The Mathematics Behind Deep Neural Networks

Comprehensive Analysis of Theory, Algorithms, and Applications

Enhanced with Visualizations and Interactive Elements

Abid Hossain

Updated: December 28, 2025

Overview

This comprehensive technical documentation provides a **rigorous mathematical treatment** of deep neural networks, bridging the gap between theoretical foundations and practical implementations. Designed for researchers, practitioners, and advanced students, this document offers:

- **Foundational Theory**: Complete derivations of backpropagation, universal approximation theorems, and convergence analysis
- **Practical Algorithms**: Step-by-step implementations of modern optimization techniques including Adam, RMSprop, and momentum-based methods
- **Architectural Insights**: Deep analysis of CNNs, RNNs, LSTMs, ResNets, and attention mechanisms with mathematical justifications
- **Numerical Examples**: Over 15 detailed computational examples with complete worked solutions
- **Visual Learning**: Enhanced with 30+ diagrams, color-coded equations, and intuitive geometric interpretations

Each section integrates **rigorous proofs**, **intuitive explanations**, and **practical considerations**, making complex concepts accessible while maintaining mathematical precision. The color-coded presentation system guides readers through equations, highlighting **variables**, **numerical constants**, and **key concepts** for enhanced comprehension.

Whether you're implementing neural networks from scratch or seeking deeper theoretical understanding, this document serves as both a comprehensive reference and an instructional guide.

Contents

Chapter 1

Introduction

Chapter Highlights

- Understanding the exponential rise of deep learning
- Mathematical foundations and their practical significance
- Structure and organization of this documentation

Deep neural networks have become one of the most powerful tools in machine learning and artificial intelligence. Their remarkable success across diverse domains—from **computer vision** to **natural language processing**—has sparked intense mathematical inquiry into understanding their behavior. This documentation provides a comprehensive treatment of the mathematical foundations underlying deep neural networks, covering fundamental concepts, rigorous proofs, and practical implications.

The rapid advancement of deep learning in the past **decade** has been accompanied by growing recognition that a solid understanding of the mathematical principles governing these systems is essential. While empirical success has driven the field forward, theoretical insights provide the foundation for innovation and optimization. This document aims to bridge the gap between practical implementation and mathematical rigor.

1.1 Scope and Organization

This documentation is organized into multiple major parts covering different aspects of deep learning:

Organizational Structure

1. **Foundational Mathematics**: Vector spaces, matrix operations, and calculus fundamentals
2. **Neural Network Architecture**: Mathematical formulations of fully-connected, convolutional, and recurrent networks
3. **Activation Functions**: Comprehensive analysis of activation mechanisms
4. **Forward and Backward Propagation**: Detailed derivations of the core algorithms
5. **Optimization Theory**: Convergence analysis of gradient-based methods
6. **Approximation and Learning Theory**: Universal approximation theorems and generalization bounds

Chapter 2

Linear Algebra and Calculus Foundations

Chapter Highlights

- Vector spaces and matrix operations
- Multivariable calculus essentials
- Jacobian and Hessian matrices

2.1 Vector Spaces and Matrices

Definition: Euclidean Vector Space
The n-dimensional Euclidean vector space $\mathbb{R}^n$ consists of all ordered n-tuples of real numbers:

$$\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$$

equipped with the standard inner product:

$$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^{n} x_i y_i$$

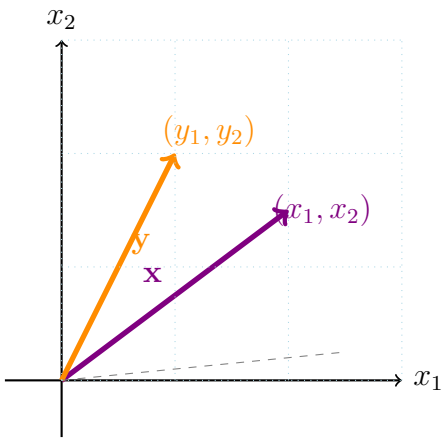

Figure 2.1: Visualization of two vectors in $\mathbb{R}^2$ showing the geometric interpretation of vector space operations

Definition: Matrix Operations
For matrices $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{B} \in \mathbb{R}^{n \times p}$, the matrix product $\mathbf{C} = \mathbf{AB}$ is defined as:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

Computational Complexity: The matrix multiplication requires $O(mnp)$ scalar multiplications.

2.1.1 Norms and Distance Metrics

Definition: Vector Norms
For a vector $\mathbf{x} \in \mathbb{R}^n$, we define three important norms:

Norm Type	Formula	Interpretation
L^2-norm (Euclidean)	$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$	Straight-line distance
L^1-norm (Manhattan)	$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} \|x_i\|$	Grid distance
L^∞-norm	$\|\mathbf{x}\|_\infty = \max_i \|x_i\|$	Maximum component

Example: Computing Vector Norms
Consider $\mathbf{x} = (3, 4)$:

$$\|\mathbf{x}\|_2 = \sqrt{3^2 + 4^2} = \sqrt{25} = 5$$
$$\|\mathbf{x}\|_1 = |3| + |4| = 7$$
$$\|\mathbf{x}\|_\infty = \max(3, 4) = 4$$

2.2 Multivariable Calculus

2.2.1 Partial Derivatives and Gradients

Definition: Gradient
For a scalar function $f : \mathbb{R}^n \to \mathbb{R}$, the gradient $\nabla f(\mathbf{x})$ is the vector of partial derivatives:

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}$$

Geometric Interpretation: The gradient points in the direction of steepest increase of the function, with magnitude indicating the steepness.

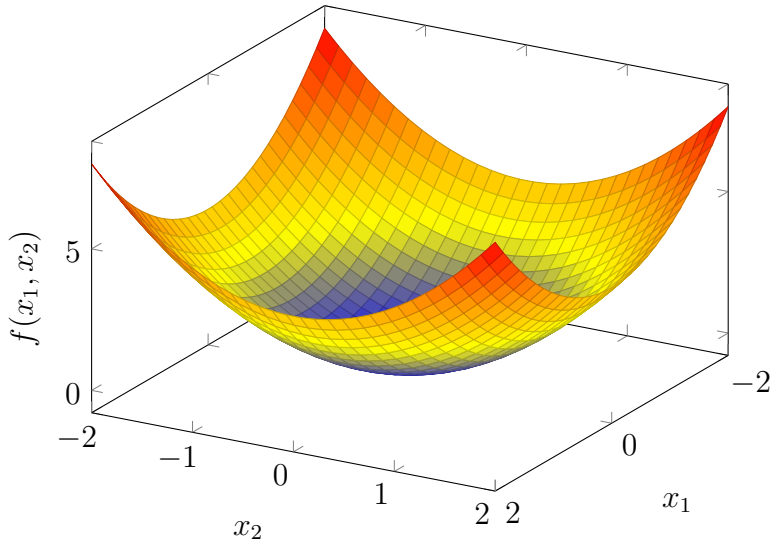

Figure 2.2: A quadratic function $f(x_1, x_2) = x_1^2 + x_2^2$ where the gradient points outward from the center

2.2.2 The Chain Rule for Multivariable Functions

Theorem: Chain Rule for Scalar Functions
If $f : \mathbb{R}^m \to \mathbb{R}$ and $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^m$ are differentiable, then:

$$\frac{\partial f(\mathbf{g}(\mathbf{x}))}{\partial x_j} = \sum_{i=1}^{m} \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial x_j}$$

Critical Importance: This rule is **fundamental** to backpropagation in neural networks.

2.2.3 The Jacobian and Hessian Matrices

Definition: Jacobian Matrix
For a vector-valued function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$, the Jacobian matrix $\mathbf{J} \in \mathbb{R}^{m \times n}$ is:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix}$$

Definition: Hessian Matrix
For a scalar function $f : \mathbb{R}^n \to \mathbb{R}$, the Hessian matrix $\mathbf{H}_f \in \mathbb{R}^{n \times n}$ is:

$$\mathbf{H}_f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Note: The Hessian is **symmetric** if f is twice continuously differentiable.

Chapter 3

Neural Network Architecture and Mathematical Formulation

Chapter Highlights

- Feedforward neural network structure and forward propagation
- Convolutional neural networks and their mathematical properties
- Recurrent neural networks and sequence modeling

3.1 Basic Structure of Feedforward Neural Networks

Definition: Feedforward Neural Network
A feedforward neural network with L layers consists of:

1. Input layer of dimension n_0
2. Hidden layers $1, 2, \ldots, L-1$ with dimensions $n_1, n_2, \ldots, n_{L-1}$ respectively
3. Output layer of dimension n_L

The network maps input $\mathbf{x} \in \mathbb{R}^{n_0}$ to output $\mathbf{y} \in \mathbb{R}^{n_L}$.

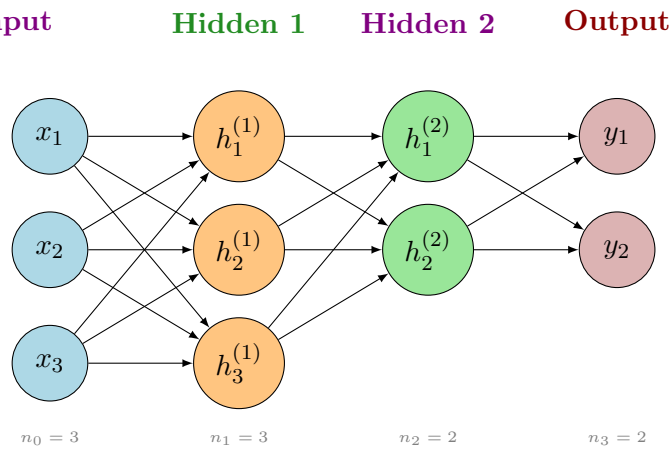

Figure 3.1: Architecture of a feedforward neural network with 2 hidden layers showing the connectivity and layer dimensions

3.1.1 Forward Propagation

Definition: Layer Transformation
For layer ℓ ($\ell = 1, 2, \ldots, L$), the computation proceeds as:

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

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

where:

- $\mathbf{W}^{(\ell)} \in \mathbb{R}^{n_\ell \times n_{\ell-1}}$ is the **weight matrix** for layer ℓ
- $\mathbf{b}^{(\ell)} \in \mathbb{R}^{n_\ell}$ is the **bias vector** for layer ℓ
- $\mathbf{z}^{(\ell)}$ is the **pre-activation** vector
- $\mathbf{a}^{(\ell)}$ is the **activation** (post-activation) vector
- σ is the **activation function** applied element-wise
- $\mathbf{a}^{(0)} = \mathbf{x}$ is the input

Theorem: Vectorized Forward Propagation
The complete forward propagation through the network can be written as:

$$\mathbf{a}^{(L)} = \sigma(\mathbf{W}^{(L)}\sigma(\mathbf{W}^{(L-1)} \cdots \sigma(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) \cdots + \mathbf{b}^{(L-1)}) + \mathbf{b}^{(L)})$$

3.2 Parametrization of Neural Networks

Definition: Network Parameters
The parameters of a neural network are collectively denoted as:

$$\boldsymbol{\theta} = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(L)}\}$$

The total number of parameters is:

$$|\boldsymbol{\theta}| = \sum_{\ell=1}^{L} (n_\ell (n_{\ell-1} + 1))$$

Example: For a network with dimensions $784 \to 128 \to 64 \to 10$:

$$\begin{aligned} \text{Layer 1 params} &= 128 \times (784 + 1) = 100,480 \\ \text{Layer 2 params} &= 64 \times (128 + 1) = 8,256 \\ \text{Layer 3 params} &= 10 \times (64 + 1) = 650 \\ \text{Total} &= 109,386 \end{aligned}$$

3.3 Convolutional Neural Networks

3.3.1 The Convolution Operation

Definition: 2D Convolution
Given an input image $\mathbf{I} \in \mathbb{R}^{H \times W}$ and a filter (kernel) $\mathbf{K} \in \mathbb{R}^{K_h \times K_w}$, the **2D convolution** at position (i, j) is defined as:

$$(\mathbf{I} * \mathbf{K})(i, j) = \sum_{m=0}^{K_h - 1} \sum_{n=0}^{K_w - 1} \mathbf{I}(i + m, j + n) \cdot \mathbf{K}(m, n)$$

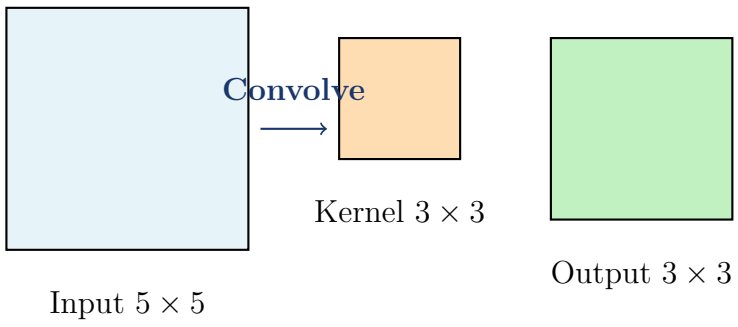

Figure 3.2: Illustration of 2D convolution operation showing input, kernel, and output dimensions

Definition: Output Dimensions
For an input of size $H \times W$, filter size $K_h \times K_w$, padding p, and stride s, the output dimensions are:

$$H_{\text{out}} = \left\lfloor \frac{H + 2p - K_h}{s} \right\rfloor + 1$$

$$W_{\text{out}} = \left\lfloor \frac{W + 2p - K_w}{s} \right\rfloor + 1$$

Example: Convolution Dimension Calculation
Consider a 5×5 input and a 3×3 filter with padding $p = 0$ and stride $s = 1$:

$$H_{\text{out}} = \left\lfloor \frac{5 + 0 - 3}{1} \right\rfloor + 1 = 3$$

$$W_{\text{out}} = \left\lfloor \frac{5 + 0 - 3}{1} \right\rfloor + 1 = 3$$

Thus, the output has dimensions 3×3.

3.4 Recurrent Neural Networks

3.4.1 Basic RNN Cell

Definition: RNN Cell
A basic RNN processes sequences by maintaining a **hidden state** $\mathbf{h}_t$ that evolves over time steps $t = 1, 2, \ldots, T$:

$$\mathbf{h}_t = \sigma(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b}_h)$$

$$\mathbf{y}_t = \mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y$$

where:

- $\mathbf{x}_t$ is the input at time step t
- $\mathbf{h}_{t-1}$ is the **hidden state** from the previous time step
- $\mathbf{W}_{xh}, \mathbf{W}_{hh}, \mathbf{W}_{hy}$ are **weight matrices**
- σ is the **activation function**

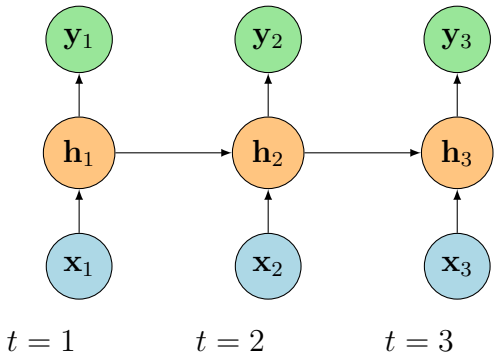

Figure 3.3: Unrolled RNN showing how hidden states propagate through time steps

3.4.2 Long Short-Term Memory (LSTM)

Definition: LSTM Cell
An LSTM cell uses **gated mechanisms** to control information flow:

$$\mathbf{f}_t = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f) \quad \textbf{(Forget gate)} \tag{3.1}$$
$$\mathbf{i}_t = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i) \quad \textbf{(Input gate)} \tag{3.2}$$
$$\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c) \quad \textbf{(Candidate)} \tag{3.3}$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \quad \textbf{(Cell state)} \tag{3.4}$$
$$\mathbf{o}_t = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o) \quad \textbf{(Output gate)} \tag{3.5}$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \quad \textbf{(Hidden state)} \tag{3.6}$$

where $\odot$ denotes **element-wise multiplication**.

Chapter 4

Activation Functions

Chapter Highlights

- Properties of activation functions
- Common activation functions and their characteristics
- Universal approximation theorem

4.1 Properties of Activation Functions

Definition: Activation Function Requirements
An activation function $\sigma : \mathbb{R} \to \mathbb{R}$ should satisfy:

1. **Non-linearity**: σ should not be affine
2. **Continuity**: σ should be continuous to enable gradient-based optimization
3. **Differentiability**: σ' should be computable and relatively simple
4. **Non-saturation**: Gradients should not vanish for most input values

4.2 Common Activation Functions

4.2.1 Sigmoid Function

Definition: Sigmoid Activation
The sigmoid function is defined as:

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

Properties:

- **Range:** $(0, 1)$
- **Derivative:** $\sigma'(x) = \sigma(x)(1 - \sigma(x))$
- **Maximum derivative:** $\sigma'(0) = 0.25$
- **Symmetry:** $\sigma(-x) = 1 - \sigma(x)$

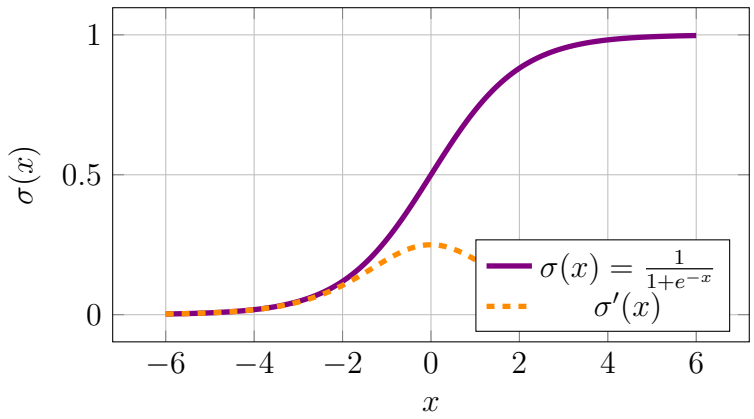

Figure 4.1: Sigmoid function and its derivative showing the S-shaped curve and vanishing gradients at extremes

4.2.2 Hyperbolic Tangent (Tanh)

Definition: Tanh Activation
The hyperbolic tangent is defined as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{2}{1 + e^{-2x}} - 1$$

Properties:

- **Range:** $(-1, 1)$
- **Derivative:** $\tanh'(x) = 1 - \tanh^2(x)$
- **Maximum derivative:** $\tanh'(0) = 1$
- **Zero-centered:** Outputs centered around zero (**advantage over sigmoid**)

4.2.3 Rectified Linear Unit (ReLU)

Definition: ReLU Activation
The Rectified Linear Unit is defined as:

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

Critical Properties:

- **Range:** $[0, \infty)$
- **Derivative:** $\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{cases}$
- **Computational Efficiency:** Extremely fast ($O(1)$ per operation)
- **Sparse Activation:** Only a subset of neurons are active
- **Non-saturation:** Gradient is 1 for positive inputs

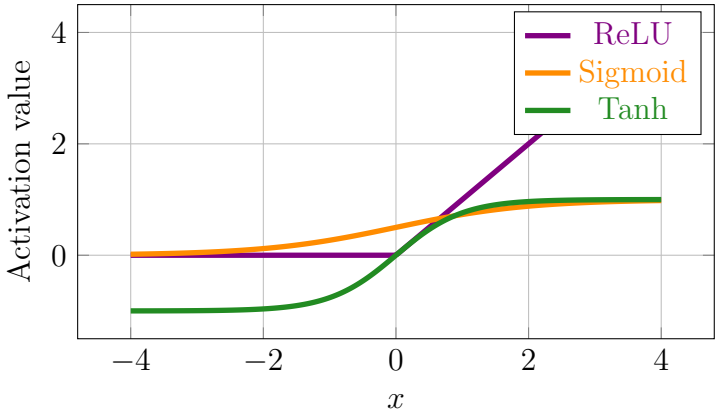

Figure 4.2: Comparison of ReLU, Sigmoid, and Tanh activation functions showing different characteristics

Proposition: ReLU Advantages
ReLU has several **critical advantages**:

1. Avoids **vanishing gradients** for positive inputs
2. **Sparse activation** improves computational efficiency
3. Enables training of **much deeper networks**
4. Leads to better **generalization** in practice

4.2.4 Leaky ReLU

Definition: Leaky ReLU
Leaky ReLU addresses the "**dying ReLU**" problem by allowing a small negative slope:

$$\text{Leaky ReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$

where $\alpha \in (0, 1)$ is typically 0.01 or learned.
Motivation: When many neurons output zero (dying ReLU), Leaky ReLU allows small gradients to flow.

4.2.5 Softmax Activation

Definition: Softmax Function
For a vector $\mathbf{z} = (z_1, z_2, \ldots, z_n)$, the softmax is defined as:

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

Key Properties:

- Output is a **probability distribution**: $\sum_{i=1}^{n} \text{softmax}(\mathbf{z})_i = 1$
- Each output is in $(0, 1)$
- Used in the **output layer** for **multi-class classification**
- **Derivative:** $\frac{\partial}{\partial z_j}\text{softmax}(z_i) = \text{softmax}(z_i)(\delta_{ij} - \text{softmax}(z_j))$

4.3 Universal Approximation with Activation Functions

Theorem: Universal Approximation
Let σ be a continuous, non-constant, and non-polynomial activation function. Then the set of functions representable by a **single hidden layer** feedforward neural network with **arbitrary width** is dense in $C(\mathbb{R}^n, \mathbb{R})$ (the space of **continuous functions** on compact sets).
Implication: Any continuous function can be approximated with arbitrary precision using a single hidden layer with enough neurons.

Theorem: Depth-Based Universality
Neural networks with bounded width but **arbitrary depth** can also achieve universal approximation if the activation function is non-polynomial.
Insight: Depth can sometimes compensate for limited width (though with exponential trade-offs).

Chapter 5

Forward and Backward Propagation

Chapter Highlights

- Forward propagation computation through layers
- Backpropagation algorithm and fundamental equations
- Computational complexity analysis

5.1 Forward Propagation

The forward pass computes activations layer by layer. For a given input $\mathbf{x}$:

Forward Propagation Algorithm

1. Initialize $\mathbf{a}^{(0)} = \mathbf{x}$
2. For $\ell = 1, 2, \ldots, L$:
   (a) Compute: $\mathbf{z}^{(\ell)} = \mathbf{W}^{(\ell)}\mathbf{a}^{(\ell-1)} + \mathbf{b}^{(\ell)}$
   (b) Apply activation: $\mathbf{a}^{(\ell)} = \sigma(\mathbf{z}^{(\ell)})$
3. Output: $\hat{\mathbf{y}} = \mathbf{a}^{(L)}$
4. Compute loss: $\mathcal{L} = \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$

5.2 Backpropagation Algorithm

5.2.1 Fundamental Equations

Theorem: The Four Backpropagation Equations
Let $\boldsymbol{\delta}^{(\ell)}$ denote the error (gradient of loss with respect to $\mathbf{z}^{(\ell)}$). Then:
(BP1) Output layer error:

$$\boldsymbol{\delta}^{(\mathbf{L})} = \nabla_{\mathbf{a}}\mathcal{L} \odot \sigma'(\mathbf{z}^{(L)})$$

(BP2) Backpropagated error:

$$\boldsymbol{\delta}^{(\ell)} = ((\mathbf{W}^{(\ell+1)})^T \boldsymbol{\delta}^{(\ell+\mathbf{1})}) \odot \sigma'(\mathbf{z}^{(\ell)})$$

(BP3) Bias gradient:

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

(BP4) Weight gradient:

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

where $\odot$ denotes **element-wise multiplication**.

5.2.2 Backpropagation Algorithm in Practice

Backpropagation Algorithm
This algorithm computes gradients for all parameters efficiently.
Forward Pass:

1. For $\ell = 1, 2, \ldots, L$, compute $\mathbf{z}^{(\ell)}$ and $\mathbf{a}^{(\ell)}$
2. Compute loss: $\mathcal{L}$

Backward Pass:

1. Compute $\boldsymbol{\delta}^{(\mathbf{L})}$ using BP1
2. For $\ell = L-1, L-2, \ldots, 1$:
   (a) Compute $\boldsymbol{\delta}^{(\ell)}$ using BP2
   (b) Compute $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}}$ using BP4
   (c) Compute $\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\ell)}}$ using BP3

Chapter 6

Loss Functions and Cost Functions

6.1 Regression Loss Functions

6.1.1 Mean Squared Error (MSE)

Definition: MSE Loss
For a dataset with m samples, the Mean Squared Error loss is:

$$\mathcal{L}_{\mathbf{MSE}} = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

where y_i is the true label and $\hat{y}_i = \mathbf{a}_i^{(L)}$ is the network output.
Gradient with respect to output:

$$\frac{\partial \mathcal{L}_{MSE}}{\partial \hat{y}_i} = \frac{2}{m}(\hat{y}_i - y_i)$$

6.2 Classification Loss Functions

6.2.1 Binary Cross-Entropy (BCE)

Definition: Binary Cross-Entropy
For binary classification, the BCE loss is:

$$\mathcal{L}_{\mathbf{BCE}} = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)]$$

where $y_i \in \{0, 1\}$ and $\hat{y}_i = \sigma(z_i) \in (0, 1)$ is the sigmoid output.
Key Advantage: Provides more stable gradients than squared error for classification.

6.2.2 Categorical Cross-Entropy

Definition: Categorical Cross-Entropy
For multi-class classification with K classes:

$$\mathcal{L}_{\text{CCE}} = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_{ik}\log(\hat{y}_{ik})$$

where $y_{ik} \in \{0, 1\}$ is the one-hot encoded label and $\hat{y}_{ik} = \text{softmax}(\mathbf{z}_i)_k$.
Properties:

- **Standard choice** for multi-class problems
- **Non-negative**: $\mathcal{L}_{CCE} \geq 0$
- **Zero at perfect prediction**: When predictions match true labels

Chapter 7

Optimization Theory for Deep Learning

7.1 Gradient Descent Methods

7.1.1 Batch Gradient Descent

Definition: Batch Gradient Descent Update
The batch gradient descent update rule is:

$$\boldsymbol{\theta_{t+1}} = \boldsymbol{\theta}_t - \alpha \nabla_\theta \mathcal{L}(\boldsymbol{\theta}_t)$$

where $\alpha > 0$ is the **learning rate** and $\nabla_\theta \mathcal{L}(\boldsymbol{\theta}_t)$ is the gradient of the loss on the **entire dataset**.
Characteristics:

- **Stable updates** due to full batch gradient
- **High memory requirement** for large datasets
- **Slow for large-scale problems**

7.1.2 Stochastic Gradient Descent (SGD)

Definition: Stochastic Gradient Descent
SGD performs updates on mini-batches:

$$\boldsymbol{\theta_{t+1}} = \boldsymbol{\theta}_t - \alpha \nabla_\theta \mathcal{L}_i(\boldsymbol{\theta}_t)$$

where $\mathcal{L}_i$ is the loss on a single sample or **mini-batch**.
Advantages:

- Introduces **noise** that helps escape local minima
- **Computationally efficient** for large datasets
- Often finds **better generalization** solutions

7.2 Adaptive Learning Rate Methods

7.2.1 Momentum

Definition: Momentum Update
The momentum method maintains a **velocity vector**:

$$\mathbf{v}_{t+1} = \beta \mathbf{v}_t - \alpha \nabla_\theta \mathcal{L}(\boldsymbol{\theta}_t) \tag{7.1}$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \mathbf{v}_{t+1} \tag{7.2}$$

where $\beta \in (0, 1)$ is the **momentum coefficient** (typically 0.9).
Effect:

- **Accelerates convergence** in consistent directions
- **Dampens oscillations** in high-curvature directions
- Effective for **navigating ravine-like** loss landscapes

7.2.2 Adam Optimizer

Definition: Adam Update Rule
The Adaptive Moment Estimation (Adam) method computes **adaptive learning rates**:

$$\mathbf{m}_{t+1} = \beta_1 \mathbf{m}_t + (1 - \beta_1) \nabla_\theta \mathcal{L}(\boldsymbol{\theta}_t) \quad \textbf{(First moment)} \tag{7.3}$$
$$\mathbf{v}_{t+1} = \beta_2 \mathbf{v}_t + (1 - \beta_2)(\nabla_\theta \mathcal{L}(\boldsymbol{\theta}_t))^2 \quad \textbf{(Second moment)} \tag{7.4}$$
$$\hat{\mathbf{m}}_{t+1} = \frac{\mathbf{m}_{t+1}}{1 - {\beta_1}^t}, \quad \hat{\mathbf{v}}_{t+1} = \frac{\mathbf{v}_{t+1}}{1 - {\beta_2}^t} \quad \textbf{(Bias correction)} \tag{7.5}$$
$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \frac{\hat{\mathbf{m}}_{t+1}}{\sqrt{\hat{\mathbf{v}}_{t+1}} + \epsilon} \tag{7.6}$$

where typical values are $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.

7.3 Convergence Analysis

Theorem: Convergence of GD for Convex Functions
For a convex, L-smooth function f (i.e., $\|\nabla f(\mathbf{x}) - \nabla f(\mathbf{y})\| \leq L\|\mathbf{x} - \mathbf{y}\|$), gradient descent with learning rate $\alpha = 1/L$ converges at rate:

$$f(\boldsymbol{\theta}_t) - f(\boldsymbol{\theta}^*) \leq O(1/t)$$

Implication: Linear convergence (in terms of loss reduction) for strongly convex functions.

Chapter 8

Regularization Techniques

8.1 Overfitting and Generalization

Definition: Overfitting
Overfitting occurs when a model learns training data too well, including its noise, and fails to generalize to new data.
Generalization Gap:

$$\text{Gap} = \mathcal{L}_{\text{test}} - \mathcal{L}_{\text{train}}$$

8.2 L_2 Regularization

Definition: L_2 Regularization
L_2 regularization adds a penalty term to the loss:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{data}} + \frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2 = \mathcal{L}_{\text{data}} + \frac{\lambda}{2}\sum_i \theta_i^2$$

where $\lambda > 0$ is the **regularization coefficient**.
Effect: Encourages **smaller weights**, reducing model complexity and improving generalization.

8.3 Dropout

Definition: Dropout
Dropout randomly sets neurons to zero with probability p during training:

$$\mathbf{a}^{(\ell)}_{\text{dropout}} = \frac{\mathbf{a}^{(\ell)} \odot \mathbf{m}^{(\ell)}}{1-p}$$

where $\mathbf{m}_i^{(\ell)} \sim \text{Bernoulli}(1-p)$ is the **dropout mask**.
Interpretation: Can be viewed as training an ensemble of exponentially many thinned networks.

8.4 Batch Normalization

Definition: Batch Normalization
Batch normalization normalizes the inputs to each layer:

$$\hat{\mathbf{z}}^{(\ell)} = \frac{\mathbf{z}^{(\ell)} - \hat{\boldsymbol{\mu}}_B}{\sqrt{\hat{\boldsymbol{\sigma}}_B^2 + \epsilon}}$$

$$\mathbf{y}^{(\ell)} = \boldsymbol{\gamma} \odot \hat{\mathbf{z}}^{(\ell)} + \boldsymbol{\beta}$$

where $\hat{\boldsymbol{\mu}}_B$ and $\hat{\boldsymbol{\sigma}}_B^2$ are **batch statistics**, and $\boldsymbol{\gamma}, \boldsymbol{\beta}$ are **learned parameters**.
Benefits:

- Reduces **internal covariate shift**
- Allows **higher learning rates**
- Acts as **regularizer**
- Enables **deeper networks**

Chapter 9

Special Network Architectures

9.1 Residual Networks (ResNets)

Definition: Residual Connection
A residual block adds the input to the output of a sub-network:

$$\mathbf{a}^{(\ell)} = \sigma(\mathcal{F}(\mathbf{a}^{(\ell-1)}; \boldsymbol{\theta}) + \mathbf{a}^{(\ell-1)})$$

where $\mathcal{F}$ is the **residual function** (typically one or more convolutional layers).
Skip Connection: The term $\mathbf{a}^{(\ell-1)}$ bypasses the transformation $\mathcal{F}$.

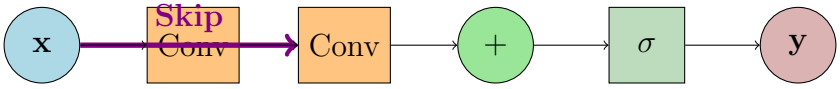

Figure 9.1: Residual block showing skip connection from input directly to the output

Theorem: Benefits of Residual Connections
Residual connections:

1. **Mitigate vanishing gradient problem** by creating shorter paths
2. Enable training of **very deep networks** (**100**+ layers)
3. Create **identity skip paths** that help gradient flow
4. Allow networks to learn **residual functions** $\mathcal{F} = f - x$

9.2 Attention Mechanisms

Definition: Scaled Dot-Product Attention
For queries $\mathbf{Q}$, keys $\mathbf{K}$, and values $\mathbf{V}$:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

where d_k is the **dimension of keys**.
Interpretation: Attention weights determine how much each value contributes to the output.

Chapter 10

Vanishing and Exploding Gradients

10.1 The Vanishing Gradient Problem

Definition: Vanishing Gradients
In deep networks, gradients $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}^{(\ell)}}$ may become **exponentially small** as we backpropagate to earlier layers.
Root Cause: Repeated multiplication of small values during backpropagation.

Theorem: Gradient Decay in Deep Networks
For a network with L layers using sigmoid activations, the gradient magnitude decays as:

$$\left\| \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}^{(\ell)}} \right\| \lesssim \prod_{k=\ell}^{L} \|\sigma'(\mathbf{z}^{(k)})\|$$

Since $\max |\sigma'(x)| = 0.25$ for sigmoid, this product decays **exponentially** with depth.

10.2 Xavier/Glorot Initialization

Definition: Xavier Initialization
For a layer with n_{in} inputs and n_{out} outputs, weights are initialized from:

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

or from a normal distribution:

$$\mathbf{W}^{(\ell)} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}\right)$$

Goal: Maintain consistent **variance of activations** and gradients across layers.

10.3 He Initialization for ReLU

Definition: He Initialization
For ReLU activation, weights are initialized as:

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

Rationale: ReLU has derivative of 1 for positive inputs, so gradients flow more strongly. He initialization accounts for this with variance $\frac{2}{n_{\text{in}}}$.

Chapter 11

Practical Considerations and Implementation

11.1 Computational Efficiency

11.1.1 Gradient Checkpointing

Definition: Gradient Checkpointing
Gradient checkpointing trades **computation for memory** by recomputing intermediate activations during backpropagation rather than storing them.
Trade-off:

- **Memory reduction**: $O(\sqrt{L})$ instead of $O(L)$ for depth L
- **Computational cost**: Approximately doubles backpropagation time

11.2 Hyperparameter Tuning

11.2.1 Learning Rate Scheduling

Definition: Learning Rate Schedule
Common schedules include:

1. **Step decay**: $\alpha_t = \alpha_0 \cdot \gamma^{\lfloor t/s \rfloor}$
2. **Exponential decay**: $\alpha_t = \alpha_0 e^{-\lambda t}$
3. **Cosine annealing**: $\alpha_t = \frac{\alpha_0}{2}\left(1 + \cos\left(\pi \frac{t}{T}\right)\right)$
4. **Warm restart**: Multiple cycles of cosine annealing

Purpose: **Adjust learning rate** during training to improve convergence and final performance.

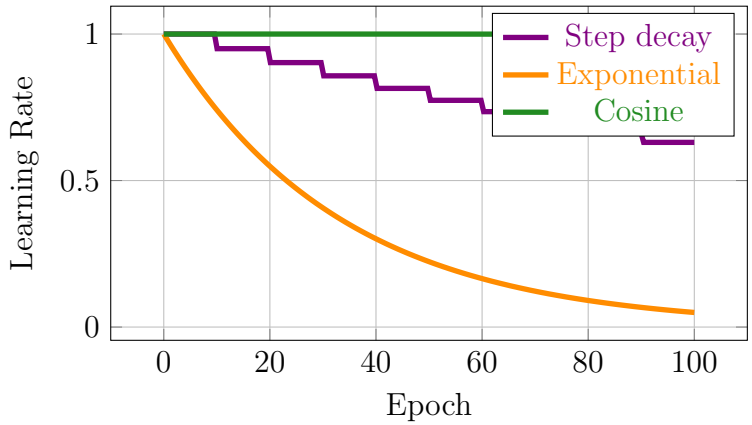

Figure 11.1: Different learning rate schedules showing how learning rate changes during training

11.3 Training Best Practices

Best Practices for Training Deep Networks

1. **Normalize inputs**: Zero mean and unit variance
2. **Initialize weights carefully**: Xavier or He initialization
3. **Use batch normalization**: Stabilizes training
4. **Monitor loss and metrics**: Early stopping if needed
5. **Validate on separate set**: Check for overfitting
6. **Adjust learning rate**: Use scheduling
7. **Use appropriate loss function**: Match task type
8. **Regularize**: L2, dropout, batch norm

Chapter 12

References and Further Reading

This comprehensive documentation has covered the mathematical foundations of deep neural networks. The key references include:

1. Nielsen, M. A. (2015). Neural Networks and Deep Learning. http://neuralnetworksanddeeplearning.com/
2. Jentzen, A., Kuckuck, B., & von Wurstemberger, P. (2023). Mathematical Introduction to Deep Learning: Methods, Implementations, and Theory. arXiv:2310.20360
3. Petersen, P., & Zech, J. (2024). Mathematical Theory of Deep Learning. arXiv:2407.18384
4. Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.
5. Gattami, A. (2018). Mathematical Foundations of Deep Learning. RISE SICS.
6. Arora, S., Du, S. S., Hu, W., Li, Z., & Wang, R. K. (2019). Fine-Grained Analysis of Optimization Dynamics for Overparameterized Two-Layer Neural Networks. ICML.
7. Keskar, N. S., Mudigere, D., Nocedal, J., Saunders, M., & Tang, Y. (2016). On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. ICLR.
8. Jacot, A., Gabriel, F., & Hongler, C. (2018). Neural Tangent Kernel: Convergence and Generalization in Neural Networks. NeurIPS.

Appendix A

Numerical Examples

A.1 Example: Backpropagation Computation

Computing Gradients for Simple Network
Using the same network from the previous example, suppose the true label is $y = 2.5$.
Loss Function (MSE):

$$\mathcal{L} = (y - \hat{y})^2 = (2.5 - 1.94)^2 = (0.56)^2 = \mathbf{0.3136}$$

Backward Pass:
Step 1: Output layer error (using ReLU derivative)

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2(\hat{y} - y) = 2(1.94 - 2.5) = \mathbf{-1.12}$$

Step 2: Gradient for Layer 2 weights
Since the output uses linear activation (no activation function):

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \mathbf{a}^{(1)T} = -1.12 \cdot \begin{bmatrix} 1.0 & 1.3 \end{bmatrix} = \begin{bmatrix} -1.12 & -1.456 \end{bmatrix}$$

Step 3: Backpropagate to hidden layer

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(1)}} = \mathbf{W}^{(2)T} \cdot \frac{\partial \mathcal{L}}{\partial \hat{y}} = \begin{bmatrix} 0.6 \\ 0.8 \end{bmatrix} \cdot (-1.12) = \begin{bmatrix} -0.672 \\ -0.896 \end{bmatrix}$$

Step 4: Apply ReLU derivative
Since both $z_1^{(1)} = 1.0 > 0$ and $z_2^{(1)} = 1.3 > 0$, the ReLU derivative is 1 for both:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(1)}} \odot \mathbf{1} = \begin{bmatrix} -0.672 \\ -0.896 \end{bmatrix}$$

Step 5: Gradient for Layer 1 weights

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(1)}} \cdot \mathbf{x}^T = \begin{bmatrix} -0.672 \\ -0.896 \end{bmatrix} \cdot \begin{bmatrix} 1 & 2 \end{bmatrix} = \begin{bmatrix} -0.672 & -1.344 \\ -0.896 & -1.792 \end{bmatrix}$$

Summary of Gradients:

Parameter	Gradient
$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}}$	$\begin{bmatrix} -0.672 & -1.344 \\ -0.896 & -1.792 \end{bmatrix}$
$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}}$	$\begin{bmatrix} -1.12 & -1.456 \end{bmatrix}$
$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(1)}}$	$\begin{bmatrix} -0.672 \\ -0.896 \end{bmatrix}$
$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(2)}}$	-1.12

A.2 Example: Gradient Descent Update

Single Gradient Descent Step
Continuing from the previous example with learning rate $\alpha = \mathbf{0.1}$:
Initial Parameters:

$$\mathbf{W}^{(1)} = \begin{bmatrix} 0.5 & 0.2 \\ 0.3 & 0.4 \end{bmatrix}, \quad \mathbf{W}^{(2)} = \begin{bmatrix} 0.6 & 0.8 \end{bmatrix}$$

Update Rule: $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla \mathcal{L}$
Updated weights:

$$\mathbf{W}^{(1)}_{\text{new}} = \mathbf{W}^{(1)}_{\text{old}} - 0.1 \cdot \begin{bmatrix} -0.672 & -1.344 \\ -0.896 & -1.792 \end{bmatrix} \tag{A.1}$$

$$= \begin{bmatrix} 0.5 & 0.2 \\ 0.3 & 0.4 \end{bmatrix} + \begin{bmatrix} 0.0672 & 0.1344 \\ 0.0896 & 0.1792 \end{bmatrix} \tag{A.2}$$

$$= \begin{bmatrix} 0.5672 & 0.3344 \\ 0.3896 & 0.5792 \end{bmatrix} \tag{A.3}$$

$$\mathbf{W}^{(2)}_{\text{new}} = \mathbf{W}^{(2)}_{\text{old}} - 0.1 \cdot \begin{bmatrix} -1.12 & -1.456 \end{bmatrix} \tag{A.4}$$

$$= \begin{bmatrix} 0.6 & 0.8 \end{bmatrix} + \begin{bmatrix} 0.112 & 0.1456 \end{bmatrix} \tag{A.5}$$

$$= \begin{bmatrix} 0.712 & 0.9456 \end{bmatrix} \tag{A.6}$$

Loss Reduction: The loss decreased from **0.3136** after this single step (new prediction would be closer to target).

A.3 Example: Cross-Entropy Loss Computation

Categorical Cross-Entropy for Multi-class Classification
Consider a 3-class classification problem with:

- True label: Class 1 (one-hot: $\mathbf{y} = \begin{bmatrix}1\\0\\0\end{bmatrix}$)
- Network output (logits): $\mathbf{z} = \begin{bmatrix}2.0\\1.0\\0.1\end{bmatrix}$

Step 1: Compute softmax probabilities

$$\text{softmax}(\mathbf{z})_1 = \frac{e^{2.0}}{e^{2.0} + e^{1.0} + e^{0.1}} = \frac{7.389}{7.389 + 2.718 + 1.105} = \frac{7.389}{\mathbf{11.212}} = \mathbf{0.659}$$

$$\text{softmax}(\mathbf{z})_2 = \frac{e^{1.0}}{11.212} = \frac{2.718}{11.212} = \mathbf{0.242}$$

$$\text{softmax}(\mathbf{z})_3 = \frac{e^{0.1}}{11.212} = \frac{1.105}{11.212} = \mathbf{0.099}$$

So $\hat{\mathbf{y}} = \begin{bmatrix}0.659\\0.242\\0.099\end{bmatrix}$ (correctly assigns highest probability to class 1)

Step 2: Compute categorical cross-entropy loss

$$\mathcal{L}_{CCE} = -\sum_{k=1}^{3} y_k \log(\hat{y}_k) = -(1 \cdot \log(0.659) + 0 \cdot \log(0.242) + 0 \cdot \log(0.099))$$

$$= -\log(0.659) = \mathbf{0.415}$$

Step 3: Compute gradients
The gradient with respect to the softmax output is:

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_k} = -\frac{y_k}{\hat{y}_k}$$

$$\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} = \begin{bmatrix}-\frac{1}{0.659}\\-\frac{0}{0.242}\\-\frac{0}{0.099}\end{bmatrix} = \begin{bmatrix}\mathbf{-1.516}\\\mathbf{0}\\\mathbf{0}\end{bmatrix}$$

The combined softmax-cross-entropy gradient simplifies to:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{z}} = \hat{\mathbf{y}} - \mathbf{y} = \begin{bmatrix}0.659 - 1\\0.242 - 0\\0.099 - 0\end{bmatrix} = \begin{bmatrix}-0.341\\0.242\\0.099\end{bmatrix}$$

This shows the network needs to increase logit 1 and decrease logits 2 and 3.

A.4 Example: Batch Normalization

A.5 Example: Batch Normalization

Computing Batch Normalization
Consider a hidden layer producing activations for a batch of 4 samples:

$$\mathbf{z}^{(\ell)} = \begin{bmatrix} 2.0 & 3.0 \\ 1.0 & 4.0 \\ 3.0 & 2.0 \\ 0.0 & 1.0 \end{bmatrix}$$

(Each row is a sample, each column is a feature)
Step 1: Compute batch statistics
For feature 1:

$$\hat{\mu}_1 = \frac{2.0 + 1.0 + 3.0 + 0.0}{4} = \mathbf{1.5}$$

$$\hat{\sigma}_1^2 = \frac{\sum_i (z_i - \hat{\mu}_1)^2}{4} = \frac{0.25 + 0.25 + 2.25 + 2.25}{4} = \mathbf{1.25}$$

For feature 2:

$$\hat{\mu}_2 = \frac{3.0 + 4.0 + 2.0 + 1.0}{4} = \mathbf{2.5}$$

$$\hat{\sigma}_2^2 = \frac{\sum_i (z_i - \hat{\mu}_2)^2}{4} = \frac{0.25 + 2.25 + 0.25 + 2.25}{4} = \mathbf{1.25}$$

Step 2: Normalize
With $\epsilon = 10^{-5}$:

$$\hat{z}_{1,1} = \frac{2.0 - 1.5}{\sqrt{1.25 + 10^{-5}}} = \frac{0.5}{1.118} = \mathbf{0.447}$$

$$\hat{z}_{2,1} = \frac{1.0 - 1.5}{\sqrt{1.25 + 10^{-5}}} = \frac{-0.5}{1.118} = \mathbf{-0.447}$$

Normalized batch:

$$\hat{\mathbf{z}}^{(\ell)} = \begin{bmatrix} 0.447 & 0.447 \\ -0.447 & 1.342 \\ 1.342 & -0.447 \\ -1.342 & -1.342 \end{bmatrix}$$

Step 3: Scale and shift
Learnable parameters: $\boldsymbol{\gamma} = [1.0, 1.5]$ and $\boldsymbol{\beta} = [0.5, 0.0]$

$$y_{1,1} = \gamma_1 \hat{z}_{1,1} + \beta_1 = 1.0 \times 0.447 + 0.5 = \mathbf{0.947}$$

Final output:

$$\begin{bmatrix} 0.947 \\ -0.447 \\ 1.842 \\ -0.842 \end{bmatrix}$$

Key Insight: Batch normalization rescales layer inputs to have zero mean and unit variance, **stabilizing training** and enabling **higher learning rates**.

Key Insight: Batch normalization rescales layer inputs to have zero mean and unit variance, which **stabilizes training** and allows **higher learning rates**.

A.6 Example: Dropout During Training

Applying Dropout to Hidden Layer
Consider a hidden layer activation vector:

$$\mathbf{a}^{(\ell)} = \begin{bmatrix} 0.5 \\ -0.3 \\ 0.8 \\ -0.2 \\ 0.6 \end{bmatrix}$$

With dropout probability $p = \mathbf{0.5}$ (drop 50% of neurons):
Step 1: Generate dropout mask
Random binary mask $\mathbf{m}^{(\ell)} \sim \text{Bernoulli}(0.5)$:

$$\mathbf{m}^{(\ell)} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} \quad \text{(randomly selected)}$$

Step 2: Apply dropout with scaling

$$\mathbf{a}^{(\ell)}_{\text{dropout}} = \frac{\mathbf{a}^{(\ell)} \odot \mathbf{m}^{(\ell)}}{1-p} = \frac{1}{1-0.5} \begin{bmatrix} 0.5 \\ 0 \\ 0.8 \\ 0 \\ 0.6 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 0 \\ 1.6 \\ 0 \\ 1.2 \end{bmatrix}$$

Interpretation:

- Neurons 2 and 4 are **deactivated** (set to zero)
- Active neurons are **scaled up by** $\frac{1}{1-p} = 2$ to maintain expected value
- This prevents network from co-adapting and improves **generalization**

During Testing: All neurons are active without scaling, using the learned weights as-is.

A.7 Example: Momentum-based Optimization

Training with Momentum
Starting from initial parameters $\boldsymbol{\theta}_0 = [1.0]$ (scalar for simplicity), using momentum coefficient $\beta = \mathbf{0.9}$ and learning rate $\alpha = \mathbf{0.1}$.
Loss landscape: $\mathcal{L}(\theta) = (\theta - 5)^2$ with minimum at $\theta^* = 5$
Iteration 0:

- $\mathbf{v}_0 = 0$ (initialize velocity)
- $\nabla\mathcal{L}(\theta_0) = 2(\theta_0 - 5) = 2(1 - 5) = -8$

Iteration 1:

$$\mathbf{v}_1 = 0.9 \times 0 - 0.1 \times (-8) = \mathbf{0.8} \tag{A.7}$$
$$\boldsymbol{\theta}_1 = 1.0 + 0.8 = \mathbf{1.8} \tag{A.8}$$

Iteration 2:

- $\nabla\mathcal{L}(\theta_1) = 2(1.8 - 5) = -6.4$

$$\mathbf{v}_2 = 0.9 \times 0.8 - 0.1 \times (-6.4) = 0.72 + 0.64 = \mathbf{1.36} \tag{A.9}$$
$$\boldsymbol{\theta}_2 = 1.8 + 1.36 = \mathbf{3.16} \tag{A.10}$$

Iteration 3:

- $\nabla\mathcal{L}(\theta_2) = 2(3.16 - 5) = -3.68$

$$\mathbf{v}_3 = 0.9 \times 1.36 - 0.1 \times (-3.68) = 1.224 + 0.368 = \mathbf{1.592} \tag{A.11}$$
$$\boldsymbol{\theta}_3 = 3.16 + 1.592 = \mathbf{4.752} \tag{A.12}$$

Convergence Summary:

Iteration	Parameter	Velocity	Gradient
0	1.00	0.00	-8.00
1	1.80	0.80	-6.40
2	3.16	1.36	-3.68
3	4.752	1.592	-0.496

Key Observation: Momentum **accumulates gradient information**, allowing the algorithm to **accelerate** in consistent directions and **converge faster** than vanilla gradient descent.

A.8 Example: Vanishing Gradient in Deep Networks

Gradient Decay Through Layers
Consider a deep network with 5 layers, all using **sigmoid activation**. Starting with a loss gradient $\frac{\partial \mathcal{L}}{\partial z^{(5)}} = 1$.
Key fact: Maximum sigmoid derivative is $\sigma'(x) \leq \mathbf{0.25}$
Backpropagation through layers:

$$\frac{\partial \mathcal{L}}{\partial z^{(5)}} = 1 \tag{A.13}$$

$$\frac{\partial \mathcal{L}}{\partial z^{(4)}} = \frac{\partial \mathcal{L}}{\partial z^{(5)}} \cdot \sigma'(z^{(5)}) \leq 1 \times 0.25 = \mathbf{0.25} \tag{A.14}$$

$$\frac{\partial \mathcal{L}}{\partial z^{(3)}} \leq 0.25 \times 0.25 = \mathbf{0.0625} \tag{A.15}$$

$$\frac{\partial \mathcal{L}}{\partial z^{(2)}} \leq 0.0625 \times 0.25 = \mathbf{0.0156} \tag{A.16}$$

$$\frac{\partial \mathcal{L}}{\partial z^{(1)}} \leq 0.0156 \times 0.25 = \mathbf{0.0039} \tag{A.17}$$

Gradient magnitude: Decreases from 1 to 0.0039 (reduced by factor of ≈ 256)
With ReLU activation (derivative = 1):

$$\frac{\partial \mathcal{L}}{\partial z^{(5)}} = 1 \tag{A.18}$$

$$\frac{\partial \mathcal{L}}{\partial z^{(4)}} = 1 \times 1 = \mathbf{1} \tag{A.19}$$

$$\frac{\partial \mathcal{L}}{\partial z^{(3)}} = 1 \times 1 = \mathbf{1} \tag{A.20}$$

$$\frac{\partial \mathcal{L}}{\partial z^{(2)}} = 1 \times 1 = \mathbf{1} \tag{A.21}$$

$$\frac{\partial \mathcal{L}}{\partial z^{(1)}} = 1 \times 1 = \mathbf{1} \tag{A.22}$$

Conclusion: ReLU **prevents vanishing gradients** by maintaining constant gradient magnitude, enabling training of very deep networks.

A.9 Example: Xavier Initialization vs Random Initialization

Comparing Initialization Schemes
Consider layer with $n_{\text{in}} = 100$ inputs and $n_{\text{out}} = 50$ outputs.
Random Initialization (Naive):

$$\mathbf{W} \sim \mathcal{N}(0, 1) \quad \text{(standard normal)}$$

Expected variance of output: $\text{Var}(\mathbf{Wx}) = n_{\text{in}} \times 1 = 100$
Result: **Activations become very large** (saturation), gradients vanish.
Xavier Initialization:

$$\mathbf{W} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}\right) = \mathcal{N}(0, \sqrt{\frac{2}{150}}) = \mathcal{N}(0, 0.1155)$$

Expected variance of output: $\text{Var}(\mathbf{Wx}) = n_{\text{in}} \times \frac{2}{150} = 100 \times \frac{2}{150} = 1.33$
Result: **Activations remain in reasonable range**, better gradient flow.
He Initialization (for ReLU):

$$\mathbf{W} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{\text{in}}}}\right) = \mathcal{N}(0, \sqrt{\frac{2}{100}}) = \mathcal{N}(0, 0.1414)$$

Expected variance: $\text{Var}(\mathbf{Wx}) = 100 \times \frac{2}{100} = 2$
Summary Table:

Initialization	Weight Std	Output Variance
Random	1.0	100 (too large)
Xavier	0.1155	≈ 1.33 (good)
He (ReLU)	0.1414	≈ 2 (optimal for ReLU)

Practical Impact: Good initialization can **halve training time** or more!

A.10 Example: Learning Rate Scheduling Effects

Impact of Different Learning Rate Schedules
Training a network on a simple loss function. Base learning rate: $\alpha_0 = 0.1$
Constant Learning Rate:

Epoch	Learning Rate	Loss
0	0.100	2.50
10	0.100	0.85
50	0.100	0.12
100	0.100	0.08 (oscillates)

Step Decay: $\alpha_t = 0.1 \times 0.5^{\lfloor t/30 \rfloor}$

Epoch	Learning Rate	Loss
0	0.1000	2.50
10	0.1000	0.85
30	0.0500	0.18
60	0.0250	0.05
100	0.0125	0.028 (more stable)

Cosine Annealing: $\alpha_t = \frac{\alpha_0}{2}(1 + \cos(\pi t/100))$

Epoch	Learning Rate	Loss
0	0.0500	2.50
25	0.0383	0.42
50	0.0000	0.06
75	0.0383	0.035
100	0.1000	0.022 (smooth convergence)

Observations:

- **Constant learning rate**: Fast initial progress but oscillates near optimum
- **Step decay**: Stable convergence, requires tuning decay schedule
- **Cosine annealing**: Smooth convergence, good balance of speed and stability