generation
2019	RoBERTa	BERT improvements	Stronger encoder
2019	Longformer	Sparse attention	Long sequences
2020	T5	Encoder-decoder unified	Unified framework
2021	ViT	Images as patches	Non-text domains
2022	PaLM	Large decoder-only	Scaling to 540B
2023	LLaMA	Efficient decoder	Open-source alternative
2024	Mixtral	MoE variant	Sparse experts

20.2 Encoder-Only: BERT and Variants

20.2.1 BERT Architecture

Bidirectional Encoder Representations from Transformers (BERT) uses a bidirectional encoder stack, pre-trained with **Masked Language Modeling (MLM)**.

20.2.2 Masked Language Modeling (MLM)

Forward pass: Mask 15% of input tokens and predict them.

$$L_{\text{MLM}} = -\sum_{i \in \text{masked}} \log P(\text{token}_i | \text{context}) \tag{20.1}$$

This forces the model to use bidirectional context to infer missing information, learning rich linguistic features.

20.3 Decoder-Only: GPT and Variants

20.3.1 GPT Architecture

Generative Pre-trained Transformer (GPT) uses a decoder-only stack with causal masking, trained for **Next Token Prediction**.

20.3.2 Causal Language Modeling

$$L_{\text{CLM}} = -\sum_{t=1}^{T} \log P(\text{token}_{t+1} | \text{token}_{1:t}) \tag{20.2}$$

BERT (Bidirectional) **GPT (Unidirectional)**

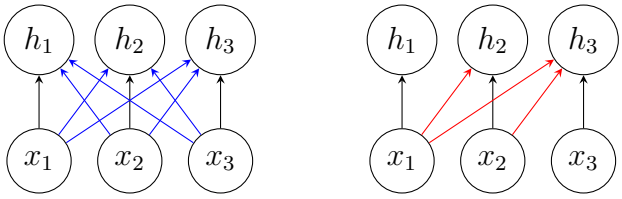

Figure 20.1: Comparison of Attention Patterns. Left: BERT allows attending to all tokens (bidirectional). Right: GPT only allows attending to past tokens (unidirectional causal). (Adapted from Radford et al., 2018; Devlin et al., 2019)

20.3.3 Instruction Tuning and RLHF

Modern LLMs (like ChatGPT) use Reinforcement Learning from Human Feedback (RLHF):

$$\mathcal{L}_{\text{RL}} = \mathbb{E}[\text{KL}(\pi_\theta \| \pi_{\text{ref}}) - \lambda R(\text{response})] \tag{20.3}$$

where R is a reward model predicting human preference.

20.4 Encoder-Decoder: T5 and Variants

20.4.1 T5: Unified Framework

T5 frames all tasks as text-to-text.

- **Translation**: "Translate English to German: ..." $\rightarrow$ Target
- **Classification**: "Classify sentiment: ..." $\rightarrow$ "Positive"

20.5 Sparse and Efficient Variants

20.5.1 The $O(T^2)$ Problem

Standard attention scales quadratically with sequence length T.

20.5.2 Longformer & BigBird

Use **Sparse Attention**:

- **Local**: Attend only to window size w.
- **Global**: Attend to specific tokens (e.g., [CLS]).

Complexity reduces to $O(T \cdot w)$.

20.5.3 Mixture of Experts (MoE)

Switch Transformers route each token to a specific "expert" FFN, allowing massive parameter counts with constant inference cost.

20.6 Multimodal and Vision Transformers

20.6.1 Vision Transformer (ViT)

Splits an image into 16×16 patches, linearly embeds them, and treats them as a sequence of tokens. This brings the scalability of Transformers to Computer Vision.

20.6.2 CLIP

Aligns image and text encoders via contrastive learning:

$$\text{sim}(I, T) = E_I(I) \cdot E_T(T) \tag{20.4}$$

20.7 Recent Trends

20.7.1 RAG (Retrieval-Augmented Generation)

Retrieves documents before generation to reduce hallucinations.

20.7.2 LoRA (Low-Rank Adaptation)

Efficient fine-tuning by decomposing weight updates:

$$\mathbf{W}_{\text{new}} = \mathbf{W}_0 + \mathbf{A}\mathbf{B}^T \tag{20.5}$$

where $\mathbf{A}, \mathbf{B}$ are low-rank matrices.

20.8 Common Questions (Transformer Variants)

20.8.1 Q1: Why is BERT bidirectional but GPT unidirectional?

A: Because their tasks imply different constraints.

- **BERT (Understanding)**: The full sentence is available. To understand "fox", looking at both "quick" (left) and "jumps" (right) helps.

- **GPT (Generation)**: Future tokens do not exist yet. The model must predict the next word using only past context.

Consistency between training and inference determines the directionality.

20.8.2 Q2: Does Masked LM leak data?

A: The masking strategy minimizes checks but isn't perfect. If "lazy" appears frequently in the data, the model might memorize it. However, since the task is to predict missing information from context, it acts as a strong regularizer rather than a trivial copy task.

20.8.3 Q3: Why do models suddenly get smarter with scale?

A: This is called **Emergent Abilities**, likely due to:

1. **Skill Interaction**: Separate skills (e.g., syntax + logic) combine to form complex reasoning.
2. **Structured Space**: Larger latent spaces allow better separation of concepts.
3. **Meta-Learning**: Large models learn to "learn from context" during pre-training.

20.8.4 Q4: What does Temperature do?

A: It controls the balance between creativity and determinism.

- **High τ (e.g., 1.0+)**: Flattens distribution, allowing diverse/rare words (Creative).
- **Low τ (e.g., 0.1)**: Sharpens distribution, picking only the most likely words (Fact-focused).

20.8.5 Q5: Why is the KL penalty needed in RLHF?

A: To prevent "Reward Hacking". Without KL penalty, the model might generate gibberish that technically satisfies the reward model (e.g., repetitive praise). The KL term forces the model to stay close to the natural language distribution learned during SFT.

20.8.6 Q6: How can T5 unify all tasks?

A: Because fundamentally, almost all NLP tasks classify as "Seq2Seq".

- Classification is Sequence $\rightarrow$ Label (Short sequence).
- Generation is Sequence $\rightarrow$ Sequence.

By using prompts, T5 learns a universal mapping function.

20.8.7 Q7: Does Sparse Attention lose information?

A: Potentially, but empirically rarely issues. In many tasks, only local context or specific global tokens matter. "Long-range" dependencies are often sparse (e.g., referencing a name from 500 words ago), which global tokens or random attention can capture efficiently.

20.8.8 Q8: Is Linear Transformer exactly equivalent?

A: No, it is a kernel approximation. Standard Softmax attention is non-linear and order-dependent in a specific way. Linear attention approximates this with feature maps $\phi(\cdot)$. It is much faster for long T, but may lose precision for short T.

20.8.9 Q9: How to decide the number of Experts (MoE)?

A: Based on compute budget and hardware. Typically, scaling experts (e.g., 64 to 128) increases model capacity without slowing down inference, but increases VRAM usage. It's a trade-off between memory and FLOPs.

20.8.10 Q10: Why is ViT better than CNNs?

A: Global Context. CNNs are local (receptive field grows slowly). ViT patches attend to all other patches immediately. With enough data (to overcome lack of inductive bias), this global view is superior.

20.8.11 Q11: RAG vs. Fine-tuning?

A:

- **RAG**: For dynamic knowledge (e.g., "News today"), privacy, and traceability.
- **Fine-tuning**: For adapting style, domain-specific language, or stable knowledge.

Often, RAG + generic LLM is more practical than frequent fine-tuning.

20.8.12 Q12: Prefix Tuning vs. LoRA?

A: LoRA is generally preferred now.

- **LoRA**: Low-rank matrix injection. No latency overhead (can merge weights).
- **Prefix Tuning**: Virtual tokens reduce context window and can be harder to optimize.

Chapter 21

BERT: Bidirectional Encoder with Masked Language Modeling

21.1 Architecture Overview

Figure 21.1 illustrates the BERT architecture, which uses a stack of Transformer encoder layers with bidirectional self-attention.

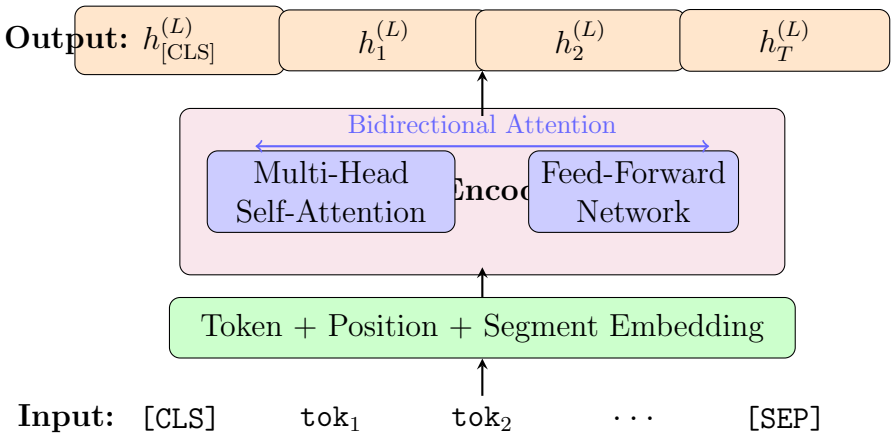

Figure 21.1: BERT Architecture: Encoder-only Transformer with bidirectional self-attention. All tokens can attend to all other tokens in the sequence.

21.1.1 Intuitive Understanding

Why Bidirectional? Traditional language models (like GPT) read text left-to-right, predicting each word based only on previous words. BERT revolutionized this by allowing each token to "see" both its left and right context simultaneously. This is like reading a sentence by looking at the whole sentence at once, rather than word by word.

Key Insight: When humans understand language, we naturally use context from both directions. For example, in "The bank by the river was steep," understanding "bank" requires seeing "river" which comes later. BERT captures this bidirectional understanding.

[CLS] Token: A special token prepended to every input. After processing through all layers, its representation aggregates information from the entire sequence, making it

ideal for classification tasks.

Trade-off: Bidirectional attention means BERT cannot generate text autoregressively (it would see future tokens). Thus, BERT excels at understanding tasks (classification, NER, QA) but not text generation.

21.2 Notation and Input Representation

Let $\mathcal{V}$ be the vocabulary (with $|\mathcal{V}|$ tokens), $T_{\max}$ the maximum sequence length, and d_{model} the model dimension. An input sequence is denoted as

$$(x_1, x_2, \ldots, x_T), \quad x_t \in \mathcal{V},\ 1 \le T \le T_{\max}.$$

We define the token embedding matrix

$$E_{\text{tok}} \in \mathbb{R}^{d_{\text{model}} \times |\mathcal{V}|},$$

the position embedding matrix

$$E_{\text{pos}} \in \mathbb{R}^{d_{\text{model}} \times T_{\max}},$$

and the segment embedding matrix (to distinguish sentence A/B)

$$E_{\text{seg}} \in \mathbb{R}^{d_{\text{model}} \times 2}.$$

Token x_t is represented as a one-hot vector $\text{one\_hot}(x_t) \in \{0,1\}^{|\mathcal{V}|}$, and its embedding is

$$e_t^{\text{tok}} = E_{\text{tok}}\, \text{one\_hot}(x_t) \in \mathbb{R}^{d_{\text{model}}}.$$

The embedding for position t is

$$e_t^{\text{pos}} = E_{\text{pos}}[:, t] \in \mathbb{R}^{d_{\text{model}}},$$

and for segment label $s_t \in \{0,1\}$ (sentence A/B)

$$e_t^{\text{seg}} = E_{\text{seg}}[:, s_t] \in \mathbb{R}^{d_{\text{model}}}.$$

These are summed to form the input to the Transformer:

$$h_t^{(0)} = e_t^{\text{tok}} + e_t^{\text{pos}} + e_t^{\text{seg}} \in \mathbb{R}^{d_{\text{model}}}, \quad t = 1, \ldots, T.$$

Stacking the column vectors $h_t^{(0)}$ vertically:

$$H^{(0)} = \begin{bmatrix} (h_1^{(0)})^\top \\ \vdots \\ (h_T^{(0)})^\top \end{bmatrix} \in \mathbb{R}^{T \times d_{\text{model}}}.$$

For batch processing with padding, with mini-batch size B and maximum length $T_{\max}$:

$$H_{\text{batch}}^{(0)} \in \mathbb{R}^{B \times T_{\max} \times d_{\text{model}}},$$

using a mask matrix

$$M_{\text{pad}} \in \{0, -\infty\}^{B \times 1 \times T_{\max}}$$

to ignore padding positions.

21.3 Encoder Architecture: Bidirectional Self-Attention

The BERT encoder consists of L Transformer blocks. For each layer $\ell = 1, \dots, L$, self-attention output is computed from input $H^{(\ell-1)} \in \mathbb{R}^{T \times d_{\text{model}}}$.

21.3.1 Multi-Head Self-Attention

For each head $h = 1, \dots, H$, define the query, key, and value matrices as

$$Q^{(\ell,h)} = H^{(\ell-1)} W_Q^{(\ell,h)} \in \mathbb{R}^{T \times d_k},$$
$$K^{(\ell,h)} = H^{(\ell-1)} W_K^{(\ell,h)} \in \mathbb{R}^{T \times d_k},$$
$$V^{(\ell,h)} = H^{(\ell-1)} W_V^{(\ell,h)} \in \mathbb{R}^{T \times d_v},$$

where

$$W_Q^{(\ell,h)}, W_K^{(\ell,h)} \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_V^{(\ell,h)} \in \mathbb{R}^{d_{\text{model}} \times d_v},$$

and typically $d_k = d_v = d_{\text{model}}/H$.

The score matrix

$$S^{(\ell,h)} = \frac{Q^{(\ell,h)} (K^{(\ell,h)})^\top}{\sqrt{d_k}} \in \mathbb{R}^{T \times T}$$

is computed. In BERT, bidirectional attention is allowed between all tokens, so only padding is masked. Extending the padding mask $M_{\text{pad}}^{(\text{1D})} \in \{0, -\infty\}^T$:

$$M_{ij}^{(\ell)} = \begin{cases} 0 & \text{if neither is padding,} \\ -\infty & \text{if at least one is padding,} \end{cases}$$

and

$$\tilde{S}^{(\ell,h)} = S^{(\ell,h)} + M^{(\ell)}.$$

Attention weights are computed via softmax:

$$A^{(\ell,h)} = \text{softmax}\big(\tilde{S}^{(\ell,h)}\big) \in \mathbb{R}^{T \times T}, \quad A_{ij}^{(\ell,h)} = \frac{\exp(\tilde{S}_{ij}^{(\ell,h)})}{\sum_{k=1}^{T} \exp(\tilde{S}_{ik}^{(\ell,h)})}.$$

Head output:

$$Y^{(\ell,h)} = A^{(\ell,h)} V^{(\ell,h)} \in \mathbb{R}^{T \times d_v}.$$

All heads are concatenated and transformed by output projection $W_O^{(\ell)} \in \mathbb{R}^{H d_v \times d_{\text{model}}}$:

$$Y^{(\ell)} = \text{Concat}\big(Y^{(\ell,1)}, \dots, Y^{(\ell,H)}\big) W_O^{(\ell)} \in \mathbb{R}^{T \times d_{\text{model}}}.$$

21.3.2 Residual Connection and Layer Normalization

The output of the self-attention sublayer is

$$\tilde{H}^{(\ell)} = \text{LN}\big(H^{(\ell-1)} + Y^{(\ell)}\big),$$

where LN is LayerNorm applied position-wise. For a vector $x \in \mathbb{R}^{d_{\text{model}}}$:

$$\mu(x) = \frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} x_i, \quad \sigma^2(x) = \frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} (x_i - \mu(x))^2,$$

$$\text{LN}(x) = \gamma \odot \frac{x - \mu(x)\mathbf{1}}{\sqrt{\sigma^2(x) + \varepsilon}} + \beta,$$

where $\gamma, \beta \in \mathbb{R}^{d_{\text{model}}}$ are learnable parameters and $\varepsilon > 0$ is a small constant.

21.3.3 Position-Wise Feed-Forward Network

The FFN at each position t is:

$$\mathrm{FFN}(u_t) = W_2^{(\ell)}\,\phi\big(W_1^{(\ell)}u_t + b_1^{(\ell)}\big) + b_2^{(\ell)},$$
$$W_1^{(\ell)} \in \mathbb{R}^{d_{\mathrm{ff}}\times d_{\mathrm{model}}}, \quad W_2^{(\ell)} \in \mathbb{R}^{d_{\mathrm{model}}\times d_{\mathrm{ff}}},$$

where ϕ is a nonlinear function such as ReLU. In matrix form:

$$\mathrm{FFN}(\tilde{H}^{(\ell)}) = \phi\big(\tilde{H}^{(\ell)}W_1^{(\ell)\top} + \mathbf{1}(b_1^{(\ell)})^\top\big)W_2^{(\ell)\top} + \mathbf{1}(b_2^{(\ell)})^\top,$$

where $\mathbf{1} \in \mathbb{R}^T$ is a vector of all ones.

With residual connection and LayerNorm, the layer output is:

$$H^{(\ell)} = \mathrm{LN}\Big(\tilde{H}^{(\ell)} + \mathrm{FFN}\big(\tilde{H}^{(\ell)}\big)\Big).$$

21.4 Pretraining Objective I: Masked Language Modeling

21.4.1 Masking Strategy

From the original sequence $(x_1, \ldots, x_T)$, mask position set $\mathcal{M} \subset \{1, \ldots, T\}$ is sampled randomly (e.g., 15% of total tokens). For $t \in \mathcal{M}$:

- With 80% probability, replace with [MASK];
- With 10% probability, replace with a random token;
- With 10% probability, keep the original token (but still include in loss).

The resulting input sequence $(\tilde{x}_1, \ldots, \tilde{x}_T)$ is passed through the encoder:

$$H^{(L)} = \mathrm{Encoder}(\tilde{x}_{1:T}), \quad h_t^{(L)} \in \mathbb{R}^{d_{\mathrm{model}}}.$$

21.4.2 Token-Level Logits and Probabilities

For masked position $t \in \mathcal{M}$, the output logits are:

$$z_t = W_{\mathrm{MLM}}h_t^{(L)} + b_{\mathrm{MLM}} \in \mathbb{R}^{|\mathcal{V}|},$$
$$p_t = \mathrm{softmax}(z_t), \quad p_t(v) = \frac{\exp(z_{t,v})}{\sum_{u\in\mathcal{V}}\exp(z_{t,u})}.$$

Let the true token be $x_t^\star \in \mathcal{V}$. The loss for a single position is the categorical cross-entropy:

$$\ell_{\mathrm{MLM}}(t) = -\log p_t(x_t^\star).$$

The MLM loss per sequence is:

$$\mathcal{L}_{\mathrm{MLM}} = \frac{1}{|\mathcal{M}|}\sum_{t\in\mathcal{M}}\ell_{\mathrm{MLM}}(t) = -\frac{1}{|\mathcal{M}|}\sum_{t\in\mathcal{M}}\log p_t(x_t^\star).$$

The expected loss (empirical risk) over dataset $\mathcal{D}$ is:

$$\mathcal{L}_{\mathrm{MLM}}(\theta) = \frac{1}{|\mathcal{D}|}\sum_{(x,y)\in\mathcal{D}}\mathbb{E}_{\mathcal{M}\sim\Pi}\left[-\frac{1}{|\mathcal{M}|}\sum_{t\in\mathcal{M}}\log p_t(x_t^\star;\theta)\right],$$

where Π is the masking distribution and θ is the full parameter set of BERT.

21.4.3 Gradient w.r.t. Logits

For position t, we derive the gradient with respect to logits $z_t \in \mathbb{R}^{|\mathcal{V}|}$. Define the one-hot label vector $y_t \in \{0,1\}^{|\mathcal{V}|}$ as:

$$(y_t)_v = \begin{cases} 1 & v = x_t^\star, \\ 0 & \text{otherwise}, \end{cases}$$

then

$$\ell_{\text{MLM}}(t) = -\sum_{v \in \mathcal{V}} (y_t)_v \log p_t(v).$$

By the standard softmax+CCE result:

$$\nabla_{z_t} \ell_{\text{MLM}}(t) = p_t - y_t.$$

Therefore, for the mini-batch average:

$$\nabla_{z_t} \mathcal{L}_{\text{MLM}} = \frac{1}{|\mathcal{M}|}(p_t - y_t).$$

This gradient propagates to the output weights W_{MLM} and hidden representation $h_t^{(L)}$:

$$\nabla_{W_{\text{MLM}}} \mathcal{L}_{\text{MLM}} = \sum_{t \in \mathcal{M}} (p_t - y_t)\,(h_t^{(L)})^\top,$$

$$\nabla_{h_t^{(L)}} \mathcal{L}_{\text{MLM}} = W_{\text{MLM}}^\top (p_t - y_t), \quad t \in \mathcal{M}.$$

21.5 Pretraining Objective II: Next Sentence Prediction

21.5.1 Pair Representation

NSP performs binary classification on a concatenated sequence of two sentences (A, B) using the [CLS] token representation. The input sequence is:

$$[\text{CLS}], A, [\text{SEP}], B, [\text{SEP}]$$

and the output vector at the [CLS] position is denoted $h_{\text{CLS}}^{(L)}$.

Linear classifier:

$$u = W_{\text{NSP}} h_{\text{CLS}}^{(L)} + b_{\text{NSP}} \in \mathbb{R}, \quad q = \sigma(u) = \frac{1}{1 + \exp(-u)},$$

For label $y_{\text{NSP}} \in \{0,1\}$ (1: correct next sentence, 0: random sentence), the loss is:

$$\mathcal{L}_{\text{NSP}} = -\left[y_{\text{NSP}} \log q + (1 - y_{\text{NSP}}) \log(1 - q)\right].$$

21.5.2 Joint Loss

The total loss for a single sample (sentence pair with masked tokens) is:

$$\mathcal{L}_{\text{BERT}} = \mathcal{L}_{\text{MLM}} + \lambda_{\text{NSP}} \mathcal{L}_{\text{NSP}},$$

Dataset average:

$$\min_\theta \frac{1}{|\mathcal{D}|} \sum_{(x,y) \in \mathcal{D}} \mathcal{L}_{\text{BERT}}(x, y; \theta).$$

21.6 Batching, Masks, and Complexity

21.6.1 Attention Mask Matrix

With mini-batch size B and maximum length $T_{\max}$, the batch tensor is:

$$H^{(\ell)} \in \mathbb{R}^{B \times T_{\max} \times d_{\text{model}}}$$

with padding mask $M_{\text{pad}} \in \{0, -\infty\}^{B \times 1 \times T_{\max}}$.

For self-attention, broadcast is applied to the score tensor

$$S \in \mathbb{R}^{B \times H \times T_{\max} \times T_{\max}}$$

as:

$$\tilde{S}_{b,h,i,j} = S_{b,h,i,j} + M_{\text{pad}}[b, 0, j] + M_{\text{pad}}[b, 0, i],$$

so that if either position is PAD, it becomes $-\infty$.

21.6.2 Computational Cost

The attention computation per layer is:

$$O\big(H\, T_{\max}^2 d_k\big),$$

FFN is:

$$O\big(T_{\max} d_{\text{model}} d_{\text{ff}}\big).$$

Total forward pass computation for all L layers:

$$O\big(L(H T_{\max}^2 d_k + T_{\max} d_{\text{model}} d_{\text{ff}})\big),$$

and backpropagation is of the same order.

21.7 Summary

BERT

- constructs contextual representations $h_t^{(L)}$ via bidirectional self-attention encoder (fully connected attention mask),
- uses a pretraining objective combining masked token reconstruction (MLM) and sentence pair prediction (NSP)

to learn general-purpose representations. All of these can be rigorously formulated as token-level log-likelihood maximization problems.

Chapter 22

GPT: Decoder-Only Autoregressive Transformer

22.1 Architecture Overview

Figure 22.1 illustrates the GPT architecture, which uses a stack of Transformer decoder layers with causal (unidirectional) self-attention.

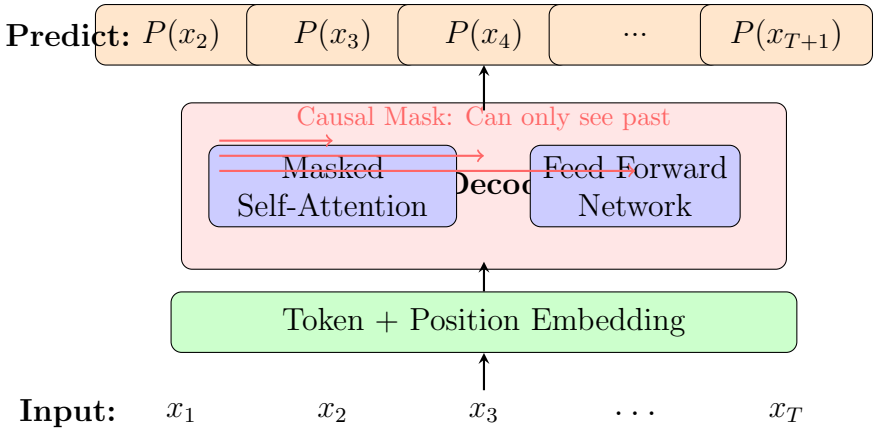

Figure 22.1: GPT Architecture: Decoder-only Transformer with causal (left-to-right) self-attention. Each position can only attend to previous positions.

22.1.1 Intuitive Understanding

Why Causal/Autoregressive? GPT models language as a sequential prediction problem: given all previous words, predict the next word. This mirrors how humans write text—one word at a time, building on what came before.

The Causal Mask: The key mechanism is a triangular attention mask that prevents each position from "peeking" at future tokens. Position 3 can see positions 1 and 2, but not 4, 5, etc. This enables training on entire sequences in parallel while maintaining the autoregressive property.

Why Decoder-Only? Unlike encoder-decoder models (T5), GPT uses only decoder blocks. This simplicity, combined with massive scale, leads to emergent abilities like in-context learning and few-shot reasoning.

Generation Process: At inference time, GPT generates text iteratively:

1. Given prompt tokens, compute all hidden states in parallel
2. Sample next token from the predicted distribution
3. Append sampled token and repeat

Scaling Insight: The GPT family demonstrated that simply scaling model size, data, and compute leads to qualitative improvements in reasoning, knowledge, and generalization ("scaling laws").

22.2 Notation and Factorization of the Language Model

Let $\mathcal{V}$ be the vocabulary with size $|\mathcal{V}|$, and $T_{\max}$ the maximum sequence length. A text is represented as a token sequence

$$x_{1:T} = (x_1, x_2, \ldots, x_T), \quad x_t \in \mathcal{V},\ 1 \le T \le T_{\max}.$$

An autoregressive language model represents the probability distribution via the chain rule:

$$p_\theta(x_{1:T}) = \prod_{t=1}^{T} p_\theta(x_t \mid x_{<t}) \quad (\text{where } x_{<t} = x_1, \ldots, x_{t-1})$$

with parameters θ.

The negative log-likelihood (per sequence) is:

$$\mathcal{L}_{\text{NLL}}(x_{1:T}; \theta) = -\log p_\theta(x_{1:T}) = -\sum_{t=1}^{T} \log p_\theta(x_t \mid x_{<t}).$$

In practice, training uses input and output sequences shifted by one token. For example, input is $(x_1, \ldots, x_T)$, target is $(x_2, \ldots, x_{T+1})$ (last is EOS):

$$\mathcal{L}_{\text{GPT}}(x_{1:T+1}; \theta) = -\frac{1}{T} \sum_{t=1}^{T} \log p_\theta(x_{t+1} \mid x_{1:t}).$$

In the following, we formulate how this conditional probability $p_\theta(\cdot \mid x_{1:t})$ is computed by the Transformer decoder.

22.3 Token, Position, and Input Embeddings

We use the vocabulary embedding matrix

$$E_{\text{tok}} \in \mathbb{R}^{d_{\text{model}} \times |\mathcal{V}|},$$

and position embedding matrix

$$E_{\text{pos}} \in \mathbb{R}^{d_{\text{model}} \times T_{\max}}.$$

Token x_t is represented as a one-hot vector $\text{one\_hot}(x_t) \in \{0,1\}^{|\mathcal{V}|}$:

$$e_t^{\text{tok}} = E_{\text{tok}}\, \text{one\_hot}(x_t) \in \mathbb{R}^{d_{\text{model}}}, \quad e_t^{\text{pos}} = E_{\text{pos}}[:, t] \in \mathbb{R}^{d_{\text{model}}}.$$

The input embedding is:

$$h_t^{(0)} = e_t^{\text{tok}} + e_t^{\text{pos}} \in \mathbb{R}^{d_{\text{model}}}, \quad t = 1, \ldots, T.$$

In matrix form:

$$H^{(0)} = \begin{bmatrix} (h_1^{(0)})^\top \\ \vdots \\ (h_T^{(0)})^\top \end{bmatrix} \in \mathbb{R}^{T \times d_{\text{model}}}.$$

22.4 Causal Multi-Head Self-Attention

GPT consists of L Transformer decoder blocks, with each layer ℓ:

$$H^{(\ell)} = \text{Block}^{(\ell)}(H^{(\ell-1)}), \quad \ell = 1, \ldots, L.$$

22.4.1 Single Head Self-Attention with Causal Mask

For $H^{(\ell-1)} \in \mathbb{R}^{T \times d_{\text{model}}}$, define the single-head query, key, and value matrices as:

$$Q = H^{(\ell-1)} W_Q, \quad K = H^{(\ell-1)} W_K, \quad V = H^{(\ell-1)} W_V,$$

$$W_Q, W_K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \quad W_V \in \mathbb{R}^{d_{\text{model}} \times d_v}.$$

Let row t be q_t, k_t, v_t.

The score matrix:

$$S \in \mathbb{R}^{T \times T}, \quad S_{ij} = \frac{q_i^\top k_j}{\sqrt{d_k}}.$$

The causal mask $M \in \{0, -\infty\}^{T \times T}$ is:

$$M_{ij} = \begin{cases} 0 & j \le i, \\ -\infty & j > i, \end{cases}$$

to prevent attention to future tokens. Masked scores:

$$\tilde{S} = S + M.$$

Attention weights via row-wise softmax:

$$A_{ij} = \frac{\exp(\tilde{S}_{ij})}{\sum_{k=1}^{T} \exp(\tilde{S}_{ik})}, \quad A \in \mathbb{R}^{T \times T}.$$

Single head output:

$$Y = AV \in \mathbb{R}^{T \times d_v}, \quad y_i = \sum_{j=1}^{T} A_{ij} v_j.$$

22.4.2 Multi-Head Attention and Output Projection

Let the number of heads be H. For each head h:

$$Q^{(h)} = H^{(\ell-1)}W_Q^{(h)}, \quad K^{(h)} = H^{(\ell-1)}W_K^{(h)}, \quad V^{(h)} = H^{(\ell-1)}W_V^{(h)},$$

$$Y^{(h)} = \mathrm{Attention}_{\mathrm{causal}}(Q^{(h)}, K^{(h)}, V^{(h)}), \quad Y^{(h)} \in \mathbb{R}^{T\times d_v}.$$

Concatenate and apply linear transformation:

$$Y^{(\ell)} = \mathrm{Concat}\big(Y^{(1)}, \ldots, Y^{(H)}\big)W_O^{(\ell)} \in \mathbb{R}^{T\times d_{\mathrm{model}}},$$

$$W_O^{(\ell)} \in \mathbb{R}^{Hd_v\times d_{\mathrm{model}}}.$$

22.4.3 Residual and Pre/Post-Norm Variants

The standard Post-LN form is:

$$\tilde{H}^{(\ell)} = \mathrm{LN}\big(H^{(\ell-1)} + Y^{(\ell)}\big),$$

$$H^{(\ell)} = \mathrm{LN}\Big(\tilde{H}^{(\ell)} + \mathrm{FFN}\Big(\tilde{H}^{(\ell)}\Big)\Big).$$

Many GPT-style models use the Pre-LN form:

$$\hat{H}^{(\ell)} = H^{(\ell-1)} + \mathrm{MHA}_{\mathrm{causal}}\big(\mathrm{LN}(H^{(\ell-1)})\big),$$

$$H^{(\ell)} = \hat{H}^{(\ell)} + \mathrm{FFN}\Big(\mathrm{LN}(\hat{H}^{(\ell)})\Big).$$

LayerNorm LN for vector $x \in \mathbb{R}^{d_{\mathrm{model}}}$ is:

$$\mathrm{LN}(x) = \gamma \odot \frac{x - \mu(x)\mathbf{1}}{\sqrt{\sigma^2(x) + \varepsilon}} + \beta,$$

$$\mu(x) = \frac{1}{d_{\mathrm{model}}}\sum_{i=1}^{d_{\mathrm{model}}} x_i, \quad \sigma^2(x) = \frac{1}{d_{\mathrm{model}}}\sum_{i=1}^{d_{\mathrm{model}}} (x_i - \mu(x))^2,$$

where $\gamma, \beta \in \mathbb{R}^{d_{\mathrm{model}}}$ are learnable parameters.

22.5 Position-Wise Feed-Forward Network

The FFN at each layer ℓ is applied independently to each position:

$$\mathrm{FFN}^{(\ell)}(u) = W_2^{(\ell)}\,\phi\Big(W_1^{(\ell)}u + b_1^{(\ell)}\Big) + b_2^{(\ell)},$$

$$W_1^{(\ell)} \in \mathbb{R}^{d_{\mathrm{ff}}\times d_{\mathrm{model}}}, \quad W_2^{(\ell)} \in \mathbb{R}^{d_{\mathrm{model}}\times d_{\mathrm{ff}}},$$

where ϕ is ReLU, GELU, etc. In matrix form:

$$\mathrm{FFN}^{(\ell)}(H) = \phi\Big(HW_1^{(\ell)\top} + \mathbf{1}(b_1^{(\ell)})^\top\Big)\,W_2^{(\ell)\top} + \mathbf{1}(b_2^{(\ell)})^\top,$$

where $\mathbf{1} \in \mathbb{R}^T$ is the all-ones vector.

22.6 Output Layer and Conditional Distribution

The final layer output is:

$$H^{(L)} = \begin{bmatrix} (h_1^{(L)})^\top \\ \vdots \\ (h_T^{(L)})^\top \end{bmatrix},$$

and vocabulary logits are:

$$z_t = W_{\mathrm{LM}} h_t^{(L)} + b_{\mathrm{LM}} \in \mathbb{R}^{|\mathcal{V}|},$$

$$p_\theta(x_{t+1} = v \mid x_{1:t}) = \frac{\exp(z_{t,v})}{\sum_{u \in \mathcal{V}} \exp(z_{t,u})}, \quad v \in \mathcal{V}.$$

With weight tying, $W_{\mathrm{LM}} = E_{\mathrm{tok}}^\top$, sharing parameters between embedding and output projection.

22.7 Training Objective and Gradients

22.7.1 Sequence Loss and Per-Token Loss

Loss for input $x_{1:T+1}$:

$$\mathcal{L}_{\mathrm{GPT}}(x_{1:T+1}; \theta) = -\frac{1}{T} \sum_{t=1}^{T} \log p_\theta(x_{t+1} \mid x_{1:t}).$$

Loss at time t:

$$\ell_t(\theta) = -\log p_\theta(x_{t+1}^\star \mid x_{1:t}),$$

With 1-hot label $y_t \in \{0,1\}^{|\mathcal{V}|}$:

$$(y_t)_v = \begin{cases} 1 & v = x_{t+1}^\star, \\ 0 & \text{otherwise}, \end{cases}$$

then

$$\ell_t(\theta) = -\sum_{v \in \mathcal{V}} (y_t)_v \log p_\theta(v \mid x_{1:t}).$$

22.7.2 Gradient w.r.t. Logits and Hidden States

By the standard softmax + cross-entropy result:

$$\nabla_{z_t} \ell_t = p_t - y_t, \quad p_t = p_\theta(\cdot \mid x_{1:t}).$$

Thus for batch average (including normalization factor $1/T$):

$$\nabla_{z_t} \mathcal{L}_{\mathrm{GPT}} = \frac{1}{T}(p_t - y_t).$$

From linear projection $z_t = W_{\mathrm{LM}} h_t^{(L)} + b_{\mathrm{LM}}$:

$$\nabla_{W_{\mathrm{LM}}} \mathcal{L}_{\mathrm{GPT}} = \frac{1}{T} \sum_{t=1}^{T} (p_t - y_t)\, (h_t^{(L)})^\top,$$

$$\nabla_{b_{\mathrm{LM}}} \mathcal{L}_{\mathrm{GPT}} = \frac{1}{T} \sum_{t=1}^{T} (p_t - y_t),$$

$$\nabla_{h_t^{(L)}} \mathcal{L}_{\mathrm{GPT}} = \frac{1}{T} W_{\mathrm{LM}}^{\top} (p_t - y_t).$$

This $\nabla_{h_t^{(L)}} \mathcal{L}_{\mathrm{GPT}}$ is backpropagated through the Transformer decoder.

22.8 Teacher Forcing and Inference

22.8.1 Teacher Forcing During Training

During training, to evaluate the conditional distribution

$$p_\theta(x_{t+1} \mid x_{1:t})$$

the input sequence $(x_1, \ldots, x_t)$ is always given the "ground truth tokens" (teacher forcing). Thus, the likelihood computed during training is:

$$p_\theta(x^\star_{2:T+1} \mid x^\star_{1:T}) = \prod_{t=1}^{T} p_\theta(x^\star_{t+1} \mid x^\star_{1:t}).$$

22.8.2 Autoregressive Generation at Test Time

During generation, model samples are recursively fed back:

$$\begin{aligned} &\text{given } x_1, \ldots, x_t, \\ &\quad p_\theta(x_{t+1} \mid x_{1:t}) = \mathrm{softmax}(W_{\mathrm{LM}} h_t^{(L)} + b_{\mathrm{LM}}), \\ &\quad x_{t+1} \sim p_\theta(\cdot \mid x_{1:t}). \end{aligned}$$

With temperature $\tau > 0$:

$$p_\theta^\tau(v \mid x_{1:t}) = \frac{\exp(z_{t,v}/\tau)}{\sum_u \exp(z_{t,u}/\tau)}.$$

22.9 Perplexity and Evaluation Metric

The average negative log-likelihood per token on test distribution $\mathcal{D}_{\mathrm{test}}$ is:

$$\bar{\ell} = \mathbb{E}_{x_{1:T} \sim \mathcal{D}_{\mathrm{test}}} \left[-\frac{1}{T} \sum_{t=1}^{T} \log_2 p_\theta(x_t \mid x_{<t}) \right],$$

then Perplexity is defined as:

$$\mathrm{PPL} = 2^{\bar{\ell}}.$$

This can be interpreted as the effective vocabulary size representing "how many choices the model is selecting from on average."

22.10 Batching, Causal Mask, and Complexity

22.10.1 Batch-Wise Causal Attention

With mini-batch size B and maximum length $T_{\max}$:

$$H^{(\ell)} \in \mathbb{R}^{B\times T_{\max}\times d_{\text{model}}},$$

for each head:

$$Q, K, V \in \mathbb{R}^{B\times H\times T_{\max}\times d_k},$$

Scores:

$$S_{b,h,i,j} = \frac{1}{\sqrt{d_k}} Q_{b,h,i,:} \cdot K_{b,h,j,:},$$

Mask:

$$M_{i,j} = \begin{cases} 0 & j \le i, \\ -\infty & j > i, \end{cases}$$

broadcast to:

$$\tilde{S}_{b,h,i,j} = S_{b,h,i,j} + M_{i,j}.$$

Attention weights via softmax:

$$A_{b,h,i,j} = \frac{\exp(\tilde{S}_{b,h,i,j})}{\sum_{k=1}^{T_{\max}} \exp(\tilde{S}_{b,h,i,k})},$$

Output:

$$Y_{b,h,i,:} = \sum_{j=1}^{T_{\max}} A_{b,h,i,j} V_{b,h,j,:}.$$

22.10.2 Computational Complexity

Per-layer computation:

$$O\big(BHT_{\max}^2 d_k\big) \quad \text{(Attention)},$$

$$O\big(BT_{\max} d_{\text{model}} d_{\text{ff}}\big) \quad \text{(FFN)}.$$

For all L layers:

$$O\Big(L\big(BHT_{\max}^2 d_k + BT_{\max} d_{\text{model}} d_{\text{ff}}\big)\Big).$$

During inference, KV cache reduces computation at time t to:

$$O\big(BHtd_k + Bd_{\text{model}} d_{\text{ff}}\big),$$

and for generating sequence of length T:

$$O\big(BHT^2 d_k + BTd_{\text{model}} d_{\text{ff}}\big).$$

22.11 Summary

GPT

- expresses $p_\theta(x_t \mid x_{<t})$ via causal masked self-attention,
- defines conditional categorical distribution via softmax from token embedding and output linear projection,
- minimizes negative log-likelihood in mini-batch units,

formulating a rigorous probabilistic model. This framework forms the foundation for scaling laws, in-context learning, and RLHF discussed in later chapters.

Chapter 23

RoBERTa: Robustly Optimized BERT Pretraining

23.1 Architectural Overview

RoBERTa has fundamentally the same Transformer encoder-only structure as BERT, applying bidirectional self-attention to all tokens.

Let $\mathcal{V}$ be the vocabulary, $T_{\max}$ the maximum sequence length, d_{model} the model dimension, and L the number of layers. For input token sequence

$$x_{1:T} = (x_1, \ldots, x_T), \quad x_t \in \mathcal{V},\ 1 \le T \le T_{\max},$$

the encoder output is:

$$H^{(L)} = \begin{bmatrix} (h_1^{(L)})^\top \\ \vdots \\ (h_T^{(L)})^\top \end{bmatrix} \in \mathbb{R}^{T \times d_{\text{model}}}.$$

The essential differences of RoBERTa are:

- Training objective (MLM only with dynamic masking),
- Removal of NSP,
- Training schedule (large batch, longer training, larger corpus).

23.2 Token and Segment Representation

Vocabulary embedding matrix:

$$E_{\text{tok}} \in \mathbb{R}^{d_{\text{model}} \times |\mathcal{V}|},$$

Position embedding:

$$E_{\text{pos}} \in \mathbb{R}^{d_{\text{model}} \times T_{\max}}.$$

Unlike BERT, RoBERTa does not use segment embeddings, treating all as a single sentence (or concatenated sentences).

With one-hot vector of token x_t as $\text{one\_hot}(x_t) \in \{0,1\}^{|\mathcal{V}|}$:

$$e_t^{\text{tok}} = E_{\text{tok}}\, \text{one\_hot}(x_t) \in \mathbb{R}^{d_{\text{model}}}, \quad e_t^{\text{pos}} = E_{\text{pos}}[:, t] \in \mathbb{R}^{d_{\text{model}}}.$$

Input embedding:

$$h_t^{(0)} = e_t^{\text{tok}} + e_t^{\text{pos}} \in \mathbb{R}^{d_{\text{model}}}, \quad t = 1, \ldots, T.$$

23.3 Bidirectional Self-Attention Encoder

For each layer ℓ:

$$H^{(\ell)} = \text{EncoderBlock}^{(\ell)}\big(H^{(\ell-1)}\big), \quad \ell = 1, \dots, L.$$

EncoderBlock consists of multi-head self-attention (unmasked), position-wise FFN, and residual connections with LayerNorm.

23.3.1 Multi-Head Self-Attention (Unmasked)

For layer ℓ input $H^{(\ell-1)} \in \mathbb{R}^{T \times d_{\text{model}}}$, for head $h = 1, \dots, H$:

$$Q^{(h)} = H^{(\ell-1)} W_Q^{(h)}, \quad K^{(h)} = H^{(\ell-1)} W_K^{(h)}, \quad V^{(h)} = H^{(\ell-1)} W_V^{(h)}.$$

Score matrix:

$$S_{ij}^{(h)} = \frac{(q_i^{(h)})^\top k_j^{(h)}}{\sqrt{d_k}},$$

Since this is BERT/RoBERTa, no causal mask is used, and scores for all i, j are passed directly to softmax:

$$A_{ij}^{(h)} = \frac{\exp(S_{ij}^{(h)})}{\sum_{k=1}^{T} \exp(S_{ik}^{(h)})}.$$

23.3.2 Pre-LN Encoder Block

In practice, Pre-LN is often used:

$$\hat{H}^{(\ell)} = H^{(\ell-1)} + \text{MHA}\big(\text{LN}(H^{(\ell-1)})\big),$$

$$H^{(\ell)} = \hat{H}^{(\ell)} + \text{FFN}\Big(\text{LN}(\hat{H}^{(\ell)})\Big).$$

23.4 Masked Language Modeling with Dynamic Masking

23.4.1 Masking Strategy as a Random Process

For input sequence $x_{1:T}$, mask position set $M \subset \{1, \dots, T\}$ is selected probabilistically. For each position t, Bernoulli variable $m_t \sim \text{Bernoulli}(p_{\text{mask}})$, $p_{\text{mask}} \approx 0.15$, mask set $M = \{t \mid m_t = 1\}$.

For each $t \in M$:

$$\tilde{x}_t = \begin{cases} \texttt{[MASK]} & \text{with prob. } 0.8, \\ u \sim \text{Unif}(\mathcal{V}) & \text{with prob. } 0.1, \\ x_t & \text{with prob. } 0.1. \end{cases}$$

RoBERTa adopts "dynamic masking," where the same sentence is masked with different M in different iterations.

23.4.2 MLM Objective as Conditional Likelihood

Let the masked input sequence be $\tilde{x}_{1:T}$ and encoder output be $H^{(L)}(\tilde{x}_{1:T})$. From hidden state $h_t^{(L)}$ at position t:

$$z_t = W_{\text{MLM}} h_t^{(L)} + b_{\text{MLM}} \in \mathbb{R}^{|\mathcal{V}|},$$
$$p_\theta(v \mid \tilde{x}_{1:T}, t) = \frac{\exp(z_{t,v})}{\sum_{u \in \mathcal{V}} \exp(z_{t,u})}.$$

MLM loss per sentence:

$$\mathcal{L}_{\text{MLM}}(x_{1:T}; \theta) = \mathbb{E}_{M, \tilde{x}_{1:T}} \left[-\frac{1}{|M|} \sum_{t \in M} \log p_\theta(x_t \mid \tilde{x}_{1:T}, t) \right].$$

23.4.3 Per-Token Cross-Entropy and Gradients

Loss for position $t \in M$:

$$\ell_t(\theta) = -\log p_\theta(x_t^\star \mid \tilde{x}_{1:T}, t).$$

By standard softmax + cross-entropy result:

$$\nabla_{z_t} \ell_t = p_t - y_t.$$

Gradient w.r.t. projection parameters:

$$\nabla_{W_{\text{MLM}}} \hat{\mathcal{L}}_{\text{MLM}} = \frac{1}{|M|} \sum_{t \in M} (p_t - y_t) \, (h_t^{(L)})^\top.$$

23.5 Dynamic vs. Static Masking: Distributional View

BERT's "static masking" samples M only once during corpus preprocessing, then trains for multiple epochs with the same mask pattern.

RoBERTa's dynamic masking samples new $M^{(e)}$ each epoch:

$$\mathcal{L}_{\text{MLM}}(x_{1:T}; \theta) = \mathbb{E}_M \big[L(x_{1:T}, M; \theta) \big],$$

providing Monte Carlo iterations that more faithfully approximate the expectation.

23.6 Pretraining Objective and Optimization

For large-scale corpus $\mathcal{D}$, RoBERTa's pretraining objective is:

$$\min_\theta \; \mathbb{E}_{x_{1:T} \sim \mathcal{D}} \Big[\mathcal{L}_{\text{MLM}}(x_{1:T}; \theta) \Big].$$

RoBERTa uses:

- Larger batch size B,
- Longer training steps,
- Larger dataset

to improve representation capability while maintaining the same architecture as BERT.

Chapter 24

Longformer: Efficient Long-Document Transformer

24.1 Motivation and the $O(T^2)$ Bottleneck

Standard Transformer self-attention scales as $O(T^2)$ in time and memory for sequence length T. For long documents ($T > 4096$ etc.), this quadratic complexity becomes a practical bottleneck. Longformer introduces **sparse attention** patterns to reduce computation to linear while preserving long-range dependencies.

24.2 Attention Mask and Sparsity Pattern

24.2.1 Full Attention Recap

In standard full attention, position i attends to all positions $j \in \{1, \ldots, T\}$:

$$A_{ij} = \frac{\exp(S_{ij})}{\sum_{k=1}^{T} \exp(S_{ik})}, \quad y_i = \sum_{j=1}^{T} A_{ij} v_j.$$

24.2.2 Sparse Attention as a Structured Mask

Longformer defines a "permitted attention set" $\mathcal{N}(i) \subseteq \{1, \ldots, T\}$ for each position i:

$$M_{ij} = \begin{cases} 0 & j \in \mathcal{N}(i), \\ -\infty & j \notin \mathcal{N}(i). \end{cases}$$

24.3 Sliding Window Attention

24.3.1 Definition of Window Size w

Sliding window attention where each position i attends only to local window of radius w:

$$\mathcal{N}_{\text{local}}(i) = \big\{ j \in \{1, \ldots, T\} \;\big|\; |i - j| \leq w \big\}.$$

Total attention entries computed across all positions is $O(Tw)$.

24.3.2 Multi-Layer Reception Field

With L stacked layers, each position can indirectly reach distance:

$$r_{\text{receptive}} = L \cdot w.$$

24.4 Dilated Sliding Window (Optional)

For further reception field expansion, dilated windows can be introduced. With stride d_ℓ at layer ℓ:

$$\mathcal{N}^{(\ell)}_{\text{dilated}}(i) = \big\{ j \;\big|\; j \in \{i - wd_\ell, i - (w-1)d_\ell, \ldots, i + wd_\ell\} \big\}.$$

24.5 Global Attention

24.5.1 Global Token Set $\mathcal{G} \subset \{1, \ldots, T\}$

Pre-specified global token set $\mathcal{G}$:

- Position $i \in \mathcal{G}$ can attend to all tokens $j \in \{1, \ldots, T\}$,
- Position $i \notin \mathcal{G}$ attends to tokens within window and all global tokens.

Thus:

$$\mathcal{N}_{\text{full}}(i) = \begin{cases} \{1, \ldots, T\} & i \in \mathcal{G}, \\ \mathcal{N}_{\text{local}}(i) \cup \mathcal{G} & i \notin \mathcal{G}. \end{cases}$$

24.5.2 Global Token Selection Strategies

$\mathcal{G}$ is determined based on task:

1. CLS token: First token $\mathcal{G} = \{1\}$.
2. Question tokens (QA task): All question tokens as $\mathcal{G}$.
3. Fixed interval: $\mathcal{G} = \{1, 1 + s, 1 + 2s, \ldots\}$.

24.6 Computational Complexity Analysis

24.6.1 Per-Layer Complexity

With number of global tokens $|\mathcal{G}| = g$:

$$\text{Attention cost per layer} = O\big(T(w + g)d_k\big).$$

In practice, $g \ll T$, so dominant term is $O(Twd_k)$, achieving linear scaling.

24.6.2 Total Model Complexity

Including all L layers and FFN:

$$\text{Total cost} = O\Big(L\big(Twd_k + gTd_k + Td_{\text{model}}d_{\text{ff}}\big)\Big).$$

24.7 Formal Attention Definition in Longformer

24.7.1 Score and Mask Computation

Longformer mask:

$$M_{ij}^{\text{Longformer}} = \begin{cases} 0 & j \in \mathcal{N}_{\text{full}}(i), \\ -\infty & j \notin \mathcal{N}_{\text{full}}(i). \end{cases}$$

24.7.2 Sparse Softmax

Softmax for row i is computed over non-zero scores only:

$$A_{ij} = \begin{cases} \dfrac{\exp(\tilde{S}_{ij})}{\sum_{k \in \mathcal{N}_{\text{full}}(i)} \exp(\tilde{S}_{ik})} & j \in \mathcal{N}_{\text{full}}(i), \\ 0 & \text{otherwise.} \end{cases}$$

Output:

$$y_i = \sum_{j \in \mathcal{N}_{\text{full}}(i)} A_{ij} v_j.$$

24.8 Gradient Flow Through Sparse Attention

Gradient computation during backpropagation is also performed only for permitted attention entries. The softmax Jacobian is:

$$\frac{\partial A_{ij}}{\partial S_{ik}} = \begin{cases} A_{ij}(\delta_{jk} - A_{ik}) & k \in \mathcal{N}_{\text{full}}(i), \\ 0 & k \notin \mathcal{N}_{\text{full}}(i). \end{cases}$$

24.9 Comparison with Other Efficient Transformers

24.9.1 BigBird

BigBird also uses sparse attention, adding random attention and block-sparse attention.

24.9.2 Linformer / Performer

Linformer achieves $O(T)$ via low-rank approximation, approximating the attention matrix itself. Performer achieves $O(T)$ via kernel trick linearization of softmax, but is not strictly equivalent to standard softmax attention.

Longformer's distinguishing feature is preserving exact attention on sparse connections.

24.10 Summary

Longformer:

- Captures local dependencies in $O(Tw)$ via sliding window attention,

- Handles long-range dependencies and sequence-wide information aggregation via global attention,
- With overall $O(T(w+g)d_k)$ complexity, handles long documents with $T \sim 4096$ or more,

significantly improving Transformer practicality for document-level NLP tasks.

Chapter 25

T5: Text-to-Text Transfer Transformer

25.1 Architecture Overview

Figure 25.1 illustrates the T5 architecture, which uses the full encoder-decoder Transformer structure.

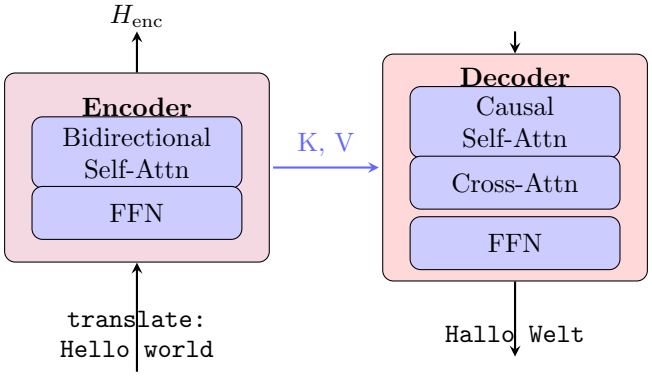

Figure 25.1: T5 Architecture: Full encoder-decoder Transformer. The encoder processes input bidirectionally; the decoder generates output autoregressively while attending to encoder representations via cross-attention.

25.1.1 Intuitive Understanding

Text-to-Text Unification: T5's revolutionary insight is that *every* NLP task can be framed as text generation:

- **Translation**: "translate English to German: Hello" → "Hallo"
- **Summarization**: "summarize: [long article]" → "[summary]"
- **Classification**: "sentiment: I love this!" → "positive"
- **Question Answering**: "question: What is 2+2? context: ..." → "4"

This unified interface allows a single model to handle diverse tasks.

Encoder-Decoder Structure: Unlike decoder-only GPT, T5 separates "understanding" (encoder) from "generation" (decoder):

- **Encoder**: Reads entire input bidirectionally, building rich representations
- **Decoder**: Generates output autoregressively, attending to encoder via cross-attention

Span Corruption Pretraining: Instead of masking single tokens (BERT) or predicting next tokens (GPT), T5 masks contiguous spans and predicts them. This teaches the model both local and global patterns.

Why This Works: By casting all tasks as sequence-to-sequence, T5 leverages the same architecture and training objective universally, enabling strong transfer learning.

25.2 Unified Text-to-Text Framework

T5 is a unified framework that treats all NLP tasks as the same input-to-output text transformation problem.

25.2.1 Task Formulation as Conditional Generation

For vocabulary $\mathcal{V}$, input sequence $x_{1:T_x} = (x_1, \ldots, x_{T_x})$, output sequence $y_{1:T_y} = (y_1, \ldots, y_{T_y})$, with $x_t, y_s \in \mathcal{V}$.

All tasks are unified as a conditional probability model:

$$p_\theta(y_{1:T_y} \mid x_{1:T_x}) = \prod_{s=1}^{T_y} p_\theta(y_s \mid x_{1:T_x}, y_{<s})$$

to treat uniformly.

25.2.2 Task Prefix and Examples

Embed task information as prefix in text:

- Translation: $x =$ "translate English to German: That is good.", $y =$ "Das ist gut."
- Summarization: $x =$ "summarize: [long document]", $y =$ "[summary]"
- Classification: $x =$ "sentiment: This movie is great!", $y =$ "positive"

25.3 Encoder-Decoder Architecture

T5 adopts standard Transformer Encoder-Decoder.

25.3.1 Encoder: Bidirectional Self-Attention

For input $x_{1:T_x}$, encoder layers $\ell = 1, \ldots, L_{\text{enc}}$ apply bidirectional self-attention where all positions can attend to each other:

$$H_{\text{enc}}^{(\ell)} = \text{EncoderBlock}^{(\ell)}\left(H_{\text{enc}}^{(\ell-1)}\right).$$

Final output:

$$H_{\text{enc}} = H_{\text{enc}}^{(L_{\text{enc}})} \in \mathbb{R}^{T_x \times d_{\text{model}}}.$$

25.3.2 Decoder: Causal Self-Attention + Cross-Attention

Decoder layers $\ell = 1, \ldots, L_{\text{dec}}$ have 3 sub-layers:

1. Masked (causal) self-attention: Apply causal mask to prevent seeing future output tokens.
2. Cross-attention: Read information from encoder output H_{enc}.
3. Feed-forward network: Position-wise nonlinear transformation.

25.3.3 Masked Self-Attention in Decoder

Causal mask:

$$M_{\text{causal},ij} = \begin{cases} 0 & j \le i, \\ -\infty & j > i, \end{cases}$$

applied as:

$$\tilde{S}_{\text{self},ij} = S_{\text{self},ij} + M_{\text{causal},ij}.$$

25.3.4 Cross-Attention to Encoder

Cross-attention to encoder output H_{enc}:

$$Q_{\text{cross}} = \hat{H}_{\text{dec}}^{(\ell)} W_Q^{\text{cross}}, \quad K_{\text{cross}} = H_{\text{enc}} W_K^{\text{cross}}, \quad V_{\text{cross}} = H_{\text{enc}} W_V^{\text{cross}}.$$

No mask here (each decoder position can see entire input):

$$A_{\text{cross},ij} = \frac{\exp(S_{\text{cross},ij})}{\sum_{k=1}^{T_x} \exp(S_{\text{cross},ik})}.$$

25.3.5 Decoder Block Composition

In Pre-LN form:

$$\begin{aligned} \hat{H}_{\text{dec}}^{(\ell)} &= H_{\text{dec}}^{(\ell-1)} + \text{MaskedSelfAttn}\big(\text{LN}(H_{\text{dec}}^{(\ell-1)})\big), \\ \tilde{H}_{\text{dec}}^{(\ell)} &= \hat{H}_{\text{dec}}^{(\ell)} + \text{CrossAttn}\big(\text{LN}(\hat{H}_{\text{dec}}^{(\ell)}), H_{\text{enc}}\big), \\ H_{\text{dec}}^{(\ell)} &= \tilde{H}_{\text{dec}}^{(\ell)} + \text{FFN}\big(\text{LN}(\tilde{H}_{\text{dec}}^{(\ell)})\big). \end{aligned}$$

25.4 Relative Position Bias

T5 uses relative position bias added to attention scores instead of absolute position embedding.

25.4.1 Bias Definition

Introduce relative position bias matrix $B^{(\ell)} \in \mathbb{R}^{T \times T}$:

$$S_{ij}^{(\text{with bias})} = \frac{q_i^\top k_j}{\sqrt{d_k}} + B_{ij}^{(\ell)}.$$

25.4.2 Bucketed Relative Position

For relative distance $d = j - i$, prepare learnable bias table $\beta^{(\ell)} \in \mathbb{R}^{N_{\text{bucket}}}$:

$$B_{ij}^{(\ell)} = \beta_{\text{bucket}(j-i)}^{(\ell)}.$$

Bucket function discretizes finely for short distances, coarsely for long distances.

25.5 Pretraining Objective: Span Corruption

T5 pretraining uses span corruption task, extending BERT's MLM.

25.5.1 Span Masking as a Random Process

For input sentence $x_{1:T}$, randomly select spans with average length λ_{span}, replace with special tokens $[\texttt{MASK}_k]$.

25.5.2 Target Sequence Construction

Target $y_{1:T_y}$ concatenates masked spans separated by sentinel tokens:
Example:

$x =$ "Thank you for inviting me to your party last week."
$\tilde{x} =$ "Thank you [X] me to your [Y] week."
$y =$ "[X] for inviting [Y] party last [Z]."

25.5.3 Denoising Objective as Conditional Likelihood

Pretraining loss:

$$\mathcal{L}_{\text{denoise}}(\theta) = \mathbb{E}_{x_{1:T},\text{spans}} \left[-\sum_{s=1}^{T_y} \log p_\theta(y_s \mid \tilde{x}_{1:T'}, y_{<s}) \right].$$

25.5.4 Per-Token Loss and Gradient

Logits at position s:

$$z_s = W_{\text{LM}} h_s^{(L_{\text{dec}})} + b_{\text{LM}} \in \mathbb{R}^{|\mathcal{V}|},$$

Gradient:

$$\nabla_{z_s} \ell_s = p_s - \text{one\_hot}(y_s^\star).$$

25.6 Teacher Forcing and Autoregressive Decoding

25.6.1 Training with Teacher Forcing

During training, ground truth output $y_{1:T_y}$ is directly fed to decoder (teacher forcing). This enables parallel computation for efficient training.

25.6.2 Inference with Autoregressive Generation

During inference, decoder is run recursively until $y_s = [\texttt{EOS}]$.

25.7 Simplified Layer Normalization (RMSNorm)

T5 uses RMSNorm:

$$\text{RMSNorm}(x) = \gamma \odot \frac{x}{\text{RMS}(x) + \varepsilon}, \quad \text{RMS}(x) = \sqrt{\frac{1}{d_{\text{model}}} \sum_{i=1}^{d_{\text{model}}} x_i^2}.$$

25.8 Computational Complexity

25.8.1 Encoder Complexity

For input length T_x, encoder complexity is:

$$O\big(L_{\text{enc}} \cdot T_x^2 d_k + L_{\text{enc}} \cdot T_x d_{\text{model}} d_{\text{ff}}\big).$$

25.8.2 Decoder Complexity

For output length T_y: self-attention $O(T_y^2 d_k)$, cross-attention $O(T_y T_x d_k)$, FFN $O(T_y d_{\text{model}} d_{\text{ff}})$.

25.9 Training Strategies and Hyperparameters

25.9.1 Pre-Training Corpus: C4

T5 is pretrained on Colossal Clean Crawled Corpus (C4) (approximately 750GB of clean web text).

25.9.2 Model Sizes

Model	d_{model}	d_{ff}	Parameters
T5-Small	512	2048	60M
T5-Base	768	3072	220M
T5-Large	1024	4096	770M
T5-3B	1024	16384	3B
T5-11B	1024	65536	11B

25.10 Comparison with BERT and GPT

Model	Architecture	Pretraining Objective	Target Tasks
BERT	Encoder-only	MLM + NSP	Classification/Extraction
GPT	Decoder-only	Causal LM	Generation
T5	Encoder-Decoder	Span corruption	All tasks unified

25.11 Summary

T5:

- Unifies all NLP tasks as conditional generation $p_\theta(y_{1:T_y} \mid x_{1:T_x})$,
- Introduces relative position bias to Encoder-Decoder Transformer,
- Performs denoising pretraining via span corruption,
- Achieves general-purpose transfer learning through text-to-text framework,

demonstrating the possibility of unified modeling across diverse NLP tasks.

Chapter 26

Vision Transformer (ViT): Transformers for Image Classification

26.1 Architecture Overview

Figure 26.1 illustrates the Vision Transformer architecture, which applies a standard Transformer encoder directly to sequences of image patches.

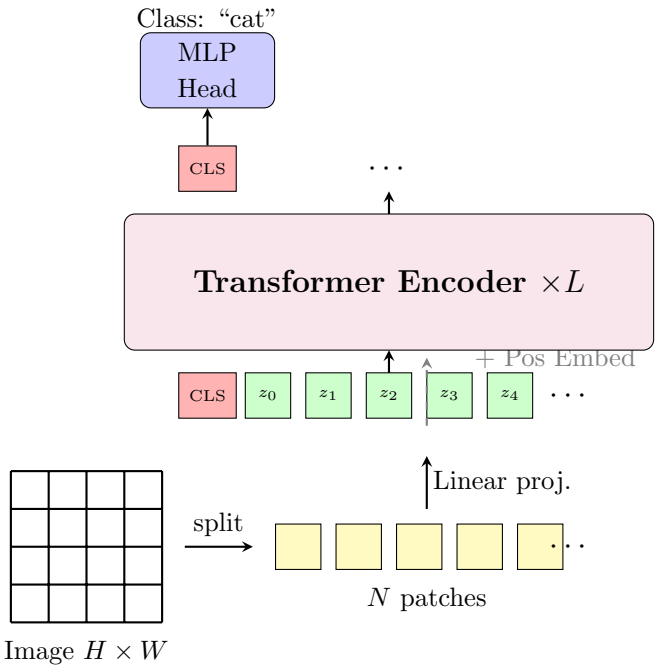

Figure 26.1: Vision Transformer (ViT) Architecture: An image is split into fixed-size patches, linearly embedded, and processed by a standard Transformer encoder. The [CLS] token’s output is used for classification.

26.1.1 Intuitive Understanding

Images as Sequences: ViT's key insight is treating an image not as a 2D grid, but as a sequence of patches—just like a sentence is a sequence of words. A 224×224 image with 16×16 patches becomes a sequence of 196 "visual tokens."

Why Patches? Self-attention has $O(n^2)$ complexity. Using individual pixels (224×224 = 50,176 tokens) would be prohibitive. Patches (14×14 = 196 tokens) make it tractable while preserving local structure within each patch.

No Convolutions: ViT demonstrates that the inductive biases of CNNs (locality, translation invariance) are not strictly necessary. Given enough data, self-attention can learn these patterns from scratch, and potentially discover more flexible representations.

The [CLS] Token: Borrowed from BERT, a learnable "class" token is prepended to the patch sequence. After passing through all layers, this token's representation contains a global summary of the image, which is fed to a classification head.

Data Hunger: Without CNN's inductive biases, ViT requires massive datasets (ImageNet-21k, JFT-300M) for pretraining. On smaller datasets, CNNs still outperform ViT. This highlights the trade-off between flexibility and data efficiency.

Position Embeddings: Unlike CNNs that inherently understand spatial relationships, ViT must learn positions via learnable embeddings. Interestingly, these learned embeddings exhibit 2D spatial structure matching patch positions.

26.2 Motivation: From Convolution to Self-Attention

Traditional image classification was dominated by convolutional neural networks (CNN) with local feature extraction and hierarchical representation learning. Vision Transformer (ViT) treats images as sequence data, directly applying standard Transformer Encoder, removing CNN's inductive bias (locality, translation invariance), enabling learning of global dependencies via pure self-attention mechanism.

26.3 Image as a Sequence: Patch Embedding

26.3.1 Image Partitioning into Patches

Let input image be $I \in \mathbb{R}^{H\times W\times C}$. Partition image into $P \times P$ pixel square patches. Number of patches:

$$N = \frac{H \cdot W}{P^2}.$$

Typically $H = W = 224$, $P = 16$ gives $N = 196$.

26.3.2 Patch Extraction as Tensor Reshaping

Partition image I into N patch vectors $\{\mathbf{x}_p^{(i)}\}_{i=1}^N$, $\mathbf{x}_p^{(i)} \in \mathbb{R}^{P^2C}$.

26.3.3 Linear Projection to Embedding Dimension

Linearly project each patch vector to model dimension d_{model}:

$$\mathbf{z}_i^{(0)} = E\mathbf{x}_p^{(i)} + \mathbf{b}, \quad E \in \mathbb{R}^{d_{\text{model}}\times(P^2C)}.$$

26.4 Prepending the Class Token

For classification tasks, add learnable class token $\mathbf{z}_{\text{cls}} \in \mathbb{R}^{d_{\text{model}}}$ at the beginning of sequence:

$$Z_{\text{full}}^{(0)} = \begin{bmatrix} \mathbf{z}_{\text{cls}}^\top \\ (\mathbf{z}_1^{(0)})^\top \\ \vdots \\ (\mathbf{z}_N^{(0)})^\top \end{bmatrix} \in \mathbb{R}^{(N+1)\times d_{\text{model}}}.$$

26.5 Positional Encoding

26.5.1 Learnable 1D Position Embedding

Add learnable position embedding for each position $i \in \{0, 1, \ldots, N\}$:

$$\mathbf{z}_i^{(0)\prime} = \mathbf{z}_i^{(0)} + \mathbf{e}_{\text{pos}}^{(i)}.$$

Position embedding matrix $E_{\text{pos}} \in \mathbb{R}^{(N+1)\times d_{\text{model}}}$ is learned from training.

26.5.2 Input Embedding Composition

Final input embedding:

$$H^{(0)} = Z_{\text{full}}^{(0)} + E_{\text{pos}} \in \mathbb{R}^{(N+1)\times d_{\text{model}}}.$$

26.6 Transformer Encoder

ViT stacks L layers of standard Transformer Encoder.

26.6.1 Multi-Head Self-Attention

For layer ℓ input $H^{(\ell-1)} \in \mathbb{R}^{(N+1)\times d_{\text{model}}}$, ViT uses bidirectional attention without mask:

$$A_{ij}^{(h)} = \frac{\exp(S_{ij}^{(h)})}{\sum_{k=0}^{N} \exp(S_{ik}^{(h)})}.$$

26.6.2 Layer Normalization and Residual Connections

In Pre-LN form:

$$\hat{H}^{(\ell)} = H^{(\ell-1)} + \text{MHA}\big(\text{LN}(H^{(\ell-1)})\big),$$
$$H^{(\ell)} = \hat{H}^{(\ell)} + \text{FFN}\Big(\text{LN}(\hat{H}^{(\ell)})\Big).$$

26.6.3 Position-Wise Feed-Forward Network

ViT uses GELU as activation function:

$$\text{FFN}(u) = W_2\,\text{GELU}(W_1 u + b_1) + b_2,$$
$$\text{GELU}(x) = x\Phi(x) = x \cdot \frac{1}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right].$$

26.7 Classification Head

26.7.1 Extracting the CLS Token

Extract CLS token representation $\mathbf{h}_{\text{cls}}^{(L)} \in \mathbb{R}^{d_{\text{model}}}$ from final layer L output.

26.7.2 Linear Classification Layer

For K-class classification:

$$\mathbf{z}_{\text{cls}} = W_{\text{head}}\mathbf{h}_{\text{cls}}^{(L)} + \mathbf{b}_{\text{head}} \in \mathbb{R}^K,$$

Prediction distribution:

$$p_\theta(y = k \mid I) = \frac{\exp(z_{\text{cls},k})}{\sum_{j=1}^{K} \exp(z_{\text{cls},j})}.$$

26.7.3 Cross-Entropy Loss

For 1-hot label $\mathbf{y} \in \{0, 1\}^K$:

$$\mathcal{L}_{\text{CE}}(\theta) = -\sum_{k=1}^{K} y_k \log p_\theta(y = k \mid I).$$

Gradient:

$$\nabla_{\mathbf{z}_{\text{cls}}} \mathcal{L}_{\text{CE}} = p_\theta - \mathbf{y}, \quad \nabla_{W_{\text{head}}} \mathcal{L}_{\text{CE}} = (p_\theta - \mathbf{y})(\mathbf{h}_{\text{cls}}^{(L)})^\top.$$

26.8 Pre-Training and Transfer Learning

26.8.1 Pre-Training on Large Datasets

ViT achieves high performance through pretraining on large-scale datasets:

- ImageNet-21k: approximately 14 million images, 21,000 classes,
- JFT-300M: approximately 300 million images.

26.8.2 Fine-Tuning on Target Datasets

For downstream tasks, reinitialize classification head and update all parameters. When fine-tuning on higher resolution images, 2D interpolate position embeddings.

26.9 Model Variants and Scaling

Model	Layers L	Hidden dim d_{model}	Heads H	Parameters
ViT-Base	12	768	12	86M
ViT-Large	24	1024	16	307M
ViT-Huge	32	1280	16	632M

26.10 Computational Complexity

26.10.1 Self-Attention Complexity

Self-attention for sequence length $N + 1$:

$$O\big((N+1)^2 d_k\big) = O\left(\left(\frac{HW}{P^2}\right)^2 d_k\right).$$

Larger P means smaller N, reducing computation.

26.10.2 Comparison with CNN

CNN is linear in image size due to local receptive fields. ViT is $O((HW/P^2)^2)$, controllable by patch size P.

26.11 Inductive Bias and Data Efficiency

CNN has inductive bias of locality and translation invariance, while ViT directly learns global dependencies across all patches via self-attention, lacking these biases.

Due to weak inductive bias, ViT underperforms CNN without large-scale dataset pretraining.

26.12 Extensions and Variants

26.12.1 DeiT (Data-efficient image Transformer)

Uses distillation to improve training efficiency on smaller datasets.

26.12.2 Swin Transformer

Hierarchical structure and shifted window self-attention for efficient computation with locality.

26.12.3 BEiT

BERT-style self-supervised learning by masking and predicting image patches.

26.13 Summary

Vision Transformer (ViT):

- Partitions image into $P \times P$ patches and embeds via linear projection,
- Adds CLS token and learnable position embeddings,
- Applies standard Transformer Encoder, learning global dependencies via bidirectional self-attention,

- Predicts via linear classification head from CLS token representation,
- Achieves performance exceeding CNN through large-scale pretraining,

establishing the paradigm of directly applying Transformers to vision tasks.

Chapter 27

PaLM: Pathways Language Model

27.1 Overview and Scaling Philosophy

PaLM (Pathways Language Model) is an ultra-large-scale decoder-only language model developed by Google, with the largest configuration having **540B parameters**. While following the autoregressive language model design of the GPT family, it improves efficiency and performance through the following innovations:

- **Pathways system**: Ultra-parallel distributed training across thousands of TPUs,
- **SwiGLU activation**: Improved FFN expressiveness,
- **Parallel layers**: Parallel execution of Attention and FFN,
- **Multi-query attention**: Efficient KV cache,
- **RoPE**: Rotary Position Embedding for relative positions.

27.2 Autoregressive Language Modeling

PaLM learns the autoregressive distribution of token sequences $x_{1:T}$:

$$p_\theta(x_{1:T}) = \prod_{t=1}^{T} p_\theta(x_t \mid x_{<t}),$$

Training objective is negative log-likelihood minimization:

$$\mathcal{L}_{\text{LM}}(\theta) = \mathbb{E}_{x_{1:T}\sim\mathcal{D}}\left[-\frac{1}{T}\sum_{t=1}^{T}\log p_\theta(x_t \mid x_{1:t-1})\right].$$

27.3 Parallel Layers Architecture

In standard Transformer, each layer is sequential: Attention $\to$ Add&Norm $\to$ FFN $\to$ Add&Norm. PaLM parallelizes Attention and FFN:

$$H^{(\ell)} = H^{(\ell-1)} + \text{Attention}^{(\ell)}\big(\text{LN}(H^{(\ell-1)})\big) + \text{FFN}^{(\ell)}\big(\text{LN}(H^{(\ell-1)})\big).$$

27.4 Multi-Query Attention

Multi-query attention shares K, V across all heads:

$$Q^{(h)} = HW_Q^{(h)} \in \mathbb{R}^{T \times d_k}, \quad h = 1, \ldots, H,$$

$$K = HW_K \in \mathbb{R}^{T \times d_k}, \quad V = HW_V \in \mathbb{R}^{T \times d_v},$$

where W_K, W_V are single matrices shared across heads. KV cache size is reduced from $O(H \cdot T \cdot d_k)$ to $O(T \cdot d_k)$.

27.5 RoPE: Rotary Position Embedding

RoPE embeds relative position information into attention scores via rotation matrices. For position t, apply rotation:

$$\tilde{q}_t = R_t q_t, \quad \tilde{k}_t = R_t k_t,$$

where R_t is a block-diagonal rotation matrix. The score becomes:

$$S_{ij} = \frac{q_i^\top R_{j-i} k_j}{\sqrt{d_k}},$$

depending only on relative position $j - i$.

27.6 SwiGLU Activation

PaLM adopts SwiGLU:

$$\text{SwiGLU}(x) = (W_1 x) \odot \text{Swish}(W_2 x),$$

$$\text{Swish}(z) = z \cdot \sigma(z) = \frac{z}{1 + e^{-z}}.$$

27.7 Model Configurations

Model	Layers	d_{model}	Heads	d_{ff}	Params
PaLM-8B	32	4096	32	16384	8B
PaLM-62B	64	8192	64	32768	62B
PaLM-540B	118	18432	48	49152	540B

27.8 Summary

PaLM achieves scaling and efficiency through RoPE, SwiGLU, parallel layers, multi-query attention, and Pathways distributed training across thousands of TPUs.

Chapter 28

LLaMA: Large Language Model Meta AI

28.1 Overview and Design Philosophy

LLaMA (Large Language Model Meta AI) is an open-source large-scale decoder-only language model developed by Meta. Key design features include:

- **RMSNorm**: Simplified LayerNorm for efficiency,
- **SwiGLU activation**: Improved FFN expressiveness,
- **RoPE**: Efficient relative position encoding,
- **Pre-normalization**: Training stability,
- **Public data only**: Transparency and reproducibility.

LLaMA provides 4 sizes: 7B, 13B, 33B, and 65B.

28.2 RMSNorm: Root Mean Square Layer Normalization

LLaMA adopts RMSNorm:

$$\text{RMSNorm}(x) = \gamma \odot \frac{x}{\text{RMS}(x)},$$

$$\text{RMS}(x) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \varepsilon},$$

where $\gamma \in \mathbb{R}^d$ is a learnable scale parameter. Unlike LayerNorm, RMSNorm omits mean shift and bias term.

28.3 Pre-Normalization Architecture

LLaMA uses Pre-LN structure:

$$\tilde{H}^{(\ell)} = H^{(\ell-1)} + \text{Attention}^{(\ell)}\big(\text{RMSNorm}(H^{(\ell-1)})\big)\,,$$

$$H^{(\ell)} = \tilde{H}^{(\ell)} + \text{FFN}^{(\ell)}\Big(\text{RMSNorm}(\tilde{H}^{(\ell)})\Big)\,.$$

28.4 Causal Self-Attention with RoPE

For query and key at position t, apply rotation matrix R_t:

$$\tilde{Q}_t^{(h)} = R_t Q_t^{(h)}, \quad \tilde{K}_t^{(h)} = R_t K_t^{(h)}.$$

Score depends only on relative position:

$$S_{ij}^{(h)} = \frac{(Q_i^{(h)})^\top R_{j-i} K_j^{(h)}}{\sqrt{d_k}}.$$

28.5 SwiGLU Feed-Forward Network

$$\text{SwiGLU}(x) = (W_1 x) \odot \text{Swish}(W_2 x),$$
$$\text{FFN}(x) = W_3\,\text{SwiGLU}(x).$$

No bias terms are used (LLaMA omits biases in all layers).

28.6 Model Configurations

Model	Layers	d_{model}	Heads	d_{ff}	Params
LLaMA-7B	32	4096	32	11008	6.7B
LLaMA-13B	40	5120	40	13824	13.0B
LLaMA-33B	60	6656	52	17920	32.5B
LLaMA-65B	80	8192	64	22016	65.2B

28.7 Training Data

LLaMA is trained on public datasets only (approximately 1.4T tokens): CommonCrawl (67%), C4 (15%), GitHub (4.5%), Wikipedia (4.5%), Books (4.5%), ArXiv (2.5%), StackExchange (2%).

28.8 LLaMA 2 Improvements

LLaMA 2 introduces:

- Grouped-Query Attention (GQA): Balance between multi-head and multi-query,
- Longer context window: $T_{\max} = 4096$,
- RLHF for safety alignment.

28.9 Summary

LLaMA achieves efficient and transparent large-scale language modeling through RMSNorm, SwiGLU, RoPE, pre-normalization, and public data training.

Chapter 29

Mixtral: Sparse Mixture of Experts Language Model

29.1 Architecture Overview

Figure 29.1 illustrates the Sparse Mixture of Experts (MoE) architecture used in Mixtral.

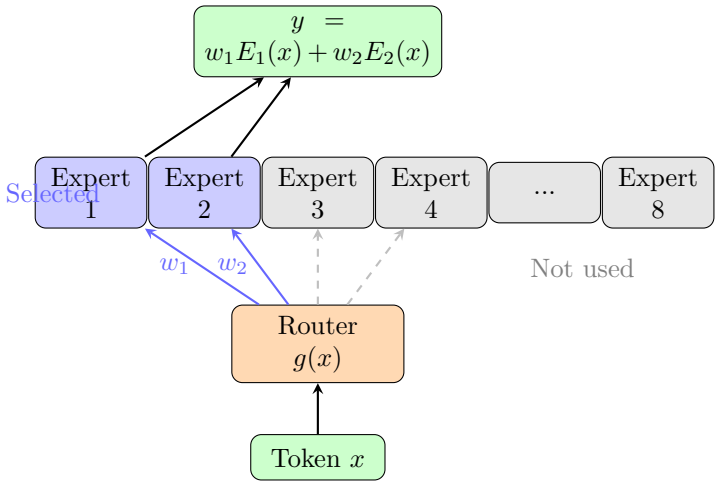

Figure 29.1: Mixtral Sparse MoE Architecture: A gating network routes each token to the top-2 experts (out of 8). Only selected experts are computed, achieving 47B total parameters with 13B active computation.

29.1.1 Intuitive Understanding

The Expert Analogy: Imagine a hospital with specialists: cardiologists, neurologists, dermatologists, etc. When a patient arrives, a triage system (router) directs them to the 1-2 most relevant specialists, not all doctors. Similarly, MoE routes each token to experts best suited for it.

Sparse = Efficient: With 8 experts but only 2 active per token, Mixtral uses $\frac{2}{8} = 25\%$ of expert computation. Total parameters (47B) provide capacity; active parameters (13B) determine speed. This decouples model capacity from inference cost.

Why It Works: Different tokens may need different computations:

- Code tokens → Expert specializing in programming patterns

- Math tokens → Expert specializing in numerical reasoning
- Language tokens → Expert specializing in linguistic patterns

The router learns to make these assignments automatically.

Load Balancing Challenge: Without constraints, the router might always pick the same experts ("expert collapse"). The auxiliary loss encourages uniform expert utilization.

Key Insight: Mixtral shows that conditional computation (using different parameters for different inputs) is a promising scaling direction, achieving better quality per FLOP than dense models.

29.2 Sparse MoE Architecture Details

Mixtral is a large-scale language model with Sparse Mixture of Experts (SMoE) architecture developed by Mistral AI. Key features:

- **8 expert FFNs**: Multiple specialist networks per layer,
- **Top-2 routing**: Select optimal 2 experts per input,
- **Total 47B parameters**: Sum of all expert parameters,
- **Active 13B parameters**: Actually used during inference,
- **Load balancing**: Auxiliary loss for equal expert utilization.

This achieves 47B parameter expressiveness with 13B model computation cost.

29.3 Mixture of Experts Formulation

$N = 8$ expert networks, each with SwiGLU structure:

$$\text{Expert}_e(x) = W_{3,e}\,\text{SwiGLU}(x).$$

Gating network computes routing probabilities:

$$g(x) = \text{softmax}(W_g x) \in \mathbb{R}^N.$$

29.4 Top-k Routing

Top-2 routing ($k = 2$) selects the two highest-scoring experts:

$$\mathcal{T}_2(x) = \{e_1, e_2\},$$

$$y = \tilde{g}_{e_1}(x) \cdot \text{Expert}_{e_1}(x) + \tilde{g}_{e_2}(x) \cdot \text{Expert}_{e_2}(x),$$

where $\tilde{g}_e$ are renormalized weights over selected experts.

Computation is reduced to $k/N = 2/8 = 1/4$ of dense MoE.

29.5 Load Balancing Loss

To prevent expert load imbalance, auxiliary loss is introduced:

$$f_e = \frac{1}{B}\sum_{i=1}^{B} g_e(x_i), \quad P_e = \frac{1}{B}\sum_{i=1}^{B} \mathbb{1}\{e \in \mathcal{T}_k(x_i)\},$$

$$\mathcal{L}_{\text{balance}} = \alpha \cdot N \sum_{e=1}^{N} f_e \cdot P_e.$$

Total loss:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{LM}} + \mathcal{L}_{\text{balance}}.$$

29.6 Sliding Window Attention

Mixtral uses sliding window attention for long context (32k tokens): Window size $w = 4096$, position i attends to $[i - w, i]$. Reduces attention from $O(T^2 d_k)$ to $O(Twd_k)$.

29.7 Parameter Count

- **Total**: 47B parameters (8 experts per layer),
- **Active**: 13B parameters (top-2 experts).

29.8 Comparison with Dense Models

Model	Total params	Active params	Inference cost
LLaMA-2-13B	13B	13B	1.0×
LLaMA-2-70B	70B	70B	5.4×
Mixtral-8x7B	47B	13B	1.0×

Mixtral achieves 70B model performance with 13B computation cost.

29.9 Summary

Mixtral achieves breakthrough efficiency through:

- Sparse MoE architecture with 8 expert FFNs,
- Top-2 routing selecting optimal 2 experts per input,
- 47B total / 13B active parameters,
- Load balancing loss for expert utilization,
- Sliding window attention for 32k context,
- LLaMA-style RMSNorm, SwiGLU, RoPE.

Mixtral demonstrates that Sparse MoE is a promising direction for future LLM scaling.

Chapter 30

Complete Backpropagation Walkthroughs: RNN to Transformer

30.1 Part I: Simple RNN - Complete Backpropagation Example

30.1.1 Setup: 2-Layer RNN with Concrete Numbers

Architecture:

- Input dimension: $d_x = 2$
- Hidden dimension: $d_h = 3$
- Output dimension: $d_y = 1$ (binary classification)
- Sequence length: $T = 3$
- Learning rate: $\alpha = 0.1$

Parameters:

$$\mathbf{W}_{hh} = \begin{pmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.1 & 0.2 \\ 0.2 & 0.3 & 0.1 \end{pmatrix}, \quad \mathbf{W}_{xh} = \begin{pmatrix} 0.5 & 0.6 \\ 0.3 & 0.4 \\ 0.2 & 0.1 \end{pmatrix} \tag{30.1}$$

$$\mathbf{W}_{hy} = \begin{pmatrix} 0.7 & 0.5 & 0.3 \end{pmatrix}, \quad \mathbf{b}_h = \begin{pmatrix} 0.1 \\ 0.0 \\ 0.1 \end{pmatrix}, \quad b_y = 0.05 \tag{30.2}$$

Activation functions: Hidden: $\sigma_h(z) = \tanh(z)$, Output: $\sigma_y(z) = \sigma(z) = \frac{1}{1+e^{-z}}$.
Input sequence:

$$\mathbf{x}^{(1)} = \begin{pmatrix} 0.5 \\ 0.2 \end{pmatrix}, \quad \mathbf{x}^{(2)} = \begin{pmatrix} 0.3 \\ 0.4 \end{pmatrix}, \quad \mathbf{x}^{(3)} = \begin{pmatrix} 0.2 \\ 0.5 \end{pmatrix} \tag{30.3}$$

Target: $y = 1$.

30.1.2 Forward Propagation

Timestep 1

$$\mathbf{z}_h^{(1)} = \mathbf{W}_{hh}\mathbf{h}^{(0)} + \mathbf{W}_{xh}\mathbf{x}^{(1)} + \mathbf{b}_h \tag{30.4}$$

$$= \mathbf{0} + \begin{pmatrix} 0.47 \\ 0.23 \\ 0.22 \end{pmatrix} \quad \text{(after calc)} \tag{30.5}$$

$$\mathbf{h}^{(1)} = \tanh(\mathbf{z}_h^{(1)}) \approx \begin{pmatrix} 0.440 \\ 0.226 \\ 0.216 \end{pmatrix} \tag{30.6}$$

$$\hat{y}^{(1)} = \sigma(\mathbf{W}_{hy}\mathbf{h}^{(1)} + b_y) = \sigma(0.5358) \approx 0.631 \tag{30.7}$$

Timestep 2

$$\mathbf{z}_h^{(2)} = \begin{pmatrix} 0.645 \\ 0.452 \\ 0.357 \end{pmatrix}, \quad \mathbf{h}^{(2)} \approx \begin{pmatrix} 0.567 \\ 0.422 \\ 0.343 \end{pmatrix}, \quad \hat{y}^{(2)} \approx 0.682 \tag{30.8}$$

Timestep 3

$$\mathbf{z}_h^{(3)} \approx \begin{pmatrix} 0.812 \\ 0.641 \\ 0.485 \end{pmatrix}, \quad \mathbf{h}^{(3)} \approx \begin{pmatrix} 0.675 \\ 0.568 \\ 0.449 \end{pmatrix}, \quad \hat{y}^{(3)} \approx 0.720 \tag{30.9}$$

30.1.3 Loss Calculation

Binary Cross-Entropy ($y = 1$):

$$L = -\log(0.720) \approx 0.329 \tag{30.10}$$

30.1.4 Backward Propagation Through Time (BPTT)

Output Delta (Timestep 3)

$$\boldsymbol{\delta}_y^{(3)} = \hat{y}^{(3)} - y = 0.720 - 1 = -0.280 \tag{30.11}$$

Hidden Delta (Timestep 3)

$$\boldsymbol{\delta}_h^{(3)} = (\mathbf{W}_{hy}^T \boldsymbol{\delta}_y^{(3)}) \odot \sigma_h'(\mathbf{z}_h^{(3)}) \tag{30.12}$$

$$= \begin{pmatrix} -0.196 \\ -0.140 \\ -0.084 \end{pmatrix} \odot \begin{pmatrix} 0.544 \\ 0.677 \\ 0.798 \end{pmatrix} = \begin{pmatrix} -0.107 \\ -0.095 \\ -0.067 \end{pmatrix} \tag{30.13}$$

Hidden Delta (Timestep 2)

Gradient only from recurrent connection (assuming output loss only at $T = 3$ for simplification):

$$\boldsymbol{\delta}_h^{(2)} = (\mathbf{W}_{hh}^T \boldsymbol{\delta}_h^{(3)}) \odot \sigma_h'(\mathbf{z}_h^{(2)}) \tag{30.14}$$

$$= \begin{pmatrix} -0.0621 \\ -0.051 \\ -0.0578 \end{pmatrix} \odot \begin{pmatrix} 0.679 \\ 0.822 \\ 0.882 \end{pmatrix} = \begin{pmatrix} -0.0422 \\ -0.0419 \\ -0.0510 \end{pmatrix} \tag{30.15}$$

Hidden Delta (Timestep 1)

$$\boldsymbol{\delta}_h^{(1)} \approx \begin{pmatrix} -0.0252 \\ -0.0265 \\ -0.0157 \end{pmatrix} \tag{30.16}$$

Parameter Gradients (Summed)

$$\frac{\partial L}{\partial \mathbf{W}_{hh}} \approx \begin{pmatrix} -0.0792 & -0.0547 & -0.0458 \\ -0.0723 & -0.0496 & -0.0416 \\ -0.0604 & -0.0398 & -0.0340 \end{pmatrix} \tag{30.17}$$

30.2 Part II: LSTM - Complete Backpropagation Example

30.2.1 Setup

Input $\mathbf{x} \in \mathbb{R}^2$, Hidden $\mathbf{h} \in \mathbb{R}^2$. 2 Timesteps. Simplified weights: $\mathbf{W} \approx 0.1\mathbf{I}$.

30.2.2 Forward Pass (Abstract)

For each timestep t:

- Gates: $\mathbf{f}, \mathbf{i}, \mathbf{o}, \tilde{\mathbf{c}}$ computed via linear map + sigmoid/tanh.
- Cell: $\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \odot \tilde{\mathbf{c}}^{(t)}$
- Hidden: $\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \odot \tanh(\mathbf{c}^{(t)})$

30.2.3 Backward Pass Logic

The crucial difference from RNN is the gradient flow through $\mathbf{c}$.

$$\frac{\partial L}{\partial \mathbf{c}^{(t)}} = \frac{\partial L}{\partial \mathbf{h}^{(t)}} \odot \mathbf{o}^{(t)} \odot (1 - \tanh^2 \mathbf{c}^{(t)}) + \frac{\partial L}{\partial \mathbf{c}^{(t+1)}} \odot \mathbf{f}^{(t+1)} \tag{30.18}$$

The term $\odot \mathbf{f}^{(t+1)}$ allows gradients to persist without decay if $\mathbf{f} \approx 1$.

30.3 Part III: Transformer - Complete Backpropagation Example

30.3.1 Setup: Minimal Attention

$T = 2, d_{\text{model}} = 4, h = 2$. $\mathbf{X} \in \mathbb{R}^{4\times 2}$ (features $\times$ tokens, noting text usually uses $T \times d$).

30.3.2 Forward: Multi-Head Attention

Projections: $\mathbf{Q}_1 = \mathbf{W}_{Q_1}\mathbf{X}$. **Scores:** $\mathbf{E}_1 = \mathbf{Q}_1\mathbf{K}_1^T/\sqrt{d_k}$. **Softmax:** $\mathbf{A}_1 = \text{softmax}(\mathbf{E}_1)$ (column-wise for tokens). **Output:** $\text{head}_1 = \mathbf{A}_1\mathbf{V}_1^T$.

30.3.3 Backward: Gradients

Gradient through Value:

$$\frac{\partial L}{\partial \mathbf{V}_1} = \frac{\partial L}{\partial \text{head}_1}\mathbf{A}_1 \tag{30.19}$$

Gradient through Attention Matrix:

$$\frac{\partial L}{\partial \mathbf{A}_1} = \left(\frac{\partial L}{\partial \text{head}_1}\right)^T \mathbf{V}_1 \tag{30.20}$$

Gradient through Softmax (Jacobian): For each column j (token scores):

$$\frac{\partial A_{ij}}{\partial E_{kj}} = A_{ij}(\delta_{ik} - A_{kj}) \tag{30.21}$$

This connects $\partial L/\partial \mathbf{A}$ to $\partial L/\partial \mathbf{E}$.

Gradient through Q, K:

$$\frac{\partial L}{\partial \mathbf{Q}_1} \propto \frac{\partial L}{\partial \mathbf{E}_1}\mathbf{K}_1 \tag{30.22}$$

30.4 Summary of Backprop Complexity

Component	Forward Complexity	Key Challenge
RNN	$O(d_h^2)$	Vanishing gradients
LSTM	$O(4d_h^2)$	Additive path stability
Attention	$O(T^2 d)$	Softmax Jacobian structure

Method	Computes	Statistics on	Best for
Batch Norm	μ_B, σ_B	Batch	Large batch, CNNs
Layer Norm	μ, σ per sample	Layer features	RNNs, Transformers
Group Norm	μ, σ per group	Groups of channels	Small batch
Instance Norm	μ, σ per channel	Channel	Style transfer