

Mathematical Foundations of Neural Networks (Vol 1)

A Rigorous Treatment with Detailed Derivations

January 21, 2026

Contents

1 Mathematical Preliminaries	13
1.1 Vectors and Matrices	13
1.1.1 Vector Spaces and Norms	13
1.1.2 Dot Product and Geometric Interpretation	13
1.1.3 Matrix Operations	14
1.1.4 Important Matrix Properties	14
1.2 Calculus: Foundations of Optimization	15
1.2.1 Derivatives and the Chain Rule	15
1.2.2 Partial Derivatives and Gradients	15
1.2.3 Matrix Calculus	16
Notation, Dimensions, and Label Conventions	17
2 The Perceptron	21
2.1 Decision Boundary as a Hyperplane	21
2.1.1 Hyperplane interpretation	21
2.1.2 Scaling invariance	21
2.2 Signed Distance to the Hyperplane	22
2.2.1 Derivation	22
2.3 Perceptron Learning Algorithm	22
2.3.1 Label convention	22
2.3.2 Update rule	22
2.4 Perceptron Convergence Theorem (Sketch)	23
2.4.1 Linear separability and margin	23
2.4.2 Theorem	23
2.4.3 Proof sketch	23
2.4.4 Remarks	24
3 Feedforward Neural Networks	25
3.1 From Scalar Sums to Matrix Form	25
3.1.1 Single neuron (scalar form)	25
3.1.2 Layer of neurons (summation index notation)	25
3.1.3 Layer of neurons (matrix form)	25
3.1.4 Mini-batch (fully vectorized) form	26
3.2 Network as Function Composition	26
3.2.1 Parameter count	26
3.3 Activation Functions and Derivatives	26
3.3.1 Why nonlinearity is required	27

3.3.2	Sigmoid	27
3.3.3	Hyperbolic tangent	27
3.3.4	ReLU	27
3.3.5	Softmax	27
3.3.6	Vanishing gradients (preview)	28
3.4	A Fully Worked Tiny Example (Optional)	28
4	Loss Functions	29
4.1	Empirical Risk Minimization	29
4.2	Regression: Mean Squared Error	29
4.2.1	Definition	29
4.2.2	Gradient w.r.t. the prediction	29
4.3	Binary Classification: Binary Cross-Entropy	30
4.3.1	Binary cross-entropy (negative log-likelihood)	30
4.3.2	Gradient w.r.t. \hat{y}	30
4.3.3	Sigmoid + BCE simplification (logit gradient)	30
4.4	Multi-class Classification: Softmax and Categorical Cross-Entropy	30
4.4.1	Categorical cross-entropy	30
4.4.2	Softmax Jacobian	31
4.4.3	Softmax + CCE simplification (logit gradient)	31
4.5	(Optional) Huber Loss for Robust Regression	31
5	Backpropagation Algorithm	33
5.1	Setup: Notation and Forward Pass	33
5.2	The Delta Terms	33
5.2.1	Definition	33
5.2.2	Output layer delta: general form	33
5.2.3	Hidden layer delta	34
5.3	Gradients for Weights and Biases	34
5.3.1	Single example	34
5.3.2	Mini-batch (average gradient)	34
5.3.3	Mini-batch matrix backpropagation (vectorized deltas)	35
5.4	Two Classic “Cancellations”	36
5.4.1	Sigmoid + Binary Cross-Entropy	36
5.4.2	Softmax + Categorical Cross-Entropy	36
5.5	Vanishing Gradients (Why Depth Is Hard)	37
5.5.1	Mitigations (mathematical view)	37
5.6	Algorithm Summary (One Iteration)	37
6	Concrete Numerical Example: A Network from Scratch	39
6.1	Problem Setup	39
6.2	Parameter Initialization	39
6.3	Forward Propagation: General Form	40
6.4	Forward Propagation: Sample 1 (Fully Expanded)	40
6.4.1	Layer 1 pre-activation	40
6.4.2	Layer 1 activation (ReLU)	40
6.4.3	Layer 2 pre-activation	40
6.4.4	Output (sigmoid)	41
6.4.5	Loss for sample 1	41

6.5	Forward Propagation: Sample 2 (Detailed)	41
6.5.1	Layer 1	41
6.5.2	Layer 2 and output	41
6.6	Batch Loss	42
6.7	Backpropagation: General Form	42
6.8	Backpropagation: Sample 1 (Fully Expanded)	42
6.8.1	Output delta	42
6.8.2	Gradients for layer 2	43
6.8.3	Hidden delta	43
6.8.4	Gradients for layer 1	43
6.9	Mini-batch Gradients (Averaging)	43
6.10	Parameter Updates (SGD)	44
6.10.1	Update layer 2	44
6.10.2	Update layer 1	44
6.11	Second Iteration (Forward Pass Check)	44
7	Advanced Numerical Demonstrations	45
7.1	Optimization Dynamics	45
7.1.1	Effect of the learning rate	45
7.1.2	Loss curve (illustrative)	46
7.2	Regularization Example: L2	46
7.2.1	Definition	46
7.2.2	Concrete computation	46
7.2.3	Gradient effect (weight decay view)	46
7.3	Momentum Optimization	47
7.3.1	Update rule	47
7.3.2	Two-step numerical example (Layer 2 weights)	47
7.4	Batch Normalization (Numerical Calculation)	47
7.4.1	Definition	47
7.4.2	Concrete computation for one neuron	48
7.5	Dropout Regularization (Numerical Calculation)	48
7.5.1	Training-time dropout	48
7.5.2	Test-time scaling	49
7.6	Multi-sample Vectorized Processing	49
8	Optimization Techniques	51
8.1	Stochastic Gradient Descent (SGD)	51
8.1.1	Full-batch gradient descent	51
8.1.2	Mini-batch SGD	51
8.1.3	Learning-rate schedules (common choices)	52
8.1.4	A basic descent inequality (smooth case)	52
8.2	Momentum	52
8.2.1	Heavy-ball momentum	52
8.2.2	Nesterov accelerated gradient (NAG)	52
8.3	Adaptive Methods (RMSProp, Adam, AdamW)	53
8.3.1	RMSProp (core idea)	53
8.3.2	Adam (Adaptive Moment Estimation)	53
8.3.3	AdamW (decoupled weight decay)	53

8.4	Regularization as Optimization	53
8.4.1	L2 regularization (weight decay) and MAP interpretation	54
8.4.2	L1 regularization and sparsity	54
8.4.3	Dropout (inverted dropout)	54
8.4.4	Batch normalization (BN)	55
8.5	Stability Tricks (practical)	55
8.5.1	Gradient clipping	55
8.5.2	Mini-batch size trade-off	55
9	Analysis and Theory	57
9.1	Function Approximation	57
9.1.1	Setting and notation	57
9.1.2	Universal Approximation Theorem (UAT)	57
9.1.3	Approximation rates (why UAT is not enough)	58
9.1.4	Why depth helps (compositional structure)	58
9.2	Depth vs. Width	58
9.2.1	Expressivity measures	58
9.2.2	Piecewise linear regions (ReLU intuition)	58
9.2.3	Separation results (functions needing depth)	59
9.3	Optimization Landscapes	59
9.3.1	Nonconvexity and critical points	59
9.3.2	Saddle points in high dimension (intuition)	59
9.3.3	Overparameterization and benign landscapes (idea)	59
9.4	Generalization	60
9.4.1	Train vs. test	60
9.4.2	Bias–variance decomposition (squared loss)	60
9.4.3	Capacity control and uniform convergence (sketch)	60
9.4.4	Implicit regularization (phenomenon)	60
9.5	Interpolation and Double Descent	61
9.5.1	Classical U-shaped curve	61
9.5.2	Interpolation threshold	61
9.5.3	Double descent (empirical phenomenon)	61
9.6	What theory does <i>not</i> yet explain	61
10	Computational Graphs and Automatic Differentiation	63
10.1	Computational Graphs: Formal Definition	63
10.1.1	Directed acyclic graphs (DAGs)	63
10.1.2	Example: Simple expression graph	63
10.1.3	Forward evaluation	64
10.2	Automatic Differentiation: Backpropagation in DAGs	64
10.2.1	Generalized chain rule	64
10.2.2	Example: Computing adjoints for $y = (x_1 + x_2) \cdot x_1$	64
10.3	Forward Mode vs. Reverse Mode Differentiation	65
10.3.1	Reverse mode (backpropagation)	65
10.3.2	Forward mode (tangent linear)	65
10.3.3	Comparison table	66
10.4	Chain Rule in Multivariate Form	66
10.4.1	Jacobian-vector products	66

10.4.2 Example: Softmax backward	66
11 Numerical Stability and Precision	67
11.1 Softmax and the Log-Sum-Exp Trick	67
11.1.1 Naive softmax (numerically unstable)	67
11.1.2 Stable variant (log-sum-exp)	67
11.1.3 Log-domain computation	67
11.2 Underflow and Overflow in Deep Networks	68
11.2.1 Activation norms	68
11.2.2 Initialization and gradient norms	68
11.2.3 Gradient clipping	68
11.3 Mixed Precision Training	68
11.3.1 Strategy	68
11.3.2 Why scaling helps	68
12 Tensor Operations and Notation	69
12.1 Tensors and Index Notation	69
12.1.1 Definition	69
12.1.2 Einstein notation (summation convention)	69
12.2 Broadcasting and Element-wise Operations	70
12.2.1 Broadcasting rules (NumPy/PyTorch convention)	70
12.3 Reshape and Transpose	70
12.3.1 Reshape (view)	70
12.3.2 Transpose (permutation)	70
12.3.3 Flattening (vectorization)	71
13 Hyperparameter Tuning and Learning Rate Schedules	73
13.1 Learning Rate Selection	73
13.1.1 Learning rate finder (LRFinder)	73
13.1.2 Learning rate schedules	73
13.2 Warmup	74
13.2.1 Linear warmup	74
13.2.2 Gradient accumulation + warmup	74
13.3 Hyperparameter Search Methods	74
13.3.1 Grid search	74
13.3.2 Random search	75
13.3.3 Bayesian optimization	75
14 Data Preprocessing and Normalization	77
14.1 Input Normalization	77
14.1.1 Standardization (Z-score)	77
14.1.2 Min-Max scaling	77
14.1.3 Data statistics (train vs. test)	77
14.2 Batch Normalization (Revisited)	78
14.2.1 Running mean and variance (inference)	78
14.3 Layer Normalization	78
14.4 Group Normalization and Instance Normalization	78
14.4.1 Group normalization	78
14.4.2 Instance normalization	78

14.5 Comparison of Normalization Methods	79
15 Recurrent Neural Networks (RNNs)	81
15.1 Sequence Data and Mathematical Formulation	81
15.1.1 Temporal Data Representation	81
15.1.2 Notation and Convention	81
15.1.3 Common Task Architectures	81
15.2 Vanilla RNN Definition and Forward Propagation	82
15.2.1 Recurrent Computation	82
15.2.2 Parameter Sharing Across Time	82
15.2.3 Vectorized Mini-batch Forward Pass	82
15.3 Computational Graph Unrolling in Time	83
15.3.1 Unrolled Graph Representation	83
15.3.2 Temporal Dependencies	83
15.4 Backpropagation Through Time (BPTT)	84
15.4.1 Loss Function and Objective	84
15.4.2 Backpropagation Through Time Algorithm	84
15.4.3 Parameter Gradients	84
15.5 Vanishing and Exploding Gradients: Mathematical Analysis	85
15.5.1 Gradient Flow Through Hidden States	85
15.5.2 Spectral Analysis	85
15.5.3 Mathematical Condition for Stability	86
15.6 Common Questions (RNNs)	86
15.6.1 Q1: Why do we share \mathbf{W}_{hh} ?	86
15.6.2 Q2: What exactly is "vanishing gradient"?	86
15.6.3 Q3: What is the computational cost of BPTT?	87
15.6.4 Q4: Why can't RNNs be parallelized?	87
15.7 Common Questions (Gradient Problems)	88
15.7.1 Q5: What is the danger of exploding gradients?	88
15.7.2 Q6: Why is the spectral radius important?	88
16 Long Short-Term Memory (LSTM)	89
16.1 Motivation and Design Principles	89
16.1.1 The Problem with Vanilla RNNs	89
16.1.2 The LSTM Solution: Additive State Update	89
16.2 Complete LSTM Cell Definition	90
16.2.1 Gate Computations	90
16.2.2 State and Hidden State Updates	90
16.3 Common Questions (LSTM)	91
16.3.1 Q7: Why doesn't the cell state gradient vanish?	91
16.3.2 Q8: Are 4 gates really necessary?	91
16.3.3 Q9: Why initialize the Forget gate to 1?	91
16.3.4 Q10: Difference between Cell State and Hidden State?	91
17 Sequence-to-Sequence Models and Attention	93
17.1 Encoder-Decoder Architecture	93
17.1.1 Motivation	93
17.1.2 Fixed-Length Context Vector	93
17.2 Attention Mechanism	93

17.2.1	The Bottleneck Problem	93
17.2.2	Attention Weights	93
17.3	Common Questions (Seq2Seq & Attention)	94
17.3.1	Q11: Limitations of Fixed Context Vectors?	94
17.3.2	Q12: What do Attention weights mean?	94
17.3.3	Q13: Which Attention score is best?	94
17.3.4	Q14: Why Multi-Head Attention?	94
18	Transformer Architecture	95
18.1	Self-Attention Mechanism	95
18.1.1	Query, Key, Value Projection	95
18.1.2	Scaled Dot-Product Attention	95
18.1.3	Why Scale by $\sqrt{d_k}$?	96
18.2	Multi-Head Attention	96
18.2.1	Multiple Attention Heads	96
18.3	Positional Encoding	96
18.3.1	Sinusoidal Positional Encoding	96
18.4	Transformer Encoder Block	97
18.4.1	Block Structure	97
18.4.2	Feed-Forward Network	97
18.4.3	Layer Normalization	97
18.5	Transformer Decoder Block	97
18.5.1	Three Sub-layers	97
18.5.2	Masked Attention	97
18.5.3	Masked Attention	97
18.6	Common Questions (Transformer)	98
18.6.1	Q15: Why does Self-Attention solve the RNN problem?	98
18.6.2	Q16: What is the difference between Query, Key, and Value?	98
18.6.3	Q17: Why scale by $\sqrt{d_k}$?	99
18.6.4	Q18: Sinusoidal vs. Learnable Positional Encoding?	99
18.6.5	Q19: Layer Norm vs. Batch Norm?	99
18.6.6	Q20: What is Masked Attention?	99
18.6.7	Q21: Is Transformer really faster than RNN?	99
19	Scaling Laws and Foundation Models	101
19.1	Scaling Laws	101
19.1.1	Empirical Observations	101
19.2	In-Context Learning	101
19.3	Common Questions (Foundation Models)	101
19.3.1	Q22: Do Scaling Laws hold forever?	101
19.3.2	Q23: Why do Emergent Abilities occur?	102
19.3.3	Q24: How does In-Context Learning work?	102
19.3.4	Q25: What makes a "Foundation" Model?	102
19.4	Summary Table: Architecture Comparison	102

20 Transformer Variants and Modern Architectures	103
20.1 Overview of Transformer Evolution	103
20.1.1 Timeline and Motivation	103
20.2 Encoder-Only: BERT and Variants	103
20.2.1 BERT Architecture	103
20.2.2 Masked Language Modeling (MLM)	103
20.3 Decoder-Only: GPT and Variants	104
20.3.1 GPT Architecture	104
20.3.2 Causal Language Modeling	104
20.3.3 Instruction Tuning and RLHF	104
20.4 Encoder-Decoder: T5 and Variants	104
20.4.1 T5: Unified Framework	104
20.5 Sparse and Efficient Variants	104
20.5.1 The $O(T^2)$ Problem	104
20.5.2 Longformer & BigBird	105
20.5.3 Mixture of Experts (MoE)	105
20.6 Multimodal and Vision Transformers	105
20.6.1 Vision Transformer (ViT)	105
20.6.2 CLIP	105
20.7 Recent Trends	105
20.7.1 RAG (Retrieval-Augmented Generation)	105
20.7.2 LoRA (Low-Rank Adaptation)	105
20.8 Common Questions (Transformer Variants)	105
20.8.1 Q26: Why is BERT bidirectional but GPT unidirectional?	105
20.8.2 Q27: Does Masked LM leak data?	106
20.8.3 Q28: Why do models suddenly get smarter with scale?	106
20.8.4 Q29: What does Temperature do?	106
20.8.5 Q30: Why is the KL penalty needed in RLHF?	106
20.8.6 Q31: How can T5 unify all tasks?	106
20.8.7 Q32: Does Sparse Attention lose information?	107
20.8.8 Q33: Is Linear Transformer exactly equivalent?	107
20.8.9 Q34: How to decide the number of Experts (MoE)?	107
20.8.10 Q35: Why is ViT better than CNNs?	107
20.8.11 Q36: RAG vs. Fine-tuning?	107
20.8.12 Q37: Prefix Tuning vs. LoRA?	107
21 Complete Backpropagation Walkthroughs: RNN to Transformer	109
21.1 Part I: Simple RNN - Complete Backpropagation Example	109
21.1.1 Setup: 2-Layer RNN with Concrete Numbers	109
21.1.2 Forward Propagation	110
21.1.3 Loss Calculation	110
21.1.4 Backward Propagation Through Time (BPTT)	110
21.2 Part II: LSTM - Complete Backpropagation Example	111
21.2.1 Setup	111
21.2.2 Forward Pass (Abstract)	111
21.2.3 Backward Pass Logic	111
21.3 Part III: Transformer - Complete Backpropagation Example	112
21.3.1 Setup: Minimal Attention	112

21.3.2 Forward: Multi-Head Attention	112
21.3.3 Backward: Gradients	112
21.4 Summary of Backprop Complexity	112

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:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix} \quad (1.1)$$

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

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

More generally, the L^p norm is:

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

For $p = 1$ (Manhattan distance):

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

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

$$\|\mathbf{v}\|_\infty = \max_i |v_i| \quad (1.5)$$

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:

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

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

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos \theta \quad (1.7)$$

This implies:

$$\cos \theta = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \|\mathbf{b}\|} \quad (1.8)$$

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:

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

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

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

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 $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{m \times p}$ where:

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

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:

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

Properties of transpose:

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

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

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

1.2 Calculus: Foundations of Optimization

1.2.1 Derivatives and the Chain Rule

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

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

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:

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx} \quad (1.16)$$

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

$$\frac{dy}{dx} = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{dx} \quad (1.17)$$

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

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

1.2.2 Partial Derivatives and Gradients

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

$$\frac{\partial f}{\partial x_i} = \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_i + h, \dots, x_n) - f(x_1, \dots, x_i, \dots, x_n)}{h} \quad (1.19)$$

The gradient is the vector of all partial derivatives:

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

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

$$\nabla_{\mathbf{d}} f = \mathbf{d}^T \nabla f = \nabla f \cdot \mathbf{d} \quad (1.21)$$

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

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

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

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

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

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

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:

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

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 $\mathcal{L}(\mathbf{x}) = \frac{1}{2} \|\mathbf{y} - \mathbf{W}\mathbf{x}\|_2^2$:

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

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):

$$X = [\mathbf{x}^{(1)} \ \mathbf{x}^{(2)} \ \cdots \ \mathbf{x}^{(m)}] \in \mathbb{R}^{d \times m}.$$

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

Network architecture and parameters

Consider an L -layer feedforward network with layer widths

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

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

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

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

Forward pass (single example):

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

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

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

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

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):

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

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)

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

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

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

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:

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

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: $\text{ReLU}(z) = \max(0, z)$ with subgradient $\text{ReLU}'(z) = 1$ 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*

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

The perceptron prediction is

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

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

2.1.1 Hyperplane interpretation

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

$$\mathcal{H} = \{\mathbf{x} \in \mathbb{R}^d : \mathbf{w}^\top \mathbf{x} + b = 0\}. \quad (2.3)$$

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:

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

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

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:

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

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 $\mathbf{w} \neq 0$. The **signed distance** from a point \mathbf{x} to the hyperplane is

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

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:

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

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

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

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

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

Then apply the perceptron update:

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

$$b_{t+1} = b_t + \eta y_i, \quad (2.12)$$

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}_*, b_*) such that

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

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

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

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

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

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

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) :

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

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

$$\mathbf{w}_M^\top \mathbf{w}_* \geq M \gamma \|\mathbf{w}_*\|_2. \quad (2.17)$$

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

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

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

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

By induction,

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

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

$$\mathbf{w}_M^\top \mathbf{w}_\star \leq \|\mathbf{w}_M\|_2 \|\mathbf{w}_\star\|_2. \quad (2.21)$$

Plugging (2.17) and (2.20):

$$M \gamma \|\mathbf{w}_\star\|_2 \leq \|\mathbf{w}_M\|_2 \|\mathbf{w}_\star\|_2 \leq R\sqrt{M} \|\mathbf{w}_\star\|_2. \quad (2.22)$$

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

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

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

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

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

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

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

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

Then (3.2) becomes the compact vector form:

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

Applying an activation function element-wise,

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

3.1.4 Mini-batch (fully vectorized) form

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

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

Then the affine map becomes

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

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

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

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 $\ell = 1, \dots, L$.

The network output is

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

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

Writing out the full composition:

$$f(\mathbf{x}; \theta) = \sigma^{(L)} \left(\mathbf{W}^{(L)} \sigma^{(L-1)} \left(\mathbf{W}^{(L-1)} \dots \sigma^{(1)} \left(\mathbf{W}^{(1)} \mathbf{x} + \mathbf{b}^{(1)} \right) \dots + \mathbf{b}^{(L-1)} \right) + \mathbf{b}^{(L)} \right), \quad (3.10)$$

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

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

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

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

By induction, the entire network becomes

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

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

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

Derivative:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z)). \quad (3.15)$$

3.3.3 Hyperbolic tangent

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

Derivative:

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

3.3.4 ReLU

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

Subgradient (used in practice):

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

3.3.5 Softmax

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

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

Its Jacobian is

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

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

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

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:

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

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

3.4 A Fully Worked Tiny Example (Optional)

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

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

Hidden pre-activations:

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

hidden activations:

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

Output layer:

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

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

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

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

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

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

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:

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

4.2.2 Gradient w.r.t. the prediction

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

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

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 $P(Y = 1 | \mathbf{x})$. 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

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

Over a dataset:

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

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

Differentiate (4.5):

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

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:

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

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

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

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

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

Over a dataset:

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

4.4.2 Softmax Jacobian

The derivative of softmax has the form (Jacobian):

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

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

Equivalently, in matrix form:

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

4.4.3 Softmax + CCE simplification (logit gradient)

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

$$\frac{\partial \ell_{\text{CCE}}}{\partial p_i} = -\frac{y_i}{p_i}. \quad (4.14)$$

Then apply chain rule:

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

Cancel p_i :

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

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

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

Therefore,

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

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 $\delta > 0$:

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

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

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

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

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

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

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

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

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

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

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

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

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,

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

where \odot denotes element-wise multiplication.

5.2.3 Hidden layer delta

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

$$\boldsymbol{\delta}^{(\ell)} = (\mathbf{W}^{(\ell+1)})^\top \boldsymbol{\delta}^{(\ell+1)} \odot \sigma^{(\ell)'}(\mathbf{z}^{(\ell)}). \quad (5.7)$$

Interpretation:

- $(\mathbf{W}^{(\ell+1)})^\top \boldsymbol{\delta}^{(\ell+1)}$ maps the error signal back to layer ℓ .
- Multiplying by $\sigma^{(\ell)'}(\cdot)$ 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

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

Hence,

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

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

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

In matrix form:

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

Similarly, for the bias:

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

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 $\mathbf{a}^{(\ell-1),(i)}$:

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

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

(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:

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

The vectorized forward pass is

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

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

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

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 $\ell = L$,

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

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

$$\Delta^{(\ell)} = (W^{(\ell+1)})^\top \Delta^{(\ell+1)} \odot \sigma^{(\ell)'}(Z^{(\ell)}). \quad (5.19)$$

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

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

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

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

Consistency check (shapes).

$$(W^{(\ell+1)})^\top \Delta^{(\ell+1)} \in \mathbb{R}^{n_\ell \times m}, \quad \Delta^{(\ell)} (A^{(\ell-1)})^\top \in \mathbb{R}^{n_\ell \times n_{\ell-1}}, \quad (5.22)$$

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

$$\Delta^{(L)} = \hat{Y} - Y. \quad (5.23)$$

- **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

$$\Delta^{(L)} = P - Y. \quad (5.24)$$

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

$$\hat{y} = \sigma(z) = \frac{1}{1 + e^{-z}}, \quad \sigma'(z) = \hat{y}(1 - \hat{y}). \quad (5.25)$$

Binary cross-entropy loss:

$$\mathcal{L}(\hat{y}, y) = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]. \quad (5.26)$$

First,

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

Then by chain rule:

$$\delta^{(L)} = \frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} = \frac{\hat{y} - y}{\hat{y}(1 - \hat{y})} \cdot \hat{y}(1 - \hat{y}) = \hat{y} - y. \quad (5.28)$$

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

$$p_k = \text{softmax}(\mathbf{z})_k = \frac{e^{z_k}}{\sum_{j=1}^K e^{z_j}}. \quad (5.29)$$

With one-hot label vector $\mathbf{y} \in \{0, 1\}^K$, categorical cross-entropy:

$$\mathcal{L}(\mathbf{p}, \mathbf{y}) = -\sum_{k=1}^K y_k \log p_k. \quad (5.30)$$

A key result (derivable from the Jacobian of softmax) is:

$$\boldsymbol{\delta}^{(L)} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \mathbf{p} - \mathbf{y}. \quad (5.31)$$

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:

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

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: $\text{ReLU}'(z) = 1$ 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 $\ell = 1, \dots, L$ 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 $\ell = L - 1, \dots, 1$ 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 $y \in \{0, 1\}$:

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

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

We use a $2 \rightarrow 3 \rightarrow 1$ 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:

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

Layer 2 parameters:

$$\mathbf{W}^{(2)} = [0.7 \ 0.8 \ 0.9] \in \mathbb{R}^{1 \times 3}, \quad b^{(2)} = 0.1. \quad (6.4)$$

6.3 Forward Propagation: General Form

For each sample \mathbf{x} :

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

$$\mathbf{a}^{(1)} = \text{ReLU}(\mathbf{z}^{(1)}), \quad (6.6)$$

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

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

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

$$\mathcal{L}^{(i)} = -\left(y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})\right). \quad (6.9)$$

Batch loss:

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

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

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

Compute entrywise:

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

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

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

Thus

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

6.4.2 Layer 1 activation (ReLU)

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

6.4.3 Layer 2 pre-activation

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

$$= 0.7 \cdot 0.10 + 0.8 \cdot 0.25 + 0.9 \cdot 0.40 + 0.1 \quad (6.18)$$

$$= 0.07 + 0.20 + 0.36 + 0.10 = 0.73. \quad (6.19)$$

6.4.4 Output (sigmoid)

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

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

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

6.4.5 Loss for sample 1

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

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

Numerically,

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

6.5 Forward Propagation: Sample 2 (Detailed)

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

6.5.1 Layer 1

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

Entrywise:

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

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

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

Since all entries are positive,

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

6.5.2 Layer 2 and output

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

$$= 0.182 + 0.488 + 0.864 + 0.1 = 1.634, \quad (6.30)$$

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

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

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

6.6 Batch Loss

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

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

Then the batch loss is

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

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

$$= \frac{2.4652}{4} \approx 0.6163. \quad (6.36)$$

6.7 Backpropagation: General Form

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

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

Then

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

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

For the hidden layer (ReLU):

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

and

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

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

Here

$$\text{ReLU}'(z) = \begin{cases} 1 & (z > 0), \\ 0 & (z \leq 0). \end{cases} \quad (6.43)$$

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),

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

6.8.2 Gradients for layer 2

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

$$= 0.6748 \cdot [0.10, 0.25, 0.40] \quad (6.46)$$

$$= [0.06748, 0.16870, 0.26992], \quad (6.47)$$

and

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

6.8.3 Hidden delta

Using (6.40):

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

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

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

6.8.4 Gradients for layer 1

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

$$= \begin{bmatrix} 0.47236 \\ 0.53984 \\ 0.60732 \end{bmatrix} [0.5 \quad 0.2] \quad (6.52)$$

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

and

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

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:

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

In the current draft, the averaged gradient is summarized as

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

6.10 Parameter Updates (SGD)

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

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

6.10.1 Update layer 2

$$\mathbf{W}_{\text{new}}^{(2)} = \mathbf{W}^{(2)} - 0.1 \frac{\partial \mathcal{L}_{\text{batch}}}{\partial \mathbf{W}^{(2)}} \quad (6.58)$$

$$= [0.7, 0.8, 0.9] - 0.1[0.0289, 0.0425, 0.0568] \quad (6.59)$$

$$= [0.6971, 0.7958, 0.8943]. \quad (6.60)$$

The bias update is similarly

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

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

6.10.2 Update layer 1

Likewise,

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

The current draft summarizes the updated parameters numerically as

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

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):

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

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

so the new loss becomes

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

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

$$\theta_{\text{new}} = \theta - \eta g, \quad (7.1)$$

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

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

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

the update magnitude depends linearly on η .

- If $\eta = 0.1$:

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

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

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

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:

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

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

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

is the squared Frobenius norm.

7.2.2 Concrete computation

Using the Chapter 6 weights:

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

Compute squared Frobenius norms:

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

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

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

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

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

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

7.2.3 Gradient effect (weight decay view)

Differentiating,

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

Thus the update becomes

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

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 :

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

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

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.

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

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

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

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

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

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:

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

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

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

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

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 $j = 1$:

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

Mean:

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

Variance:

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

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

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

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

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

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

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

and applies

$$\mathbf{h}_{\text{drop}} = \mathbf{h} \odot \mathbf{m}. \quad (7.18)$$

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

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

Then

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

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

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

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):

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

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

$$\mathbf{X} = [\mathbf{x}^{(1)} \ \mathbf{x}^{(2)} \ \dots \ \mathbf{x}^{(m)}] \in \mathbb{R}^{d \times m}. \quad (7.20)$$

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

$$\mathbf{Z} = \mathbf{WX} + \mathbf{b}\mathbf{1}^\top, \quad (7.19)$$

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

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

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

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

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

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

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 $\mathcal{B}_t \subset \{1, \dots, N\}$ be a mini-batch of size B sampled at iteration t . Define the mini-batch objective

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

and its gradient estimate

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

Then SGD updates

$$\theta_{t+1} = \theta_t - \eta_t g_t. \quad (8.5)$$

Under uniform sampling (with standard independence assumptions),

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

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:

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

Then a standard inequality implies

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

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

$$v_{t+1} = \beta v_t + g_t, \quad (8.9)$$

$$\theta_{t+1} = \theta_t - \eta_t v_{t+1}, \quad (8.10)$$

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

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

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

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:

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

$$\theta_{t+1} = \theta_t - \eta_t v_{t+1}. \quad (8.13)$$

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:

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

and update

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

where all operations are element-wise.

8.3.2 Adam (Adaptive Moment Estimation)

Adam maintains first and second moment estimates

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1)g_t, \quad (8.16)$$

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2)(g_t \odot g_t), \quad (8.17)$$

with bias corrections

$$\hat{m}_{t+1} = \frac{m_{t+1}}{1 - \beta_1^{t+1}}, \quad (8.18)$$

$$\hat{v}_{t+1} = \frac{v_{t+1}}{1 - \beta_2^{t+1}}. \quad (8.19)$$

The Adam update is

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

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:

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

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:

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

Then

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

With SGD, the update becomes

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

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:

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

The subgradient satisfies

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

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

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

In inverted dropout, training-time activations are

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

Then

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

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

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

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

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

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

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:

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

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 $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ be an activation function. A one-hidden-layer (two-layer) network with width N is

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

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

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

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

$$\inf_{f_N \in \mathcal{F}_N} \|f_N - f\|? \quad (9.3)$$

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:

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

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

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

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

A critical point satisfies

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

Second-order behavior is characterized by the Hessian

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

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 :

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

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

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

and empirical risk (training loss)

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

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 :

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

(An additional irreducible noise term appears when $Y = f^*(X) + \varepsilon$ 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} :

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

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:

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

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:

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

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, \dots, 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

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

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}):

$$\frac{\partial \mathcal{L}}{\partial v_i} = \sum_{\substack{\text{paths } i \rightarrow \text{root} \\ \text{edges in path}}} \prod \frac{\partial v_{\text{dest}}}{\partial v_{\text{src}}}. \quad (10.2)$$

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

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

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. $v_4 \rightarrow v_5$: edge with $\frac{\partial v_5}{\partial v_4} = 1$, so

$$\bar{v}_4 = \bar{v}_5 \cdot 1 = 20. \quad (10.4)$$

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

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

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

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

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

$$\bar{v}_2 = \bar{v}_3 \cdot 1 = 40. \quad (10.7)$$

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,

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

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

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

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

$$\overline{v_k^{\text{in}}} = J_k^\top \overline{v_k^{\text{out}}}. \quad (10.10)$$

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

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

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

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

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:

$$z'_i = z_i - \max_k z_k, \quad (11.1)$$

then compute

$$p_i = \frac{e^{z'_i}}{\sum_j e^{z'_j}}. \quad (11.2)$$

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

Mathematically:

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

11.1.3 Log-domain computation

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

$$\log p_i = z'_i - \log \left(\sum_j e^{z'_j} \right) = z'_i - \text{logsumexp}(z'). \quad (11.4)$$

This is especially useful when computing cross-entropy:

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

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 $|z^{(\ell)}| \rightarrow 0$ (underflow), gradients vanish. If $|z^{(\ell)}| \rightarrow \infty$ (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:

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

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:

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

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}):

$$\hat{\mathcal{L}} = L_{\text{scale}} \cdot \mathcal{L}. \quad (11.8)$$

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 $T \in \mathbb{R}^{d_1 \times \dots \times d_n}$ is a multi-dimensional array.

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

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

12.1.2 Einstein notation (summation convention)

In Einstein notation, repeated indices imply summation:

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

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

Example: Matrix-vector product

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

Example: Convolution (1D, single sample)

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

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

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

Example: Batched matrix multiplication

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

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

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:

$$Y_{b,m,n} = X_{b,m,n} \cdot \gamma_n \quad (12.5)$$

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:

$$X \in \mathbb{R}^{B \times C \times H \times W} \rightarrow X' \in \mathbb{R}^{BC \times HW}. \quad (12.6)$$

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

12.3.2 Transpose (permutation)

Reorder dimensions:

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

In Einstein notation:

$$Y_{ijkl} = X_{kilj}. \quad (12.8)$$

12.3.3 Flattening (vectorization)

Combining batch and spatial dimensions:

$$X \in \mathbb{R}^{B \times C \times H \times W} \rightarrow \vec{X} \in \mathbb{R}^{B \cdot C \cdot H \cdot W}. \quad (12.9)$$

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 η_{\min} (e.g., 10^{-5}).
2. Train for one epoch, exponentially increasing η at each batch:

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

where T is total batches, η_{\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:

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

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

Exponential decay:

$$\eta_t = \eta_0 \cdot e^{-\lambda t}, \quad (13.3)$$

where $\lambda > 0$ is decay rate.

Cosine annealing:

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

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

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

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

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:

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

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:

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

13.3 Hyperparameter Search Methods

13.3.1 Grid search

Enumerate all combinations of discrete values:

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

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:

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

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:

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

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:

$$x'_i = \frac{x_i - \mu_i}{\sigma_i}, \quad (14.1)$$

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]):

$$x'_i = \frac{x_i - \min_j x_{ij}}{\max_j x_{ij} - \min_j x_{ij}}. \quad (14.2)$$

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:

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

14.2.1 Running mean and variance (inference)

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

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

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):

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

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

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:

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

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:

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

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: $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(T)}$ where each $\mathbf{x}^{(t)} \in \mathbb{R}^{d_x}$
- Target sequence: $\mathbf{y}^{(1)}, \mathbf{y}^{(2)}, \dots, \mathbf{y}^{(T)}$ (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:

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

15.1.3 Common Task Architectures

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

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

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

- Input: single \mathbf{x}

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

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

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

15.2 Vanilla RNN Definition and Forward Propagation

15.2.1 Recurrent Computation

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

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

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

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

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

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:

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

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

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

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

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):

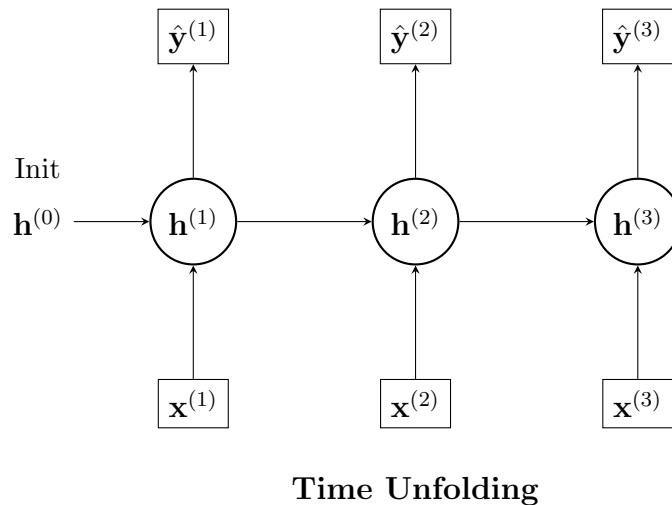


Figure 15.1: Unrolled Recurrent Neural Network. The hidden state $\mathbf{h}^{(t)}$ passes information to the next timestep. (Adapted from Goodfellow et al., 2016)

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:

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

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:

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

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:

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

Derivation:

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

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

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

From the output layer:

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

From the next timestep (via the recurrent connection):

$$\frac{\partial L^{(t+1:T)}}{\partial \mathbf{h}^{(t)}} = \mathbf{W}_{hh}^\top \delta_h^{(t+1)} \quad (15.15)$$

Therefore:

$$\delta_h^{(t)} = \left(\mathbf{W}_{hy}^\top \delta_y^{(t)} + \mathbf{W}_{hh}^\top \delta_h^{(t+1)} \right) \odot \sigma'_h(\mathbf{z}_h^{(t)}) \quad (15.16)$$

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

15.4.3 Parameter Gradients

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

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

$$= \sum_{t=1}^T \delta_h^{(t)} (\mathbf{h}^{(t-1)})^\top \quad (15.18)$$

Similarly:

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

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

For biases:

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

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

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:

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

Derivation:

By the chain rule:

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

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

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

$$= \text{diag}(\sigma'_h(\mathbf{z}_h^{(j)})) \mathbf{W}_{hh} \quad (15.26)$$

15.5.2 Spectral Analysis

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

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

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 $\|\mathbf{W}_{hh}\|^{t-k} \rightarrow 0$ as $t - k \rightarrow \infty$.

- Consequence: Gradients for distant timesteps become negligible.
- Learning long-term dependencies becomes slow.
- **Exploding gradients:** If $\rho > 1$, then $\|\mathbf{W}_{hh}\|^{t-k} \rightarrow \infty$ as $t - k \rightarrow \infty$.
 - 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**:

$$\kappa_T = \rho^T \quad (15.28)$$

For long sequences (T large):

- If $\rho < 1$: exponential decay of $\kappa_T \rightarrow 0$ (vanishing)
- If $\rho > 1$: exponential growth of $\kappa_T \rightarrow \infty$ (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:

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

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 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.

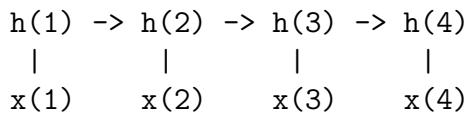
Since data must be kept for gradient computation across all steps, it is memory-heavy.

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:



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:

$$\frac{\partial L}{\partial \mathbf{h}^{(1)}} \propto \prod_{t=2}^T \mathbf{W}_{hh}^\top \text{diag}(\sigma'_h) \quad (16.1)$$

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:

$$0.9^{100} \approx 2.7 \times 10^{-5} \quad (16.2)$$

The gradient vanishes severely.

16.1.2 The LSTM Solution: Additive State Update

Instead of a multiplicative update $\mathbf{h}^{(t)} = \sigma_h(\mathbf{W}_{hh}\mathbf{h}^{(t-1)} + \dots)$, LSTM uses an **additive state update**:

$$\mathbf{c}^{(t)} = \mathbf{c}^{(t-1)} + (\text{something new}) \quad (16.3)$$

where $\mathbf{c}^{(t)}$ is the **cell state** (internal memory).

Key insight: The gradient flowing through the cell state is:

$$\frac{\partial \mathbf{c}^{(t)}}{\partial \mathbf{c}^{(t-1)}} = \mathbf{f}^{(t)} \quad (16.4)$$

where $\mathbf{f}^{(t)}$ is the **forget gate**. If $\mathbf{f}^{(t)} \approx 1$, then:

$$\prod_{j=k}^t \frac{\partial \mathbf{c}^{(j)}}{\partial \mathbf{c}^{(j-1)}} = \prod_{j=k}^t \mathbf{f}^{(j)} \approx 1 \quad (16.5)$$

The product remains stable (close to 1), preventing vanishing gradients!

16.2 Complete LSTM Cell Definition

16.2.1 Gate Computations

Forget Gate:

$$\mathbf{f}^{(t)} = \sigma(\mathbf{W}_{xf}\mathbf{x}^{(t)} + \mathbf{W}_{hf}\mathbf{h}^{(t-1)} + \mathbf{b}_f) \quad (16.6)$$

where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid (output in $[0, 1]$).

Input Gate:

$$\mathbf{i}^{(t)} = \sigma(\mathbf{W}_{xi}\mathbf{x}^{(t)} + \mathbf{W}_{hi}\mathbf{h}^{(t-1)} + \mathbf{b}_i) \quad (16.7)$$

Output Gate:

$$\mathbf{o}^{(t)} = \sigma(\mathbf{W}_{xo}\mathbf{x}^{(t)} + \mathbf{W}_{ho}\mathbf{h}^{(t-1)} + \mathbf{b}_o) \quad (16.8)$$

Cell State Candidate (Tanh pre-activation):

$$\tilde{\mathbf{c}}^{(t)} = \tanh(\mathbf{W}_{xc}\mathbf{x}^{(t)} + \mathbf{W}_{hc}\mathbf{h}^{(t-1)} + \mathbf{b}_c) \quad (16.9)$$

16.2.2 State and Hidden State Updates

Cell State Update (additive, the crucial part):

$$\mathbf{c}^{(t)} = \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} + \mathbf{i}^{(t)} \odot \tilde{\mathbf{c}}^{(t)} \quad (16.10)$$

where \odot denotes element-wise multiplication.

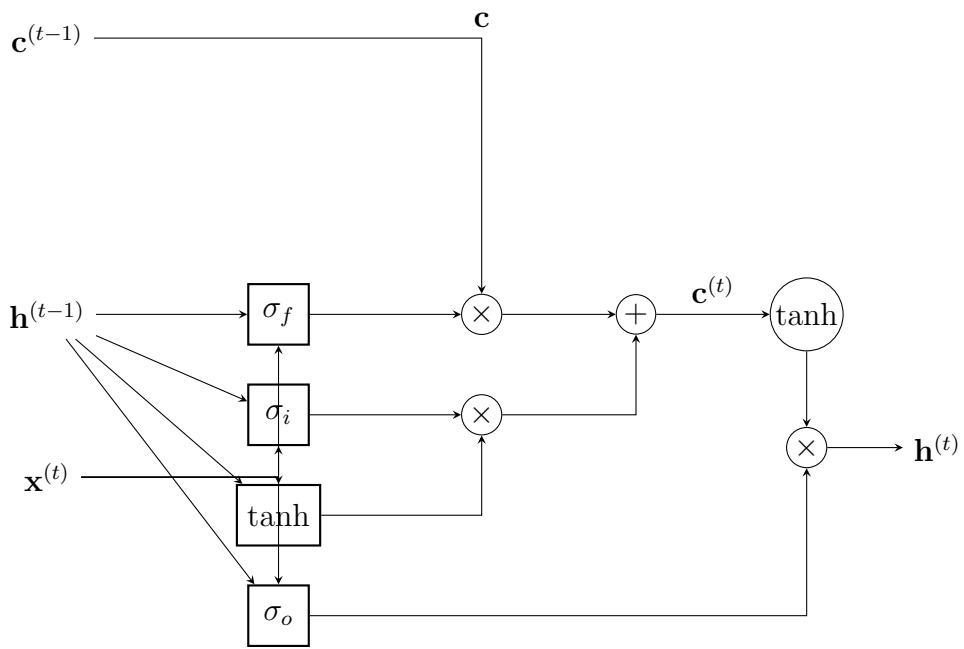


Figure 16.1: Structure of the LSTM Cell. The cell state \mathbf{c} runs along the top "highway," interacting linearly via the forget and input gates. (Adapted from Olah, 2015)

Hidden State Output:

$$\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \odot \tanh(\mathbf{c}^{(t)}) \quad (16.11)$$

16.3 Common Questions (LSTM)

16.3.1 Q7: Why doesn't the cell state gradient vanish?

A: Due to the additive connection ($\mathbf{c}^{(t)} = \mathbf{c}^{(t-1)} + \dots$), 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.

16.3.2 Q8: 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.

16.3.3 Q9: Why initialize the Forget gate to 1?

A: To ensure the model **preserves past information by default** at the start of training.

Bias initialization:

$$b_f = 1.0 \implies \sigma(1.0) \approx 0.73 \quad (16.12)$$

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.

16.3.4 Q10: 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:

$$\mathbf{c} = \mathbf{h}_{\text{enc}}^{(T)} \quad (17.1)$$

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 ($s = 1 \dots T$):

$$\alpha_{t,s} = \frac{\exp(\text{score}(\mathbf{s}_t, \mathbf{h}_s))}{\sum_{k=1}^T \exp(\text{score}(\mathbf{s}_t, \mathbf{h}_k))} \quad (17.2)$$

The context vector becomes dynamic:

$$\mathbf{c}_t = \sum_{s=1}^T \alpha_{t,s} \mathbf{h}_s \quad (17.3)$$

17.3 Common Questions (Seq2Seq & Attention)

17.3.1 Q11: 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 Q12: 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 Q13: Which Attention score is best?

A: Scaled Dot-Product is the standard winner.

- **Dot:** $\mathbf{s}^T \mathbf{h}$ (Fast).
- **Additive:** $\mathbf{v}^T \tanh(\dots)$ (Flexible).
- **Scaled Dot:** $\frac{\mathbf{s}^T \mathbf{h}}{\sqrt{d}}$ (Fast + Stable).

17.3.4 Q14: 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:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q, \quad \mathbf{Q} \in \mathbb{R}^{T \times d_k} \quad (18.1)$$

$$\mathbf{K} = \mathbf{X}\mathbf{W}_K, \quad \mathbf{K} \in \mathbb{R}^{T \times d_k} \quad (18.2)$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}_V, \quad \mathbf{V} \in \mathbb{R}^{T \times d_v} \quad (18.3)$$

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):

$$\mathbf{E} = \mathbf{Q}\mathbf{K}^\top, \quad \mathbf{E} \in \mathbb{R}^{T \times T} \quad (18.4)$$

Scale by $\sqrt{d_k}$:

$$\mathbf{E}_{\text{scaled}} = \frac{\mathbf{E}}{\sqrt{d_k}} \quad (18.5)$$

Softmax normalization:

$$\mathbf{A} = \text{softmax}(\mathbf{E}_{\text{scaled}}) = \text{softmax} \left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}} \right) \quad (18.6)$$

Attention output:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \mathbf{AV} \quad (18.7)$$

18.1.3 Why Scale by $\sqrt{d_k}$?

For random vectors $\mathbf{q}, \mathbf{k} \sim \mathcal{N}(0, 1)^{d_k}$:

$$\text{Var}[\mathbf{q}^\top \mathbf{k}] = d_k \quad (18.8)$$

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 :

$$\text{head}_i = \text{Attention}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i) = \text{softmax}\left(\frac{\mathbf{Q}_i \mathbf{K}_i^\top}{\sqrt{d_k}}\right) \mathbf{V}_i \quad (18.9)$$

Concatenate heads:

$$\text{MultiHead}(\mathbf{X}) = \text{Concat}(\text{head}_1, \dots, \text{head}_h) \mathbf{W}_O \quad (18.10)$$

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 :

$$PE(\text{pos}, 2i) = \sin\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right) \quad (18.11)$$

$$PE(\text{pos}, 2i + 1) = \cos\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right) \quad (18.12)$$

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

$$\mathbf{Z}_{\text{ffn}} = \max(0, \mathbf{Y}_{\text{attn}} \mathbf{W}_1 + \mathbf{b}_1) \mathbf{W}_2 + \mathbf{b}_2 \quad (18.13)$$

where typically $d_{\text{ffn}} = 4 \times d_{\text{model}}$.

18.4.3 Layer Normalization

$$\text{LayerNorm}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \mathbb{E}[\mathbf{x}]}{\sqrt{\text{Var}[\mathbf{x}] + \epsilon}} + \beta \quad (18.14)$$

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:

$$\mathbf{E}_{\text{masked}} = \mathbf{E} + \mathbf{M} \quad (18.15)$$

where:

$$M_{i,j} = \begin{cases} 0 & \text{if } j \leq i \\ -\infty & \text{if } j > i \end{cases} \quad (18.16)$$

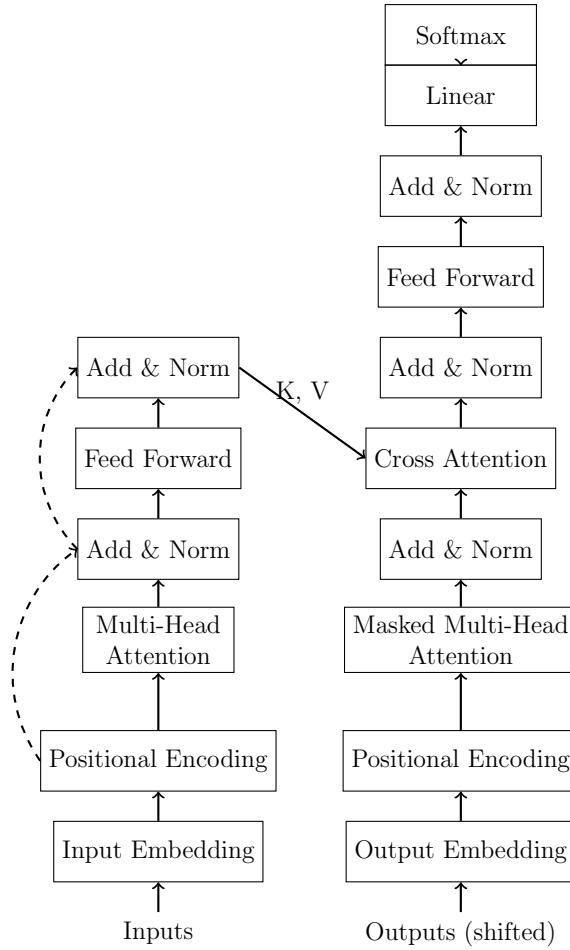


Figure 18.1: The Transformer Architecture. Left: Encoder block. Right: Decoder block. (Based on Vaswani et al., 2017)

18.6 Common Questions (Transformer)

18.6.1 Q15: 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 $1 \rightarrow 2 \rightarrow \dots \rightarrow 100$. 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).

18.6.2 Q16: 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 → Retrieve Value weighted by match strength.

18.6.3 Q17: 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.

18.6.4 Q18: 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.

18.6.5 Q19: 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.

18.6.6 Q20: 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.

18.6.7 Q21: Is Transformer really faster than RNN?

A:

- **Training:** Yes (Much faster). Fully parallelizable across time.
- **Inference:** No (Slower). Auto-regressive generation requires attending to all past tokens ($O(T^2)$), whereas RNN is $O(T)$. Systems use KV-caching to mitigate this.

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):

$$L(N) \propto N^{-\alpha_N}, \quad L(P) \propto P^{-\alpha_P}, \quad L(C) \propto C^{-\alpha_C} \quad (19.1)$$

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 Q22: 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 Q23: 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 Q24: 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" → "Output". Mechanistically, Induction Heads (special attention heads) copy patterns from previous context.

19.3.4 Q25: 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 → 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.

$$L_{\text{MLM}} = - \sum_{i \in \text{masked}} \log P(\text{token}_i | \text{context}) \quad (20.1)$$

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

$$L_{\text{CLM}} = - \sum_{t=1}^T \log P(\text{token}_{t+1} | \text{token}_{1:t}) \quad (20.2)$$

BERT (Bidirectional) GPT (Unidirectional)

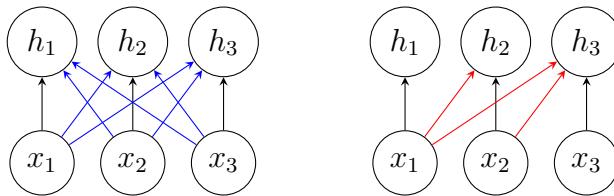


Figure 20.1: Comparison of Attention Patterns. Left: BERT allows attending to all tokens (bidirectional). Right: GPT only allows attending to past tokens (unidirectional causal). (Adapted from Radford et al., 2018; Devlin et al., 2019)

20.3.3 Instruction Tuning and RLHF

Modern LLMs (like ChatGPT) use Reinforcement Learning from Human Feedback (RLHF):

$$\mathcal{L}_{\text{RL}} = \mathbb{E}[\text{KL}(\pi_\theta \| \pi_{\text{ref}}) - \lambda R(\text{response})] \quad (20.3)$$

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: ..." → Target
- **Classification:** "Classify sentiment: ..." → "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:

$$\text{sim}(I, T) = E_I(I) \cdot E_T(T) \quad (20.4)$$

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:

$$\mathbf{W}_{\text{new}} = \mathbf{W}_0 + \mathbf{A}\mathbf{B}^T \quad (20.5)$$

where \mathbf{A}, \mathbf{B} are low-rank matrices.

20.8 Common Questions (Transformer Variants)

20.8.1 Q26: 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 Q27: 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 Q28: 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 Q29: 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 Q30: 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 Q31: How can T5 unify all tasks?

A: Because fundamentally, almost all NLP tasks classify as "Seq2Seq".

- Classification is Sequence → Label (Short sequence).
- Generation is Sequence → Sequence.

By using prompts, T5 learns a universal mapping function.

20.8.7 Q32: 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 Q33: 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 Q34: 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 Q35: 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 Q36: 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 Q37: 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

Complete Backpropagation Walkthroughs: RNN to Transformer

21.1 Part I: Simple RNN - Complete Backpropagation Example

21.1.1 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:

$$\mathbf{W}_{hh} = \begin{pmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.1 & 0.2 \\ 0.2 & 0.3 & 0.1 \end{pmatrix}, \quad \mathbf{W}_{xh} = \begin{pmatrix} 0.5 & 0.6 \\ 0.3 & 0.4 \\ 0.2 & 0.1 \end{pmatrix} \quad (21.1)$$

$$\mathbf{W}_{hy} = \begin{pmatrix} 0.1 \\ 0.7 & 0.5 & 0.3 \end{pmatrix}, \quad \mathbf{b}_h = \begin{pmatrix} 0.0 \\ 0.1 \end{pmatrix}, \quad b_y = 0.05 \quad (21.2)$$

Activation functions: Hidden: $\sigma_h(z) = \tanh(z)$, Output: $\sigma_y(z) = \sigma(z) = \frac{1}{1+e^{-z}}$.

Input sequence:

$$\mathbf{x}^{(1)} = \begin{pmatrix} 0.5 \\ 0.2 \end{pmatrix}, \quad \mathbf{x}^{(2)} = \begin{pmatrix} 0.3 \\ 0.4 \end{pmatrix}, \quad \mathbf{x}^{(3)} = \begin{pmatrix} 0.2 \\ 0.5 \end{pmatrix} \quad (21.3)$$

Target: $y = 1$.

21.1.2 Forward Propagation

Timestep 1

$$\mathbf{z}_h^{(1)} = \mathbf{W}_{hh} \mathbf{h}^{(0)} + \mathbf{W}_{xh} \mathbf{x}^{(1)} + \mathbf{b}_h \quad (21.4)$$

$$= \mathbf{0} + \begin{pmatrix} 0.47 \\ 0.23 \\ 0.22 \end{pmatrix} \quad (\text{after calc}) \quad (21.5)$$

$$\mathbf{h}^{(1)} = \tanh(\mathbf{z}_h^{(1)}) \approx \begin{pmatrix} 0.440 \\ 0.226 \\ 0.216 \end{pmatrix} \quad (21.6)$$

$$\hat{y}^{(1)} = \sigma(\mathbf{W}_{hy} \mathbf{h}^{(1)} + b_y) = \sigma(0.5358) \approx 0.631 \quad (21.7)$$

Timestep 2

$$\mathbf{z}_h^{(2)} = \begin{pmatrix} 0.645 \\ 0.452 \\ 0.357 \end{pmatrix}, \quad \mathbf{h}^{(2)} \approx \begin{pmatrix} 0.567 \\ 0.422 \\ 0.343 \end{pmatrix}, \quad \hat{y}^{(2)} \approx 0.682 \quad (21.8)$$

Timestep 3

$$\mathbf{z}_h^{(3)} \approx \begin{pmatrix} 0.812 \\ 0.641 \\ 0.485 \end{pmatrix}, \quad \mathbf{h}^{(3)} \approx \begin{pmatrix} 0.675 \\ 0.568 \\ 0.449 \end{pmatrix}, \quad \hat{y}^{(3)} \approx 0.720 \quad (21.9)$$

21.1.3 Loss Calculation

Binary Cross-Entropy ($y = 1$):

$$L = -\log(0.720) \approx 0.329 \quad (21.10)$$

21.1.4 Backward Propagation Through Time (BPTT)

Output Delta (Timestep 3)

$$\delta_y^{(3)} = \hat{y}^{(3)} - y = 0.720 - 1 = -0.280 \quad (21.11)$$

Hidden Delta (Timestep 3)

$$\delta_h^{(3)} = (\mathbf{W}_{hy}^T \delta_y^{(3)}) \odot \sigma'_h(\mathbf{z}_h^{(3)}) \quad (21.12)$$

$$= \begin{pmatrix} -0.196 \\ -0.140 \\ -0.084 \end{pmatrix} \odot \begin{pmatrix} 0.544 \\ 0.677 \\ 0.798 \end{pmatrix} = \begin{pmatrix} -0.107 \\ -0.095 \\ -0.067 \end{pmatrix} \quad (21.13)$$

Hidden Delta (Timestep 2)

Gradient only from recurrent connection (assuming output loss only at $T = 3$ for simplification):

$$\delta_h^{(2)} = (\mathbf{W}_{hh}^T \delta_h^{(3)}) \odot \sigma'_h(\mathbf{z}_h^{(2)}) \quad (21.14)$$

$$= \begin{pmatrix} -0.0621 \\ -0.051 \\ -0.0578 \end{pmatrix} \odot \begin{pmatrix} 0.679 \\ 0.822 \\ 0.882 \end{pmatrix} = \begin{pmatrix} -0.0422 \\ -0.0419 \\ -0.0510 \end{pmatrix} \quad (21.15)$$

Hidden Delta (Timestep 1)

$$\delta_h^{(1)} \approx \begin{pmatrix} -0.0252 \\ -0.0265 \\ -0.0157 \end{pmatrix} \quad (21.16)$$

Parameter Gradients (Summed)

$$\frac{\partial L}{\partial \mathbf{W}_{hh}} \approx \begin{pmatrix} -0.0792 & -0.0547 & -0.0458 \\ -0.0723 & -0.0496 & -0.0416 \\ -0.0604 & -0.0398 & -0.0340 \end{pmatrix} \quad (21.17)$$

21.2 Part II: LSTM - Complete Backpropagation Example

21.2.1 Setup

Input $\mathbf{x} \in \mathbb{R}^2$, Hidden $\mathbf{h} \in \mathbb{R}^2$. 2 Timesteps. Simplified weights: $\mathbf{W} \approx 0.1\mathbf{I}$.

21.2.2 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: $\mathbf{h}^{(t)} = \mathbf{o}^{(t)} \odot \tanh(\mathbf{c}^{(t)})$

21.2.3 Backward Pass Logic

The crucial difference from RNN is the gradient flow through \mathbf{c} .

$$\frac{\partial L}{\partial \mathbf{c}^{(t)}} = \frac{\partial L}{\partial \mathbf{h}^{(t)}} \odot \mathbf{o}^{(t)} \odot (1 - \tanh^2 \mathbf{c}^{(t)}) + \frac{\partial L}{\partial \mathbf{c}^{(t+1)}} \odot \mathbf{f}^{(t+1)} \quad (21.18)$$

The term $\odot \mathbf{f}^{(t+1)}$ allows gradients to persist without decay if $\mathbf{f} \approx 1$.

21.3 Part III: Transformer - Complete Backpropagation Example

21.3.1 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 $T \times d$).

21.3.2 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:** $\text{head}_1 = \mathbf{A}_1 \mathbf{V}_1^T$.

21.3.3 Backward: Gradients

Gradient through Value:

$$\frac{\partial L}{\partial \mathbf{V}_1} = \frac{\partial L}{\partial \text{head}_1} \mathbf{A}_1 \quad (21.19)$$

Gradient through Attention Matrix:

$$\frac{\partial L}{\partial \mathbf{A}_1} = \left(\frac{\partial L}{\partial \text{head}_1} \right)^T \mathbf{V}_1 \quad (21.20)$$

Gradient through Softmax (Jacobian): For each column j (token scores):

$$\frac{\partial A_{ij}}{\partial E_{kj}} = A_{ij} (\delta_{ik} - A_{kj}) \quad (21.21)$$

This connects $\partial L / \partial \mathbf{A}$ to $\partial L / \partial \mathbf{E}$.

Gradient through \mathbf{Q}, \mathbf{K} :

$$\frac{\partial L}{\partial \mathbf{Q}_1} \propto \frac{\partial L}{\partial \mathbf{E}_1} \mathbf{K}_1 \quad (21.22)$$

21.4 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	$O(T^2d)$	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