Neural Network Theory: From Simple Network to Complex Handwritten digit Recognition

Complete Mathematical Fromulation Guide

CSCI 431 - Machine Learning
Artificial Neural Network Fundamentals
Prithvi Raj Singh

October 12, 2025

Abstract

This document provides a comprehensive mathematical formulation of feedforward neural networks, backpropagation, and stochastic gradient descent. It covers the theoretical foundations needed to implement a multi-layer perceptron from scratch, with specific applications to small networks - that accepts simple input number and the MNIST handwritten digit recognition problem - a classic ML problem. The sample codes are in JAVA, but one can implement simple neural network in any language once they understand the foundation.

Contents

1 Introduction to Neural Networks

1.1 Basic Concepts

A neural network is a computational model inspired by biological neural systems. It consists of interconnected processing units (neurons) organized in layers that transform input data into meaningful outputs through learned representations.

Definition 1.1 (Artificial Neuron)**.** *An artificial neuron computes a weighted sum of its inputs, adds a bias term, and applies an activation function:*

$$y = \sigma\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{1}$$

where x_i are inputs, w_i are weights, b is the bias, and σ is the activation function.

1.2 Network Architecture

Definition 1.2 (Multi-Layer Perceptron)**.** *A multi-layer perceptron (MLP) consists of:*

- ***Input Layer**: Receives raw input features $\mathbf{x} \in \mathbb{R}^{n_0}$*
- ***Hidden Layer(s)**: Intermediate representations $\mathbf{h} \in \mathbb{R}^{n_1}$*
- ***Output Layer**: Final predictions $\mathbf{y} \in \mathbb{R}^{n_2}$*

Let's consider two architectures:

1. **Small Network**: $4 \to 3 \to 2$ (4 inputs, 3 hidden, 2 outputs)
2. **MNIST Network**: $784 \to 15 \to 10$ (784 inputs, 15 hidden, 10 outputs)

2 Forward Propagation

2.1 Mathematical Formulation

Forward propagation computes the network output given an input vector.

Forward Propagation Algorithm

Given input $\mathbf{x} \in \mathbb{R}^{n_0}$, compute:
Layer 1 (Input $\to$ Hidden):

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

$$\mathbf{a}^{(1)} = \sigma(\mathbf{z}^{(1)}) \tag{3}$$

Layer 2 (Hidden $\to$ Output):

$$\mathbf{z}^{(2)} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)} \tag{4}$$

$$\mathbf{a}^{(2)} = \sigma(\mathbf{z}^{(2)}) \tag{5}$$

where:

- $\mathbf{W}^{(1)} \in \mathbb{R}^{n_1 \times n_0}$: Weight matrix from input to hidden
- $\mathbf{b}^{(1)} \in \mathbb{R}^{n_1}$: Bias vector for hidden layer
- $\mathbf{W}^{(2)} \in \mathbb{R}^{n_2 \times n_1}$: Weight matrix from hidden to output
- $\mathbf{b}^{(2)} \in \mathbb{R}^{n_2}$: Bias vector for output layer
- $\mathbf{z}^{(l)}$: Pre-activation (linear combination)
- $\mathbf{a}^{(l)}$: Activation (after applying σ)

2.2 Sigmoid Activation Function

Definition 2.1 (Sigmoid Function)**.** *The sigmoid (logistic) activation function is defined as:*

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{6}$$

Properties:

- Output range: $(0, 1)$
- Monotonically increasing
- Differentiable everywhere
- $\lim_{z\to\infty} \sigma(z) = 1$, $\lim_{z\to-\infty} \sigma(z) = 0$

Theorem 2.1 (Sigmoid Derivative)**.** *The derivative of the sigmoid function has a simple form:*

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z)) \tag{7}$$

Proof. Let $\sigma(z) = (1 + e^{-z})^{-1}$. Using the chain rule:

$$\begin{aligned}
\frac{d\sigma}{dz} &= -(1 + e^{-z})^{-2} \cdot (-e^{-z}) \\
&= \frac{e^{-z}}{(1 + e^{-z})^2} \\
&= \frac{1}{1 + e^{-z}} \cdot \frac{e^{-z}}{1 + e^{-z}} \\
&= \frac{1}{1 + e^{-z}} \cdot \frac{(1 + e^{-z}) - 1}{1 + e^{-z}} \\
&= \sigma(z) \cdot (1 - \sigma(z))
\end{aligned}$$

This derivative form is computationally efficient since we already have $\sigma(z)$ from forward propagation.

2.3 Matrix Operations

Definition 2.2 (Matrix Multiplication)**.** *For $\mathbf{A} \in \mathbb{R}^{m\times n}$ and $\mathbf{B} \in \mathbb{R}^{n\times p}$, the product $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{m\times p}$ is:*

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} \tag{8}$$

Definition 2.3 (Hadamard Product)**.** *The element-wise (Hadamard) product $\mathbf{C} = \mathbf{A} \odot \mathbf{B}$ is:*

$$C_{ij} = A_{ij} \cdot B_{ij} \tag{9}$$

3 Loss Function

3.1 Mean Squared Error

For regression and simple network, we use Mean Squared Error (MSE):

Definition 3.1 (MSE Loss)**.** *Given predicted output $\hat{\mathbf{y}}$ and target $\mathbf{y}$:*

$$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{2}\|\hat{\mathbf{y}} - \mathbf{y}\|^2 = \frac{1}{2}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{10}$$

The factor of $\frac{1}{2}$ simplifies the derivative. The gradient with respect to predictions is:

$$\frac{\partial \mathcal{L}}{\partial \hat{\mathbf{y}}} = \hat{\mathbf{y}} - \mathbf{y} \tag{11}$$

4 Backpropagation

4.1 The Chain Rule

Backpropagation is the application of the chain rule to compute gradients efficiently.

Theorem 4.1 (Chain Rule for Gradients)**.** *For a composition of functions $f(g(x))$:*

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x} \tag{12}$$

4.2 Gradient Computation

Backpropagation Algorithm

Step 1: Output Layer Error

$$\boldsymbol{\delta}^{(2)} = (\mathbf{a}^{(2)} - \mathbf{y}) \odot \sigma'(\mathbf{a}^{(2)}) \tag{13}$$

where $\sigma'(\mathbf{a}^{(2)}) = \mathbf{a}^{(2)} \odot (\mathbf{1} - \mathbf{a}^{(2)})$.

Step 2: Hidden Layer Error

$$\boldsymbol{\delta}^{(1)} = \left((\mathbf{W}^{(2)})^\top \boldsymbol{\delta}^{(2)}\right) \odot \sigma'(\mathbf{a}^{(1)}) \tag{14}$$

Step 3: Compute Gradients

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}} = \boldsymbol{\delta}^{(2)} (\mathbf{a}^{(1)})^\top \tag{15}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(2)}} = \boldsymbol{\delta}^{(2)} \tag{16}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} = \boldsymbol{\delta}^{(1)} \mathbf{x}^\top \tag{17}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(1)}} = \boldsymbol{\delta}^{(1)} \tag{18}$$

Step 4: Update Parameters

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} \tag{19}$$

$$\mathbf{b}^{(l)} \leftarrow \mathbf{b}^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} \tag{20}$$

where η is the learning rate.

4.3 Derivation of Backpropagation

4.3.1 Output Layer Gradient

Starting with the loss $\mathcal{L} = \frac{1}{2}\|\mathbf{a}^{(2)} - \mathbf{y}\|^2$:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(2)}} \frac{\partial \mathbf{a}^{(2)}}{\partial \mathbf{z}^{(2)}} \frac{\partial \mathbf{z}^{(2)}}{\partial \mathbf{W}^{(2)}} \tag{21}$$

$$= (\mathbf{a}^{(2)} - \mathbf{y}) \odot \sigma'(\mathbf{z}^{(2)}) \cdot (\mathbf{a}^{(1)})^\top \tag{22}$$

$$= \boldsymbol{\delta}^{(2)} (\mathbf{a}^{(1)})^\top \tag{23}$$

4.3.2 Hidden Layer Gradient

For the hidden layer, we propagate the error backwards:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(1)}} = (\mathbf{W}^{(2)})^\top \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(2)}} \tag{24}$$

$$= (\mathbf{W}^{(2)})^\top \boldsymbol{\delta}^{(2)} \tag{25}$$

Then:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{a}^{(1)}} \frac{\partial \mathbf{a}^{(1)}}{\partial \mathbf{z}^{(1)}} \frac{\partial \mathbf{z}^{(1)}}{\partial \mathbf{W}^{(1)}} \tag{26}$$

$$= \left[(\mathbf{W}^{(2)})^\top \boldsymbol{\delta}^{(2)} \odot \sigma'(\mathbf{z}^{(1)}) \right] \mathbf{x}^\top \tag{27}$$

$$= \boldsymbol{\delta}^{(1)} \mathbf{x}^\top \tag{28}$$

5 Stochastic Gradient Descent

5.1 Optimization Methods

Definition 5.1 (Batch Gradient Descent)**.** *Update parameters using the entire training set:*

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{N} \sum_{i=1}^{N} \nabla_{\mathbf{W}} \mathcal{L}(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) \tag{29}$$

Definition 5.2 (Stochastic Gradient Descent (SGD))**.** *Update parameters using one example at a time:*

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \nabla_{\mathbf{W}} \mathcal{L}(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) \tag{30}$$

Definition 5.3 (Mini-batch SGD)**.** *Update parameters using a small batch of B examples:*

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{1}{B} \sum_{j=1}^{B} \nabla_{\mathbf{W}} \mathcal{L}(\mathbf{x}^{(j)}, \mathbf{y}^{(j)}) \tag{31}$$

5.2 Mini-batch Training Algorithm

Algorithm 1 Mini-batch Stochastic Gradient Descent

Require: Training data $\{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{N}$, batch size B, learning rate η, epochs E
Ensure: Trained parameters $\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}$

```
Initialize weights randomly
for e = 1 to E do
    Shuffle training data
    for each mini-batch B = {(x^(j), y^(j))}_{j=1}^{B} do
        Initialize gradient accumulators: ∇W^(1) = 0, ∇b^(1) = 0, etc.
        for each (x^(j), y^(j)) ∈ B do
            Forward propagate: compute a^(1), a^(2)
            Backward propagate: compute δ^(1), δ^(2)
            Accumulate gradients: ∇W^(l) += ∂L/∂W^(l)
        end for
        Average gradients: ∇W^(l) ← (1/B) ∇W^(l)
        Update parameters: W^(l) ← W^(l) − η ∇W^(l)
    end for
    Evaluate performance on training/validation set
end for
```

5.3 Learning Rate Selection

The learning rate η controls the step size:

- **Too small** ($\eta \ll 1$): Slow convergence, many epochs needed
- **Too large** ($\eta \gg 1$): Unstable, may diverge
- **Recommended values**:
  - Small network (Part 1): $\eta = 10$
  - MNIST network (Part 2): $\eta = 3$

6 Implementation Details

6.1 Weight Initialization

- **Part 1**: Use predefined weights from spreadsheet
- **Part 2**: Random initialization uniformly in $[-1, 1]$

Listing 1: Random Weight Initialization in Java

```
Random rand = new Random();
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        weights[i][j] = rand.nextDouble() * 2 - 1;
    }
}
```

6.2 Input Normalization

MNIST pixel values range from 0 to 255. Normalize to $[0, 1]$:

$$x_{\text{norm}} = \frac{x_{\text{raw}}}{255} \tag{32}$$

This prevents sigmoid saturation and improves gradient flow.

6.3 One-Hot Encoding

Convert class labels to vectors:

Example 6.1 (One-Hot Encoding)**.** *For 10 classes (digits 0-9):*

$$\textit{Label } 0 \to [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]^\top$$
$$\textit{Label } 7 \to [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]^\top$$
$$\textit{Label } 9 \to [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]^\top$$

Listing 2: One-Hot Encoding in Java

```
public static double[] toOneHot(int label, int numClasses) {
    double[] oneHot = new double[numClasses];
    oneHot[label] = 1.0;
    return oneHot;
}
```

6.4 Making Predictions

The predicted class is the output neuron with maximum activation:

$$\hat{y} = \arg\max_i a_i^{(2)} \tag{33}$$

Listing 3: Get Prediction in Java

```java
public static int getPrediction(double[] output) {
    int maxIndex = 0;
    double maxValue = output[0];
    for (int i = 1; i < output.length; i++) {
        if (output[i] > maxValue) {
            maxValue = output[i];
            maxIndex = i;
        }
    }
    return maxIndex;
}
```

7 Worked Example

7.1 Problem Setup

Consider the small network with following layers:

- Architecture: $4 \to 3 \to 2$
- Input: $\mathbf{x} = [0, 1, 0, 1]^\top$
- Target: $\mathbf{y} = [0, 1]^\top$
- Learning rate: $\eta = 10$

Initial weights:

$$\mathbf{W}^{(1)} = \begin{bmatrix} -0.21 & 0.72 & -0.25 & 1.00 \\ -0.94 & -0.41 & -0.47 & 0.63 \\ 0.15 & 0.55 & -0.49 & -0.75 \end{bmatrix}, \quad \mathbf{b}^{(1)} = \begin{bmatrix} 0.10 \\ -0.36 \\ -0.31 \end{bmatrix} \tag{34}$$

$$\mathbf{W}^{(2)} = \begin{bmatrix} 0.76 & 0.48 & -0.73 \\ 0.34 & 0.89 & -0.23 \end{bmatrix}, \quad \mathbf{b}^{(2)} = \begin{bmatrix} 0.16 \\ -0.46 \end{bmatrix} \tag{35}$$

7.2 Forward Pass

Hidden layer:

$$\begin{aligned} \mathbf{z}^{(1)} &= \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \\ &= \begin{bmatrix} -0.21 & 0.72 & -0.25 & 1.00 \\ -0.94 & -0.41 & -0.47 & 0.63 \\ 0.15 & 0.55 & -0.49 & -0.75 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 0.10 \\ -0.36 \\ -0.31 \end{bmatrix} \\ &= \begin{bmatrix} 0.72 + 1.00 + 0.10 \\ -0.41 + 0.63 - 0.36 \\ 0.55 - 0.75 - 0.31 \end{bmatrix} = \begin{bmatrix} 1.82 \\ -0.14 \\ -0.51 \end{bmatrix} \end{aligned}$$

$$\begin{aligned} \mathbf{a}^{(1)} &= \sigma(\mathbf{z}^{(1)}) \\ &= \begin{bmatrix} \sigma(1.82) \\ \sigma(-0.14) \\ \sigma(-0.51) \end{bmatrix} = \begin{bmatrix} 0.8605 \\ 0.4651 \\ 0.3752 \end{bmatrix} \end{aligned}$$

Output layer:

$$\begin{aligned}\mathbf{z}^{(2)} &= \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)} \\ &= \begin{bmatrix} 0.76 & 0.48 & -0.73 \\ 0.34 & 0.89 & -0.23 \end{bmatrix} \begin{bmatrix} 0.8605 \\ 0.4651 \\ 0.3752 \end{bmatrix} + \begin{bmatrix} 0.16 \\ -0.46 \end{bmatrix} \\ &= \begin{bmatrix} 0.7633 \\ 0.1602 \end{bmatrix}\end{aligned}$$

$$\mathbf{a}^{(2)} = \sigma(\mathbf{z}^{(2)}) = \begin{bmatrix} 0.6821 \\ 0.5400 \end{bmatrix}$$

7.3 Backward Pass

Output deltas:

$$\begin{aligned}\boldsymbol{\delta}^{(2)} &= (\mathbf{a}^{(2)} - \mathbf{y}) \odot \mathbf{a}^{(2)} \odot (\mathbf{1} - \mathbf{a}^{(2)}) \\ &= \begin{bmatrix} 0.6821 - 0 \\ 0.5400 - 1 \end{bmatrix} \odot \begin{bmatrix} 0.6821 \times 0.3179 \\ 0.5400 \times 0.4600 \end{bmatrix} \\ &= \begin{bmatrix} 0.6821 \\ -0.4600 \end{bmatrix} \odot \begin{bmatrix} 0.2169 \\ 0.2484 \end{bmatrix} = \begin{bmatrix} 0.1479 \\ -0.1143 \end{bmatrix}\end{aligned}$$

Hidden deltas:

$$\begin{aligned}\boldsymbol{\delta}^{(1)} &= \left[(\mathbf{W}^{(2)})^\top \boldsymbol{\delta}^{(2)}\right] \odot \mathbf{a}^{(1)} \odot (\mathbf{1} - \mathbf{a}^{(1)}) \\ &= \begin{bmatrix} 0.76 & 0.34 \\ 0.48 & 0.89 \\ -0.73 & -0.23 \end{bmatrix} \begin{bmatrix} 0.1479 \\ -0.1143 \end{bmatrix} \odot \begin{bmatrix} 0.8605 \times 0.1395 \\ 0.4651 \times 0.5349 \\ 0.3752 \times 0.6248 \end{bmatrix} \\ &= \begin{bmatrix} 0.0736 \\ -0.0307 \\ -0.0817 \end{bmatrix} \odot \begin{bmatrix} 0.1200 \\ 0.2488 \\ 0.2344 \end{bmatrix} = \begin{bmatrix} 0.0088 \\ -0.0076 \\ -0.0192 \end{bmatrix}\end{aligned}$$

7.4 Weight Updates

Output weights:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}} &= \boldsymbol{\delta}^{(2)}(\mathbf{a}^{(1)})^\top \\ &= \begin{bmatrix} 0.1479 \\ -0.1143 \end{bmatrix} \begin{bmatrix} 0.8605 & 0.4651 & 0.3752 \end{bmatrix} \\ &= \begin{bmatrix} 0.1273 & 0.0688 & 0.0555 \\ -0.0983 & -0.0531 & -0.0429 \end{bmatrix}\end{aligned}$$

$$\begin{aligned}\mathbf{W}^{(2)}_{\text{new}} &= \mathbf{W}^{(2)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}} \\ &= \begin{bmatrix} 0.76 & 0.48 & -0.73 \\ 0.34 & 0.89 & -0.23 \end{bmatrix} - 10 \begin{bmatrix} 0.1273 & 0.0688 & 0.0555 \\ -0.0983 & -0.0531 & -0.0429 \end{bmatrix} \\ &= \begin{bmatrix} -0.513 & -0.208 & -1.285 \\ 1.323 & 1.421 & 0.199 \end{bmatrix}\end{aligned}$$

Hidden weights:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} &= \boldsymbol{\delta}^{(1)} \mathbf{x}^\top \\ &= \begin{bmatrix} 0.0088 \\ -0.0076 \\ -0.0192 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} 0 & 0.0088 & 0 & 0.0088 \\ 0 & -0.0076 & 0 & -0.0076 \\ 0 & -0.0192 & 0 & -0.0192 \end{bmatrix}\end{aligned}$$

Note: Gradients are zero for weights connected to zero inputs.

8 Performance Metrics

8.1 Accuracy

For classification tasks:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of examples}} \times 100\% \tag{36}$$

8.2 Per-Class Accuracy

For digit $d \in \{0, 1, \ldots, 9\}$:

$$\text{Accuracy}_d = \frac{\text{Correct predictions for digit } d}{\text{Total examples of digit } d} \tag{37}$$

8.3 Expected Results

Network	Configuration	Expected Accuracy
Part 1 (Small)	$\eta = 10$, 6 epochs	Outputs match spreadsheet
Part 2 (MNIST)	$\eta = 3$, batch=10, 30 epochs	$> 95\%$ on test set

Table 1: Expected performance metrics

9 Common Issues and Solutions

9.1 Numerical Stability

Problem: Sigmoid overflow for large $|z|$.
Solution: Clip extreme values:

Listing 4: Numerically Stable Sigmoid

```
public double sigmoid(double z) {
    if (z < -500) return 0.0;
    if (z > 500) return 1.0;
    return 1.0 / (1.0 + Math.exp(-z));
}
```

9.2 Vanishing Gradients

Problem: Gradients become very small in deep networks.
Cause: Sigmoid derivative $\sigma'(z) \leq 0.25$, so repeated multiplication makes gradients vanish.
Solutions:

- Use ReLU activation (for deeper networks)

- Proper weight initialization
- Batch normalization (advanced)

9.3 Debugging Strategies

1. **Gradient Checking**: Compare analytical gradients with numerical approximation:

$$\frac{\partial \mathcal{L}}{\partial w} \approx \frac{\mathcal{L}(w+\epsilon) - \mathcal{L}(w-\epsilon)}{2\epsilon} \tag{38}$$

   where $\epsilon = 10^{-7}$.

2. **Monitor Loss**: Loss should decrease monotonically (with mini-batch noise).
3. **Check Dimensions**: Ensure all matrix operations have compatible dimensions.
4. **Print Intermediate Values**: Compare with known correct values (e.g., spreadsheet).

10 Advanced Topics

10.1 Alternative Activation Functions

10.1.1 ReLU (Rectified Linear Unit)

$$\text{ReLU}(z) = \max(0, z), \quad \text{ReLU}'(z) = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases} \tag{39}$$

Advantages: No vanishing gradient, computationally efficient.

10.1.2 Tanh (Hyperbolic Tangent)

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \quad \tanh'(z) = 1 - \tanh^2(z) \tag{40}$$

Advantages: Zero-centered output, stronger gradients than sigmoid.

10.2 Regularization

10.2.1 L2 Regularization (Weight Decay)

Add penalty term to loss:

$$\mathcal{L}_{\text{reg}} = \mathcal{L} + \frac{\lambda}{2} \sum_l \|\mathbf{W}^{(l)}\|_F^2 \tag{41}$$

Gradient update becomes:

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \left(\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(l)}} + \lambda \mathbf{W}^{(l)} \right) \tag{42}$$

10.2.2 Dropout

Randomly set neuron activations to zero with probability p during training. Prevents co-adaptation of features.

10.3 Optimization Improvements

10.3.1 Momentum

$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + (1-\beta) \nabla_{\mathbf{W}} \mathcal{L} \tag{43}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \mathbf{v}_t \tag{44}$$

10.3.2 Adam Optimizer

Combines momentum with adaptive learning rates:

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1)\nabla_{\mathbf{W}}\mathcal{L} \tag{45}$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2)(\nabla_{\mathbf{W}}\mathcal{L})^2 \tag{46}$$

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1 - \beta_1^t}, \quad \hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1 - \beta_2^t} \tag{47}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon} \tag{48}$$

11 Implementation Checklist

11.1 Part 1: Small Network

Please don't expect to get similar output results when you run the code.

- ☐ Initialize weights with given values
- ☐ Implement sigmoid and its derivative
- ☐ Implement forward propagation (input → hidden → output)
- ☐ Implement backward propagation (compute deltas)
- ☐ Implement weight updates
- ☐ Train for few epochs with $\eta = 10$ or variable, batch size = 2

11.2 Part 2: MNIST Network

- ☐ Implement CSV data loading
- ☐ Normalize pixel values (divide by 255)
- ☐ Implement one-hot encoding
- ☐ Initialize weights randomly in $[-1, 1]$
- ☐ Implement mini-batch training with shuffling
- ☐ Implement save/load functionality
- ☐ Implement accuracy evaluation
- ☐ Implement ASCII visualization
- ☐ Create menu-driven interface
- ☐ Train for $20 \rightarrow 30 epochs with \eta = 3$, batch size = 10
- ☐ Achieve $> 90\%$ test accuracy

12 Mathematical Summary

Key Equations

Forward Propagation:

$$\begin{aligned}\mathbf{z}^{(1)} &= \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)} \\ \mathbf{a}^{(1)} &= \sigma(\mathbf{z}^{(1)}) \\ \mathbf{z}^{(2)} &= \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)} \\ \mathbf{a}^{(2)} &= \sigma(\mathbf{z}^{(2)})\end{aligned}$$

Backward Propagation:

$$\begin{aligned}\boldsymbol{\delta}^{(2)} &= (\mathbf{a}^{(2)} - \mathbf{y}) \odot \sigma'(\mathbf{a}^{(2)}) \\ \boldsymbol{\delta}^{(1)} &= [(\mathbf{W}^{(2)})^\top \boldsymbol{\delta}^{(2)}] \odot \sigma'(\mathbf{a}^{(1)})\end{aligned}$$

Gradients:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(2)}} &= \boldsymbol{\delta}^{(2)}(\mathbf{a}^{(1)})^\top \\ \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(2)}} &= \boldsymbol{\delta}^{(2)} \\ \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}} &= \boldsymbol{\delta}^{(1)}\mathbf{x}^\top \\ \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(1)}} &= \boldsymbol{\delta}^{(1)}\end{aligned}$$

Parameter Updates:

$$\begin{aligned}\mathbf{W} &\leftarrow \mathbf{W} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}} \\ \mathbf{b} &\leftarrow \mathbf{b} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{b}}\end{aligned}$$

Activation:

$$\begin{aligned}\sigma(z) &= \frac{1}{1 + e^{-z}} \\ \sigma'(a) &= a(1 - a)\end{aligned}$$

13 Conclusion

This document has presented the complete mathematical theory for implementing feedforward neural networks with backpropagation. The key insights are:

1. **Forward propagation** computes predictions through successive linear transformations and non-linear activations.
2. **Backpropagation** efficiently computes gradients by applying the chain rule backwards through the network.
3. **Stochastic gradient descent** optimizes parameters by iteratively moving in the direction that reduces loss.
4. **Mini-batches** provide a balance between computational efficiency and gradient stability.

The mathematics, while appearing complex, follows directly from calculus fundamentals. Understanding these principles enables implementation from scratch and provides the foundation for more advanced architectures.

A Matrix Calculus Identities

Function	Derivative
$f(\mathbf{x}) = \mathbf{A}\mathbf{x}$	$\nabla_{\mathbf{x}} f = \mathbf{A}^\top$
$f(\mathbf{x}) = \mathbf{x}^\top \mathbf{A}\mathbf{x}$	$\nabla_{\mathbf{x}} f = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$
$f(\mathbf{X}) = \text{tr}(\mathbf{A}\mathbf{X})$	$\nabla_{\mathbf{X}} f = \mathbf{A}^\top$
$f(x) = \sigma(x)$	$f'(x) = \sigma(x)(1 - \sigma(x))$

Table 2: Useful matrix calculus identities

B Notation Reference

Symbol	Meaning
$\mathbf{x}$	Input vector
$\mathbf{y}$	Target output vector
$\mathbf{W}^{(l)}$	Weight matrix for layer l
$\mathbf{b}^{(l)}$	Bias vector for layer l
$\mathbf{z}^{(l)}$	Pre-activation at layer l
$\mathbf{a}^{(l)}$	Activation at layer l
$\boldsymbol{\delta}^{(l)}$	Error term (delta) at layer l
σ	Sigmoid activation function
η	Learning rate
$\mathcal{L}$	Loss function
$\odot$	Element-wise (Hadamard) product
$\top$	Matrix transpose
∇	Gradient operator

Table 3: Mathematical notation used throughout this document

C Dimensionality Reference

For a network with architecture $n_0 \to n_1 \to n_2$:

Variable	Dimensions
$\mathbf{x}$	$n_0 \times 1$
$\mathbf{W}^{(1)}$	$n_1 \times n_0$
$\mathbf{b}^{(1)}$	$n_1 \times 1$
$\mathbf{z}^{(1)}$	$n_1 \times 1$
$\mathbf{a}^{(1)}$	$n_1 \times 1$
$\mathbf{W}^{(2)}$	$n_2 \times n_1$
$\mathbf{b}^{(2)}$	$n_2 \times 1$
$\mathbf{z}^{(2)}$	$n_2 \times 1$
$\mathbf{a}^{(2)}$	$n_2 \times 1$
$\mathbf{y}$	$n_2 \times 1$
$\boldsymbol{\delta}^{(2)}$	$n_2 \times 1$
$\boldsymbol{\delta}^{(1)}$	$n_1 \times 1$

Table 4: Dimensions for network variables

D Code Snippets

D.1 Sigmoid Function

Listing 5: Sigmoid Implementation

```
private double sigmoid(double z) {
    if (z < -500) return 0.0;
    if (z > 500) return 1.0;
    return 1.0 / (1.0 + Math.exp(-z));
}

private double sigmoidDerivative(double a) {
    return a * (1.0 - a);
}
```

D.2 Matrix Multiplication

Listing 6: Matrix Multiplication

```
public static double[][] matrixMultiply(double[][] A, double[][] B) {
    int m = A.length;
    int n = A[0].length;
    int p = B[0].length;
    double[][] C = new double[m][p];

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < p; j++) {
            C[i][j] = 0;
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return C;
}
```

D.3 Forward Propagation

Listing 7: Forward Propagation

```
public double[] forwardPropagate(double[] input) {
    // Hidden layer
    double[] hidden = new double[hiddenSize];
    for (int i = 0; i < hiddenSize; i++) {
        double sum = biasHidden[i];
        for (int j = 0; j < inputSize; j++) {
            sum += weightsInputHidden[i][j] * input[j];
        }
        hidden[i] = sigmoid(sum);
    }

    // Output layer
    double[] output = new double[outputSize];
    for (int i = 0; i < outputSize; i++) {
        double sum = biasOutput[i];
        for (int j = 0; j < hiddenSize; j++) {
            sum += weightsHiddenOutput[i][j] * hidden[j];
        }
        output[i] = sigmoid(sum);
    }

    return output;
}
```

Listing 8: Backward Propagation

```
public void backwardPropagate(double[] target) {
        System.out.println("\n=== Backward Propagation ===");
        System.out.println("Target: " + arrayToString(target));

        // STEP 1: Calculate output layer error
        // Delta = (Output - Target)       '(Output)
        double[] outputDeltas = new double[outputSize];
        for (int i = 0; i < outputSize; i++) {
            double error = outputActivations[i] - target[i];
            outputDeltas[i] = error * sigmoidDerivative(outputActivations[i]);
            System.out.printf("Output Delta[%d]: %.6f\n", i, outputDeltas[i]);
        }

        // STEP 2: Calculate hidden layer error
        // Error_hidden = W2^T    Delta_output
        // Delta_hidden = Error_hidden       '(Hidden)
        double[] hiddenDeltas = new double[hiddenSize];
        for (int i = 0; i < hiddenSize; i++) {
            double error = 0.0;
            // Sum of (weight    output_delta) for all output neurons
            for (int j = 0; j < outputSize; j++) {
                error += weightsHiddenOutput[j][i] * outputDeltas[j];
            }
            hiddenDeltas[i] = error * sigmoidDerivative(hiddenActivations[i]);
            System.out.printf("Hidden Delta[%d]: %.6f\n", i, hiddenDeltas[i]);
        }

        // STEP 3: Update weights and biases
        System.out.println("\n=== Weight Updates ===");

        // Update W2 (hidden     output weights)
        for (int i = 0; i < outputSize; i++) {
            for (int j = 0; j < hiddenSize; j++) {
                double gradient = outputDeltas[i] * hiddenActivations[j];
                double oldWeight = weightsHiddenOutput[i][j];
                weightsHiddenOutput[i][j] -= learningRate * gradient;
                System.out.printf("W2[%d][%d]: %.6f     %.6f (grad=%.6f)\n",
                    i, j, oldWeight, weightsHiddenOutput[i][j], gradient);
            }
        }

        // Update B2 (output biases)
        for (int i = 0; i < outputSize; i++) {
            double oldBias = biasOutput[i];
            biasOutput[i] -= learningRate * outputDeltas[i];
            System.out.printf("B2[%d]: %.6f     %.6f\n",
                i, oldBias, biasOutput[i]);
        }

        //Similar logic will be applied to update the W1 (input hidden weights)
            and B1 (hidden biases)
    }
```

References

[1] Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). *Learning representations by back-propagating errors*. Nature, 323(6088), 533-536.

[2] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). *Gradient-based learning applied to document recognition*. Proceedings of the IEEE, 86(11), 2278-2324.

[3] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.

[4] Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Determination Press.

[5] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.