Artificial Neural Networks – a gentle introduction for MCMP-21, University of Luxembourg

Kyunghoon Han & Sergio Suárez Dou

November 2024

1 Perceptron

Consider a problem predicting the target data $\mathbf{y} \in \mathbb{R}^n$ from the inputs $\mathbf{x} \in \mathbb{R}^n$. The prediction can be seen as an unknown function φ such that

$$\varphi(\mathbf{x}) = \mathbf{y}. \tag{1}$$

Any machine learning algorithm can be considered an approximation for obtaining this φ.

The simplest of such φ that produces a binary output (i.e. the outputs are 0 or 1) can take a linear form known as a *perceptron* as shown below

$$\varphi(x) \approx \varphi_p(x) = h(\mathbf{w} \cdot \mathbf{x}) + b = \hat{\mathbf{y}} \tag{2}$$

where $\mathbf{w} \in \mathbb{R}^n$ is called the weight vector, $h : \mathbb{R}^n \to 0, 1$ is the Heaviside step-function and $b \in \mathbb{R}$ is called the bias. The weight and the bias are the ones to fit so that the approximation of Equation 2 makes sense.

The question one can now ask is *how do we find these parameters?* The easiest approach can be obtained by comparing how different $\hat{\mathbf{y}}$ is from $\mathbf{y}$ then update linearly through small number $0 < \eta \ll 1$, called the *learning rate* as below.

$$w_i^{\text{new}} = w_i^{\text{old}} + \eta\,(y_i - \hat{y_i})\,x_i$$
$$b^{\text{new}} = b^{\text{old}} + \eta \sum_i (y_i - \hat{y_i})$$

Note that the subscript i stands for the i-th element of the respective vector. This simple algorithm is shown to be successful in predicting AND, OR, NOT, NAND, NOR gates. Figures 1 and 2 show the training accuracies over iterations and the final decision plots of AND and OR gates.

So far, the logic gates are well-reconstructed from the simple perceptron model. However, this fails once we try to fit an XOR gate as shown in Figre 3.

1.1 Problems of perceptrons

There are two issues with single-layer perceptrons as shown in Figure 3.

- When the data are intrinsically non-linear, the fitting process may exhibit significant errors.
- A single weight vector and bias term cannot adequately fit *all types* of non-linear data.

The former issue can be solved by replacing the Heaviside step function in Equation 2 to another nonlinear function, where the latter can be addressed by introducing multiple layers of perceptrons, each with its own set of weights and biases. These modifications lead us to the concept of multilayer perceptrons, which can handle more complex, nonlinear decision boundaries through a process called backpropagation.

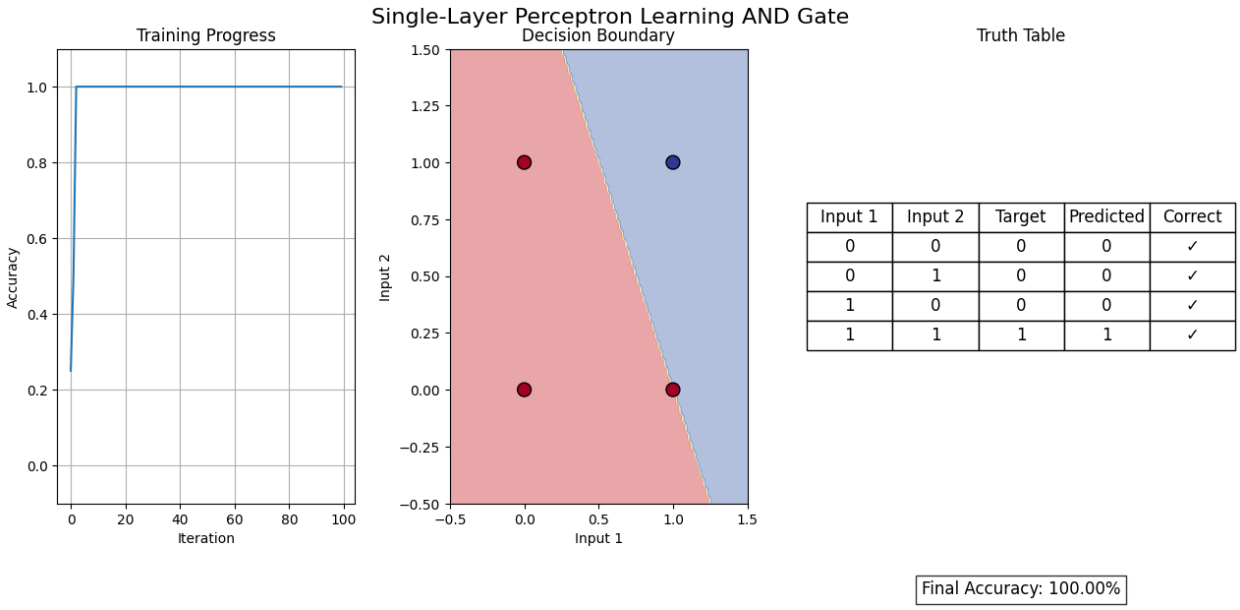

Figure 1: AND gate single-layer perceptron training result.

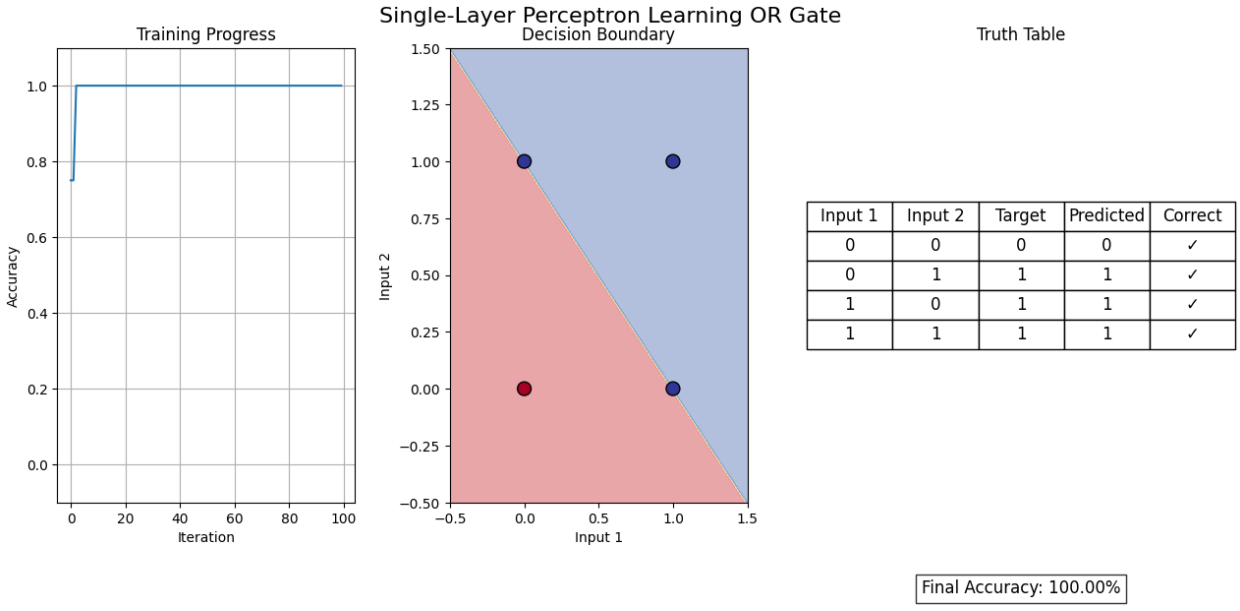

Figure 2: AND gate single-layer perceptron training result.

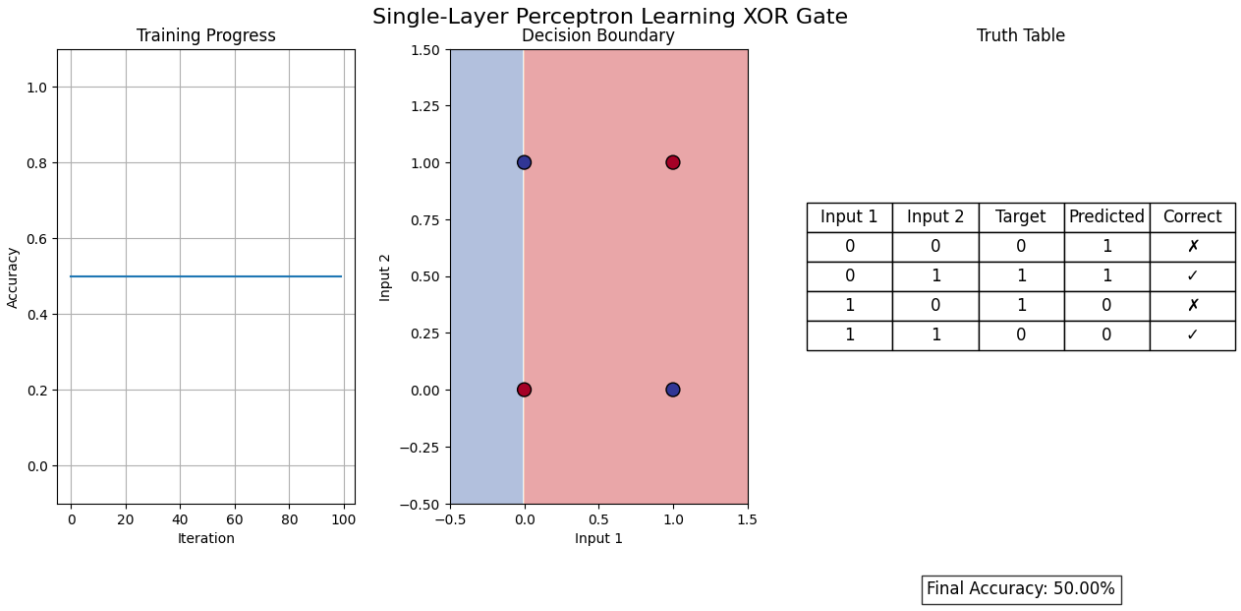

Figure 3: XOR gate single-layer perceptron training result.

1.2 Multilayer Perceptron

The multilayer perceptron (MLP) extends Equation 2 by introducing multiple layers of perceptrons. The key idea is to transform the input data through successive nonlinear mappings, allowing the network to learn more complex decision boundaries. Each transformation can be thought of as projecting the data into a new space where the classification becomes progressively easier.

For a network with L layers, the general form can be written as:

$$\mathbf{z}_l = \mathbf{W}_l \mathbf{a}_{l-1} + \mathbf{b}_l, \quad l = 1, \ldots, L \tag{3}$$

$$\mathbf{a}_l = h_l(\mathbf{z}_l) \tag{4}$$

where $\mathbf{a}_0 = \mathbf{x}$ is the input, $\mathbf{W}_l$ and $\mathbf{b}_l$ are the weights and biases for layer l, and h_l is a (nonlinear) function called *the activation function* for layer l. The final output $\mathbf{a}_L$ gives us our prediction.

For simplicity in understanding the learning process, let us consider a two-layer network with sigmoid activation functions, where the first layer contains multiple perceptrons (hidden layer) and the second layer contains a single output perceptron. This minimal architecture is already sufficient to solve nonlinear problems like the XOR gate that single-layer perceptrons could not handle.

The sigmoid function σ and its derivative are given by:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{5}$$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{6}$$

For input $\mathbf{x} \in \mathbb{R}^n$, our two-layer perceptron can be written as:

$$\mathbf{z}_1 = \mathbf{W}_1 \mathbf{x} + \mathbf{b}_1 \tag{7}$$

$$\mathbf{a}_1 = \sigma(\mathbf{z}_1) \tag{8}$$

$$z_2 = \mathbf{w}_2 \cdot \mathbf{a}_1 + b_2 \tag{9}$$

$$\hat{y} = \sigma(z_2) \tag{10}$$

where $\mathbf{W}_1 \in \mathbb{R}^{m \times n}$ is the first layer weight matrix, $\mathbf{b}_1 \in \mathbb{R}^m$ is the first layer bias vector, $\mathbf{w}_2 \in \mathbb{R}^m$ is the second layer weight vector, and $b_2 \in \mathbb{R}$ is the second layer bias. The dimension m represents the number of hidden units, which is a design choice that affects the network's capacity to learn complex patterns.

1.2.1 Backpropagation

The parameters are updated through a process called backpropagation, which computes the error from the output layer backward. This algorithm applies the chain rule of calculus to compute how each parameter contributes to the error. Let us first understand this process for our two-layer network before generalizing to arbitrary depths.

For a single training example $(\mathbf{x}, y)$, we define our error as:

$$E = \frac{1}{2}(\hat{y} - y)^2 \tag{11}$$

Using the chain rule, we can compute how each parameter affects this error. For the output layer:

$$\frac{\partial E}{\partial \mathbf{w}_2} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2} \frac{\partial z_2}{\partial \mathbf{w}_2} \tag{12}$$

$$= (\hat{y} - y)\sigma'(z_2)\mathbf{a}_1 \tag{13}$$

$$= \delta_2 \mathbf{a}_1 \tag{14}$$

where we define $\delta_2 = (\hat{y} - y)\sigma'(z_2)$ as the output layer error.

For the hidden layer, the chain rule gives us:

$$\frac{\partial E}{\partial \mathbf{W}_1} = \frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2} \frac{\partial z_2}{\partial \mathbf{a}_1} \frac{\partial \mathbf{a}_1}{\partial \mathbf{z}_1} \frac{\partial \mathbf{z}_1}{\partial \mathbf{W}_1} \tag{15}$$

$$= (\hat{y} - y)\sigma'(z_2)\mathbf{w}_2 \odot \sigma'(\mathbf{z}_1)\mathbf{x}^\top \tag{16}$$

$$= \boldsymbol{\delta}_1 \mathbf{x}^\top \tag{17}$$

where $\boldsymbol{\delta}_1 = (\mathbf{w}_2 \delta_2) \odot \sigma'(\mathbf{z}_1)$ represents the hidden layer error, and $\odot$ denotes element-wise multiplication.

This computation reveals a pattern that can be generalized to networks of arbitrary depth. For any layer l, the error term $\boldsymbol{\delta}_l$ depends on the error from the layer above $\boldsymbol{\delta}_{l+1}$ weighted by the connecting weights and multiplied element-wise with the derivative of the activation function.

1.2.2 General Backpropagation Algorithm

For a network with L layers, the algorithm proceeds as follows:

- Forward Pass: Compute all layer activations

$$\mathbf{z}_l = \mathbf{W}_l \mathbf{a}_{l-1} + \mathbf{b}_l, \quad l = 1, \ldots, L \tag{18}$$

$$\mathbf{a}_l = h_l(\mathbf{z}_l) \tag{19}$$

  where $\mathbf{a}_0 = \mathbf{x}$ is the input.

- Backward Pass: Compute error terms

$$\boldsymbol{\delta}_L = (\mathbf{a}_L - \mathbf{y}) \odot h_L'(\mathbf{z}_L) \tag{20}$$

$$\boldsymbol{\delta}_l = (\mathbf{W}_{l+1}^\top \boldsymbol{\delta}_{l+1}) \odot h_l'(\mathbf{z}_l), \quad l = L-1, \ldots, 1 \tag{21}$$

- Parameter Updates:

$$\mathbf{W}_l^{\text{new}} = \mathbf{W}_l^{\text{old}} - \eta \boldsymbol{\delta}_l \mathbf{a}_{l-1}^\top \tag{22}$$

$$\mathbf{b}_l^{\text{new}} = \mathbf{b}_l^{\text{old}} - \eta \boldsymbol{\delta}_l \tag{23}$$

  where η is the learning rate that controls the size of each update step.

This algorithm efficiently computes all partial derivatives through a single forward and backward pass, making it computationally feasible to train deep neural networks with many layers. Figure 4 demonstrates how even a simple two-layer network trained with this algorithm successfully learns the XOR function, which was impossible with a single-layer perceptron.

1.3 Universal Approximation Theorem

While multilayer perceptrons have shown remarkable capabilities in solving complex problems like the XOR gate, one might ask: what are the theoretical limits of such networks and how powerful can the single layer of perceptron be? The Universal Approximation Theorem provides a powerful answer to this question.

Theorem 1 (Universal Approximation)**.** *Let $C(X, \mathbb{R}^m)$ denote the set of continuous functions from a subset X of $\mathbb{R}^n$ to $\mathbb{R}^m$, and let $\sigma \in C(\mathbb{R}, \mathbb{R})$ be an activation function where $(\sigma \circ \mathbf{x})_i = \sigma(x_i)$ applies σ componentwise.*

Then σ is not polynomial if and only if for every:

- *$n, m \in \mathbb{N}$ (input and output dimensions)*
- *compact subset $K \subseteq \mathbb{R}^n$*
- *continuous function $f \in C(K, \mathbb{R}^m)$*
- *error tolerance $\varepsilon > 0$*

there exist parameters:

- *$k \in \mathbb{N}$ (number of hidden units)*
- *$\mathbf{A} \in \mathbb{R}^{k \times n}$ (weight matrix)*
- *$\mathbf{b} \in \mathbb{R}^k$ (bias vector)*
- *$\mathbf{C} \in \mathbb{R}^{m \times k}$ (output weights)*

such that

$$\sup_{\mathbf{x} \in K} \|f(\mathbf{x}) - g(\mathbf{x})\| < \varepsilon \tag{24}$$

where $g(\mathbf{x}) = \mathbf{C} \cdot (\sigma \circ (\mathbf{A} \cdot \mathbf{x} + \mathbf{b}))$.

This remarkable theorem states that a single hidden layer neural network with a non-polynomial activation function can approximate any continuous function to arbitrary precision over a compact domain. In other words, given enough hidden units, a single hidden layer is theoretically sufficient for universal approximation.

Some key implications of this theorem are:

- The choice of non-polynomial activation function (like sigmoid) is crucial
- The approximation is guaranteed only on compact (closed and bounded) subsets
- While the theorem guarantees existence, it doesn't provide the number of required hidden units
- Multiple hidden layers, while not theoretically necessary, may lead to more efficient representations in practice

This theoretical foundation helps explain why multilayer perceptrons have been so successful in various applications, from simple logic gates to complex pattern recognition tasks.

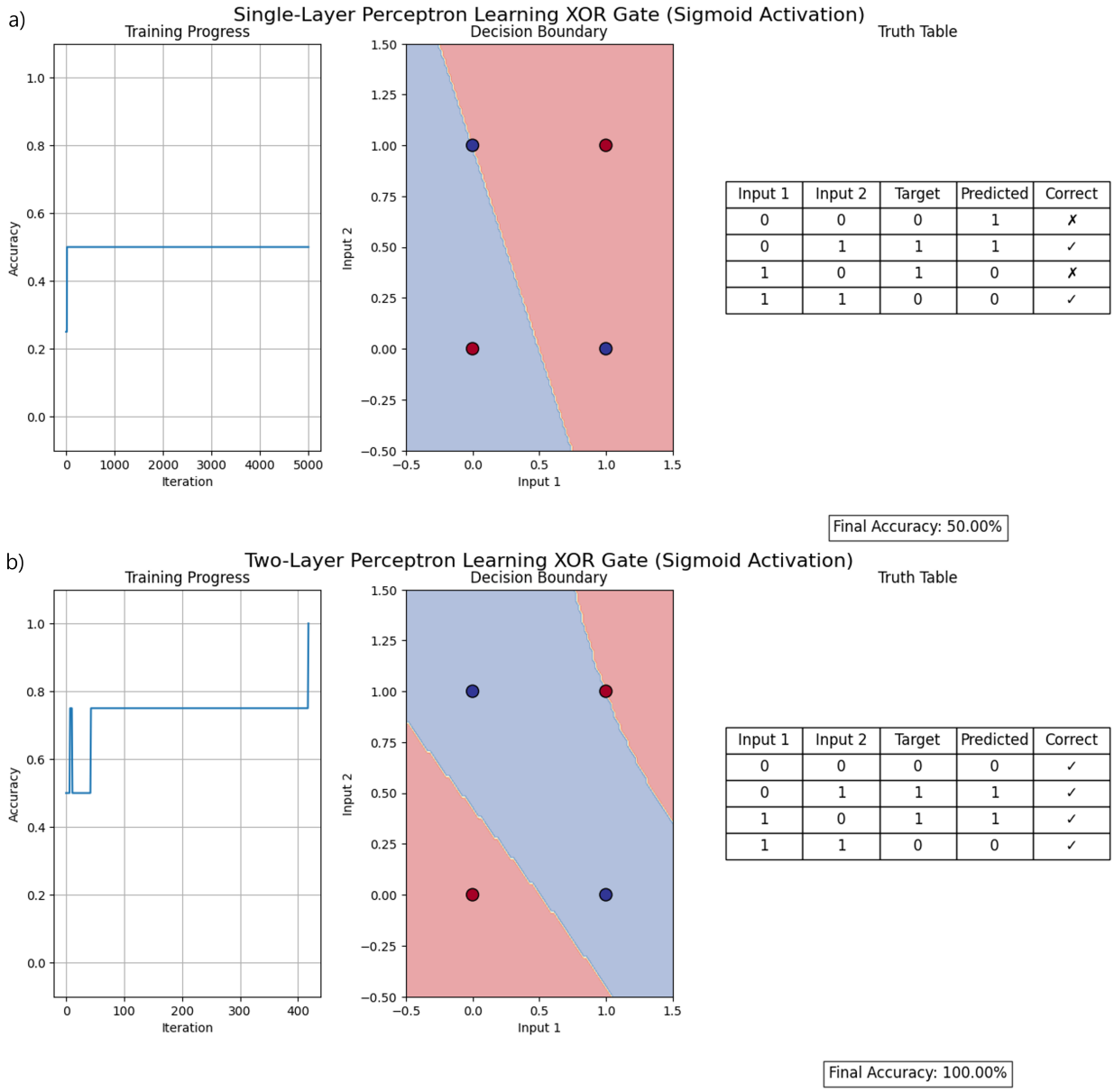

Figure 4: XOR gate single (a) and two (b) layer perceptron training results.

2 Neural Networks – Theory

The universal approximation theorem tells us that a single layer of weights with a proper activation function is enough for approximating arbitrary functions, or datasets. However, this also means that we may need infinitely many weight parameters to make a true *universal approximator*. One way to solve this matter is by introducing multiple layers and activations where the parameters are organically modified according to the data the machine faces. This section presents foundational knowledge the readers may need in their career on neural networks. In this section, the author will use the notation $\mathfrak{N}$ to denote a neural network for brevity.

Before jumping onto the main topic, let's define two terminologies: training step and epoch. A training step refers to a single update of the neural network model's parameters during training and an epoch referees to a complete pass through the entire training dataset. A neural network model sees and learns from all training data points exactly once in an epoch.

2.1 Forward propagation

Consider a neural network, $\mathfrak{N}$, with L layers. Each l-th layer serves as a transformation unit with three essential components:

1. Weight Matrix $\mathbf{W}_l \in \mathbb{R}^{n_l \times n_{l-1}}$:
   - Represents the connections between neurons in consecutive layers
   - Each row corresponds to a neuron in the l-th layer
   - Each column represents the influence of a neuron from $l-1$-th layer
   - The dimension n_l indicates the number of neurons in l-th layer
2. Bias Vector $\mathbf{b}_l \in \mathbb{R}^{n_l}$:
   - Allows the network to shift the activation function
   - Adds a constant term to the weighted sum
   - Helps the network learn patterns that don't pass through the origin
   - Each neuron has its own bias term
3. Activation Function $h_l : \mathbb{R} \to \mathbb{R}$:
   - Introduces non-linearity into the network
   - Allows the network to learn complex patterns
   - Maps the weighted sum to a new value range
   - Applied element-wise to the output of each neuron

The *forward propagation* on $\mathfrak{N}$ is then defined as:

$$\mathbf{x}_{l+1} = h_l\left(\mathbf{W}_l \cdot \mathbf{x}_l + \mathbf{b}_l\right), \quad \text{for} \quad l = 1, \ldots, L, \tag{25}$$

where the input to the first layer is $\mathbf{x}_1 = \mathbf{x}$, and the final output is $\mathbf{x}_{L+1} = y_{i,e}$, the output of the neural network at i-th training step of e-th epoch. Note that this step essentially evaluates the model being trained.

The choice of these components directly impacts the network's ability to learn. For instance:

- Larger weight matrices allow more complex patterns to be learned since the *latent spaces*, i.e. vector spaces spanned by the hidden layer outputs, will have higher dimensions
- Biases enable the network to shift decision boundaries
- Different activation functions suit different types of problems since the activations define the range of the output values.

2.2 Activation Functions and Loss Functions

Activation functions introduce nonlinearity into neural networks, enabling them to learn complex patterns and transformations. The choice of activation function is crucial as it determines the network's learning capacity, range of the problem the network can solve and training dynamics.

2.2.1 Common Activation Functions

1. The Sigmoid Function

   The sigmoid function, denoted as $\sigma : \mathbb{R} \to (0, 1)$, is defined as:

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

   Its derivative has a particularly elegant form:

   $$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

   Key mathematical properties:

   a. Bounded output: $\lim_{x\to\infty} \sigma(x) = 1$ and $\lim_{x\to-\infty} \sigma(x) = 0$

   b. Symmetry around the origin: $\sigma(-x) = 1 - \sigma(x)$

   c. Maximum gradient at x = 0: $\sigma'(0) = 0.25$

   d. Gradient vanishing: $\lim_{|x|\to\infty} \sigma'(x) = 0$

   The sigmoid function's biological motivation comes from its similarity to the firing rate of a neuron: gradually increasing activation until saturation.

2. The Rectified Linear Unit (ReLU)

   ReLU is defined as:

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

   Its derivative is:

   $$\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$$

   Mathematical properties:

   a. Scale-invariant: $\text{ReLU}(ax) = a\text{ReLU}(x)$ for $a > 0$

   b. No saturation in positive domain: prevents vanishing gradients

   c. Sparsity: produces true zero values, unlike sigmoid

   d. Non-differentiable at x = 0: typically handled by choosing either 0 or 1 as the gradient

3. The Softmax Function

   For an input vector $\mathbf{x} \in \mathbb{R}^n$, the softmax function $S : \mathbb{R}^n \to (0, 1)^n$ is defined as:

   $$S(\mathbf{x})_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

The Jacobian matrix has elements:

$$\frac{\partial S(\mathbf{x})_i}{\partial x_j} = S(\mathbf{x})_i(\delta_{ij} - S(\mathbf{x})_j)$$

Key properties:

a. Normalization: $\sum_i S(\mathbf{x})_i = 1$
b. Monotonicity: preserves the order of inputs
c. Scale invariance: $S(\mathbf{x} + c\mathbf{1}) = S(\mathbf{x})$ for any $c \in \mathbb{R}$
d. Differentiable: enables end-to-end training

2.2.2 Loss Functions

Loss functions measure the discrepancy between predicted and true values. The choice of loss function depends on the type of task and desired properties of the solution.

1. Binary Cross-Entropy

   For binary classification with true label $y \in \{0, 1\}$ and prediction $\hat{y} \in (0, 1)$:

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

   Mathematical properties:

   a. Non-negative: $\mathcal{L}(\hat{y}, y) \geq 0$
   b. Minimal at true values: $\mathcal{L}(y, y) = 0$
   c. Gradient: $\frac{\partial \mathcal{L}}{\partial \hat{y}} = -\frac{y}{\hat{y}} + \frac{1-y}{1-\hat{y}}$
   d. Convexity: strictly convex in $\hat{y}$

2. Categorical Cross-Entropy

   For multi-class classification with $\mathbf{y}, \hat{\mathbf{y}} \in [0, 1]^n$:

   $$\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$$

   Properties:

   a. Generalization of binary cross-entropy
   b. Suitable for one-hot encoded targets
   c. Gradient: $\frac{\partial \mathcal{L}}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i}$
   d. Often used with softmax activation

3. Mean Squared Error

   For regression problems:

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

   Mathematical characteristics:

   a. L^2 norm of error: $\mathcal{L}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{2n} \|\mathbf{y} - \hat{\mathbf{y}}\|_2^2$

b. Gradient: $\frac{\partial \mathcal{L}}{\partial \hat{y}_i} = \frac{1}{n}(\hat{y}_i - y_i)$

c. Strongly convex: unique global minimum

d. Sensitive to outliers due to squared term

The interaction between activation functions and loss functions is crucial for successful training. For instance, combining softmax activation with categorical cross-entropy loss provides numerical stability and meaningful gradients for multi-class classification.

2.3 Automatic Differentiation

Training neural networks requires computing gradients of the loss function with respect to all network parameters as the update rule for $\mathfrak{N}$ would involve the gradients. This is accomplished through automatic differentiation (AD), which provides an efficient and exact way to compute derivatives. Let's understand how AD works and why it's particularly well-suited for neural networks.

2.3.1 How Automatic Differentiation Works

Consider a simple function that we want to differentiate:

$$f(x) = \log(\sin(x^2))$$

AD breaks this into elementary operations and builds a computational graph:

$$v_1 = x^2$$

$$v_2 = \sin(v_1)$$

$$v_3 = \log(v_2) = f(x)$$

Each elementary operation has known derivatives that can be computed analytically. AD then applies the chain rule systematically to compute the overall derivative. There are two main modes of AD:

1. Forward Mode (forward propagation):
   - Computes derivatives alongside the function evaluation
   - Efficient when function has few inputs and many outputs
   - For each elementary operation, tracks both value and derivative:

$$\dot{v}_1 = \frac{\partial v_1}{\partial x}\dot{x} = 2x\dot{x}$$

$$\dot{v}_2 = \cos(v_1)\dot{v}_1$$

$$\dot{v}_3 = \frac{1}{v_2}\dot{v}_2$$

2. Reverse Mode (**backpropagation**):
   - First computes all function values, then propagates derivatives backward
   - Efficient when function has many inputs and few outputs (like neural networks)
   - Computes adjoints (partial derivatives) backward:

$$\bar{v}_3 = 1$$

$$\bar{v}_2 = \bar{v}_3 \cdot \frac{1}{v_2}$$

$$\bar{v}_1 = \bar{v}_2 \cdot \cos(v_1)$$

$$\bar{x} = \bar{v}_1 \cdot 2x$$

2.3.2 Why Reverse Mode for Neural Networks?

To understand why reverse mode automatic differentiation is crucial for neural networks, let's examine a concrete example and analyze its computational efficiency.

A Simple Neural Network Example

Consider a small neural network with:

- 2 hidden layers of 100 neurons each
- Input dimension of 10
- Output dimension of 1 (binary classification)

The number of parameters in this network would be:

$$\text{Layer 1: } 10 \times 100 + 100 = 1,100 \text{ parameters}$$

$$\text{Layer 2: } 100 \times 100 + 100 = 10,100 \text{ parameters}$$

$$\text{Output: } 100 \times 1 + 1 = 101 \text{ parameters}$$

$$\text{Total: } 11,301 \text{ parameters}$$

Computational Complexity Analysis

When training this network, we need to compute derivatives of the loss function with respect to all parameters. Let's compare forward and reverse mode AD:

1. Forward Mode:
   - Must perform one sweep for each parameter
   - Number of sweeps = 11,301
   - Each sweep computes one partial derivative
   - Total complexity: $\mathcal{O}(np)$ where n is number of parameters
2. Reverse Mode:
   - Performs one forward pass to compute values
   - One backward pass computes all derivatives
   - Number of sweeps = 1
   - Total complexity: $\mathcal{O}(p)$ where p is number of operations

Real-World Scale Modern neural networks are much larger:

- ResNet-50 has about 25 million parameters
- GPT-3 has more than 200 billion parameters
- Forward mode would require billions of sweeps
- Reverse mode still requires just one sweep

2.3.3 Error Terms

The error terms $\boldsymbol{\delta}_l$ represent how the loss changes with respect to the pre-activation values at each layer.

1. Output Layer Error

   For the final layer L:

   $$\boldsymbol{\delta}_L = \frac{\partial \mathcal{L}}{\partial \mathbf{a}_L} \odot h'_L(\mathbf{z}_L)$$

   Components:

   a. $\frac{\partial \mathcal{L}}{\partial \mathbf{a}_L}$: How loss changes with output
   b. $h'_L(\mathbf{z}_L)$: Local gradient of activation
   c. $\odot$: Element-wise multiplication
   d. This error initiates the backward pass

2. Hidden Layer Error

   For layers $l < L$:

   $$\boldsymbol{\delta}_l = (\mathbf{W}_{l+1}^\top \boldsymbol{\delta}_{l+1}) \odot h'_l(\mathbf{z}_l)$$

   Components:

   a. $\mathbf{W}_{l+1}^\top \boldsymbol{\delta}_{l+1}$: Error propagated from next layer
   b. $h'_l(\mathbf{z}_l)$: Local gradient of activation
   c. This represents the chain rule application
   d. Error diminishes with depth (vanishing gradient problem)

2.3.4 Gradient Computation

Given the error terms, we can compute gradients for all parameters:

1. Weight Gradients

   For each layer l:

   $$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_l} = \boldsymbol{\delta}_l \mathbf{a}_{l-1}^\top$$

   Properties:

   a. Outer product of error and input
   b. Matches weight matrix dimensions
   c. Represents credit assignment
   d. Scale depends on activation magnitudes

2. Bias Gradients

   For each layer l:

   $$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_l} = \boldsymbol{\delta}_l$$

   Properties:

   a. Direct use of error term
   b. Independent of input activations
   c. Usually smaller than weight gradients
   d. Represents baseline adjustment

2.3.5 Parameter Updates

The final step is updating the parameters using computed gradients:

1. Weight Updates

   Using learning rate η:

   $$\mathbf{W}_l^{(t+1)} = \mathbf{W}_l^{(t)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{W}_l}$$

   Characteristics:

   a. Learning rate controls update magnitude
   b. Updates move in direction of steepest descent
   c. Momentum can be added for stability
   d. Adaptive methods adjust η per parameter

2. Bias Updates

   Similarly for biases:

   $$\mathbf{b}_l^{(t+1)} = \mathbf{b}_l^{(t)} - \eta \frac{\partial \mathcal{L}}{\partial \mathbf{b}_l}$$

   Characteristics:

   a. Same learning rate as weights
   b. Updates are typically smaller
   c. Less affected by vanishing gradients
   d. Can be initialized to zero

2.4 Weight Initialization

Proper initialization of neural network parameters is crucial for successful training. The main goal is to prevent both the vanishing and exploding gradient problems while maintaining useful signal propagation through the network.

Xavier/Glorot initialization, proposed by Xavier Glorot and Yoshua Bengio in 2010, addresses the challenge of maintaining variance across layers in deep neural networks. Their key insight was that the variance of the activations and gradients should remain consistent across layers to prevent exponential growth or decay. For a layer with n_{l-1} inputs and n_l outputs, they showed that initializing weights from a distribution with variance $2/(n_l + n_{l-1})$ helps maintain this property, particularly when using symmetric activation functions like tanh or sigmoid. This initialization scheme assumes that the activation function operates in its linear regime around zero, where its derivative is approximately one.

He initialization, introduced by Kaiming He in 2015, modifies the Xavier initialization to account for the specific properties of ReLU activation functions. The key difference is that ReLU sets negative inputs to zero, effectively reducing the variance of its output by half compared to its input. To compensate for this variance reduction, He initialization uses a larger scaling factor, drawing weights from a distribution with variance $2/n_{l-1}$. This adjustment ensures that the variance of the activations remains constant across layers even with the ReLU nonlinearity, which is crucial for training very deep networks.

Together, these initialization schemes highlight the importance of considering the interaction between weight initialization and activation functions. While Xavier initialization is optimal for activation functions that preserve variance in their linear regime (like tanh), He initialization is specifically designed for ReLU and its variants, where the variance-reducing property of the activation function must be explicitly compensated for in the initialization. This understanding has been fundamental in enabling the training of increasingly deep neural networks, as proper initialization ensures stable gradient flow during the critical early stages of training.

1. Principles of Weight Initialization

   The key objectives of weight initialization are:

$$\text{Var}(\mathbf{z}_l) \approx \text{Var}(\mathbf{z}_{l-1})$$

$$\text{Var}(\frac{\partial \mathcal{L}}{\partial \mathbf{z}_l}) \approx \text{Var}(\frac{\partial \mathcal{L}}{\partial \mathbf{z}_{l-1}})$$

   Properties:

   a. Forward signal propagation should maintain variance
   b. Backward gradient flow should maintain magnitude
   c. Different activations require different scalings
   d. Initialization affects optimization trajectory

2. Xavier/Glorot Initialization

   For sigmoid/tanh activations:

$$\mathbf{W}_l \sim \mathcal{N}(0, \sqrt{\frac{2}{n_l + n_{l-1}}})$$

   Derivation and properties:

   a. Assumes linear region of sigmoid/tanh where $h'(0) \approx 1$
   b. For input x with Var(x) = 1, the variance of output z is:

$$\text{Var}(z) = n_{l-1} \cdot \text{Var}(w) \cdot \text{Var}(x)$$

   c. Setting Var(z) = 1 leads to:

$$\text{Var}(w) = \frac{2}{n_l + n_{l-1}}$$

   d. This balances forward and backward signal propagation for symmetric activations

3. He Initialization

   For ReLU activations:

$$\mathbf{W}_l \sim \mathcal{N}(0, \sqrt{\frac{2}{n_{l-1}}})$$

   Theoretical basis:

   a. ReLU sets half its inputs to zero: $\mathbb{E}[h'(x)] = \frac{1}{2}$
   b. For preserved variance after ReLU:

$$\text{Var}(h(z)) = \frac{1}{2}\text{Var}(z)$$

   c. Compensating for this halving effect requires larger initial weights
   d. Results in variance-preserving initialization for ReLU networks

4. Zero Initialization for Biases

   Setting all biases to zero:

$$\mathbf{b}_l = \mathbf{0}$$

   Rationale:

a. Breaking symmetry is handled by random weight initialization
b. Zero initialization centers activations initially
c. For ReLU, positive weights ensure gradient flow
d. Exception: output layer bias might be initialized based on target distribution

5. Practical Considerations

   Implementation aspects:

   a. Use appropriate initialization based on activation function:

   $$\text{sigmoid/tanh} \rightarrow \text{Xavier}; \quad \text{ReLU} \rightarrow \text{He}$$

   b. Consider layer width when choosing scale factors
   c. Monitor initial activations and gradients
   d. Batch normalization can help mitigate poor initialization

Understanding proper initialization is crucial because:

1. Poor initialization can lead to saturated neurons (vanishing gradients)
2. Excessive initial values can cause exploding gradients
3. Proper scaling ensures effective gradient flow
4. Different activation functions require different initialization strategies

The key is maintaining appropriate signal magnitude throughout the network, both during forward propagation and backpropagation. This ensures that learning can begin effectively from the first iteration.

2.5 Mini-batch Processing and Numerical Stability

Training neural networks on large datasets requires efficient processing strategies. Mini-batch processing offers a balance between computational efficiency and statistical stability in gradient estimation.

2.5.1 Batch Gradient Computation

The fundamental idea of mini-batch processing is to estimate gradients using a subset of training data. For a mini-batch of size m:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_l} = \frac{1}{m} \sum_{i=1}^{m} \boldsymbol{\delta}_l^{(i)} (\mathbf{a}_{l-1}^{(i)})^\top$$

1. Statistical Properties

   a. Unbiased estimate of full gradient: $\mathbb{E}[\nabla \mathcal{L}_{\text{batch}}] = \nabla \mathcal{L}_{\text{full}}$
   b. Variance scales with $1/m$: $\text{Var}(\nabla \mathcal{L}_{\text{batch}}) \propto 1/m$
   c. Trade-off between computation time and estimation accuracy
   d. Natural regularization through noise in gradient estimates

2. Learning Rate Adjustment

   The effective learning rate needs adjustment based on batch size:

   $$\eta_{\text{effective}} = \frac{\eta}{\sqrt{m}}$$

   Reasoning:

a. Gradient variance affects update magnitude
b. Larger batches require larger learning rates
c. Square root scaling balances noise and convergence
d. Empirically validated on various architectures

2.5.2 Numerical Considerations

Training deep neural networks requires careful attention to numerical stability to ensure reliable convergence.

1. Softmax Stability

   The numerically stable softmax computation:

   $$\text{softmax}(\mathbf{x})_i = \frac{e^{x_i - \max_j x_j}}{\sum_j e^{x_j - \max_j x_j}}$$

   Key aspects:

   a. Prevents overflow in exponentials
   b. Maintains relative differences: $\max_j x_j$ subtraction is translation-invariant
   c. Improves numerical precision in floating-point arithmetic
   d. Particularly important for large logits

2. Gradient Clipping

   To prevent exploding gradients, implement norm-based clipping:

   $$\|\boldsymbol{\delta}_l\| \leq \text{threshold}$$

   Implementation:

   a. Global gradient norm: $g_{\text{norm}} = \sqrt{\sum_l \|\boldsymbol{\delta}_l\|^2}$
   b. Scaling factor: $\lambda = \min(1, \frac{\text{threshold}}{g_{\text{norm}}})$
   c. Clipped gradients: $\boldsymbol{\delta}_l^{\text{clipped}} = \lambda \boldsymbol{\delta}_l$
   d. Typical thresholds range from 1 to 10

3. Activation Function Stability

   For sigmoid activation:

   $$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

   Numerical considerations:

   a. Avoids separate computation of exponentials
   b. Reuses computed sigmoid values
   c. Prevents underflow/overflow in derivative computation
   d. Similar tricks apply for tanh: $\tanh'(x) = 1 - \tanh^2(x)$

4. Loss Function Stability

   For cross-entropy with softmax:

   $$\mathcal{L} = -\sum_i y_i \log(\hat{y}_i + \epsilon)$$

   Stabilization techniques:

a. Add small epsilon ($\epsilon \approx 10^{-7}$) to prevent log(0)

b. Use numerically stable log-sum-exp tricks

c. Combine softmax and cross-entropy computation

d. Monitor for NaN/Inf values during training

2.5.3 Implementation Guidelines

When implementing these considerations:

1. Choose batch size based on:

   a. Available memory

   b. Desired training stability

   c. Hardware utilization

   d. Convergence requirements

2. Monitor numerical stability through:

   a. Gradient norms across layers

   b. Activation statistics

   c. Loss values and their gradients

   d. Parameter update magnitudes

Proper handling of these numerical considerations ensures stable and efficient training, particularly for deep architectures where small numerical instabilities can compound across layers.

3 Neural Network Implementation: MNIST Classifier

The MNIST handwritten digit classification task serves as an excellent introduction to neural network implementation. By building this classifier from scratch, we can understand how each component of a neural network works together, from the basic tensor operations to the complete training process. Our implementation demonstrates key concepts including automatic differentiation, backpropagation, and efficient batch processing.

3.1 Core Components

3.1.1 The Tensor Class: Foundation of Automatic Differentiation

At the heart of our neural network implementation lies the Tensor class, which forms the foundation for automatic differentiation. Unlike simple numerical arrays, our Tensor class needs to track both data and gradients:

Listing 1: Tensor Class Implementation

```
class Tensor:
    def __init__(self, data, requires_grad=False):
        self.data = np.array(data, dtype=np.float32)
        self.requires_grad = requires_grad
        self.grad = None if requires_grad else None
```

This seemingly simple class handles three crucial aspects:

1. **Data Storage**: The `data` attribute stores the actual numerical values as 32-bit floating-point numbers. We use NumPy arrays for efficient numerical computations.

2. **Gradient Tracking**: The `requires_grad` flag indicates whether we need to compute gradients for this tensor during backpropagation. This is particularly important for parameters like weights and biases that need updating during training.

3. **Gradient Storage**: The `grad` attribute stores the computed gradients during backpropagation. It's initialized as None and gets populated during the backward pass.

The Tensor class also implements basic mathematical operations through operator overloading:

Listing 2: Tensor Operations

```
def __add__(self, other):
    if not isinstance(other, Tensor):
        other = Tensor(other)
    return Tensor(self.data + other.data)

def __mul__(self, other):
    if not isinstance(other, Tensor):
        other = Tensor(other)
    return Tensor(self.data * other.data)

def __matmul__(self, other):
    if not isinstance(other, Tensor):
        other = Tensor(other)
    return Tensor(self.data @ other.data)
```

These operations enable us to write natural mathematical expressions while maintaining the computational graph needed for automatic differentiation. For instance, when we write `a @ w + b` for a linear layer, we're actually creating a chain of Tensor operations that track the relationships needed for gradient computation.

3.1.2 The Module System

Building on the Tensor foundation, we implement a module system that provides a structured way to compose neural network layers. The base Module class defines the interface for all neural network components:

Listing 3: Base Module Implementation

```
class Module:
    def zero_grad(self):
        for p in self.parameters():
            if p.grad is not None:
                p.grad = np.zeros_like(p.data)

    def parameters(self):
        return []
```

The `Module` class serves several important purposes:

1. **Parameter Management**: Through the `parameters()` method, modules can expose their trainable parameters. This is crucial for optimization as it provides access to all parameters that need gradient updates.

2. **Gradient Zeroing**: The `zero_grad()` method resets gradients before each backward pass, ensuring that gradients don't accumulate incorrectly across multiple backward passes.

3. Building on this foundation, we implement the Sequential module that allows us to chain layers together:

Listing 4: Sequential Module Implementation

```
class Sequential(Module):
    def __init__(self, *layers):
        self.layers = layers

    def forward(self, x):
        for layer in self.layers:
            x = layer.forward(x)
        return x

    def parameters(self):
        return [p for layer in self.layers
                for p in layer.parameters()]
```

The Sequential module provides two key functionalities:

1. **Forward Propagation**: The `forward` method chains the computation through each layer sequentially, transforming the input data through the network.

2. **Parameter Collection**: The `parameters` method aggregates parameters from all layers, which is essential for optimization during training.

3.2 Layer Implementations

3.2.1 Linear Layer

The Linear layer implements the affine transformation fundamental to neural networks. It performs the operation $\mathbf{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$, where $\mathbf{W}$ represents the weights and $\mathbf{b}$ the bias:

Listing 5: Linear Layer Implementation

```
class Linear(Module):
   def __init__(self, in_features, out_features):
       # He initialization for better gradient flow
       scale = np.sqrt(2.0 / in_features)
       self.weights = Tensor(
           np.random.normal(0, scale, (in_features, out_features)),
           requires_grad=True
       )
       self.bias = Tensor(np.zeros(out_features), requires_grad=True)

   def forward(self, x):
       return x @ self.weights + self.bias

   def parameters(self):
       return [self.weights, self.bias]
```

Several key aspects of the Linear layer implementation deserve attention:

Weight Initialization:

We use He initialization, scaling the weights by $\sqrt{2/n_{in}}$ where n_{in} is the number of input features. This initialization strategy helps maintain appropriate gradient magnitudes during backpropagation, particularly with ReLU activation functions. Without proper scaling, gradients could vanish or explode during training.

Parameter Management:

Both weights and bias are created as Tensor objects with `requires_grad=True`, indicating they should accumulate gradients during backpropagation. The `parameters()` method exposes these tensors for optimization.

Forward Computation:

The forward method implements the affine transformation using matrix multiplication (@) and addition operations defined in our Tensor class. This operation maintains the computational graph needed for automatic differentiation.

3.2.2 Activation Functions

Our implementation includes two crucial activation functions: ReLU and Softmax. Each serves a distinct purpose in the network:

Listing 6: ReLU Implementation

```
class ReLU(Module):
    def forward(self, x):
        return Tensor(np.maximum(0, x.data))
```

The Rectified Linear Unit (ReLU) introduces non-linearity into the network while maintaining simple gradient computation. Its operation can be expressed mathematically as:

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

The gradient of ReLU is straightforward:

$$\frac{\partial \text{ReLU}}{\partial x} = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

For the output layer, we implement the Softmax activation:

Listing 7: Softmax Implementation

```
class Softmax(Module):
    def forward(self, x):
        exp_x = np.exp(x.data - np.max(x.data, axis=1, keepdims=True))
        return Tensor(exp_x / np.sum(exp_x, axis=1, keepdims=True))
```

The Softmax function converts the network's raw outputs into a probability distribution:

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Our implementation includes two important numerical considerations:

1. **Maximum Subtraction**: We subtract the maximum value before exponentiating to prevent numerical overflow:
$$\text{Softmax}(x_i) = \frac{e^{x_i - \max_j x_j}}{\sum_j e^{x_j - \max_j x_j}}$$
2. **Batch Processing**: The implementation maintains proper dimensions for batch processing using `keepdims=True`, allowing efficient processing of multiple samples simultaneously.

3.3 The MNIST Classifier Architecture

With these components in place, we can now implement our MNIST classifier:

Listing 8: MNIST Classifier Implementation

```
class MNISTClassifier(Sequential):
   def __init__(self):
       super().__init__(
           Linear(784, 256),  # Input layer -> First hidden layer
           ReLU(),            # First activation
           Linear(256, 32),   # First hidden -> Second hidden
           ReLU(),            # Second activation
           Linear(32, 10),    # Second hidden -> Output
           Softmax()          # Output activation
       )
```

The architecture follows a dimensionality reduction pattern:

1. **Input Layer** ($784 \rightarrow 256$): Processes the flattened 28×28 MNIST images (784 pixels) and maps them to a 256-dimensional hidden representation. The ReLU activation allows the network to learn non-linear features from the input.

2. **Hidden Layer** ($256 \rightarrow 32$): Compresses the representation further to 32 dimensions, maintaining essential features for classification while reducing model complexity. Another ReLU activation maintains non-linearity.

3. **Output Layer** ($32 \rightarrow 10$): Produces logits for each of the 10 possible digits (0-9). The Softmax activation converts these logits into probabilities, facilitating classification and training with cross-entropy loss.

3.4 Training Process and Backpropagation

3.4.1 Loss Function

For our classification task, we implement the cross-entropy loss function, which measures the difference between predicted and true probability distributions:

Listing 9: Cross Entropy Loss Implementation

```
class CrossEntropyLoss:
   def __call__(self, pred, target):
       eps = 1e-12
       pred_clipped = np.clip(pred.data, eps, 1 - eps)
       loss = -np.sum(target.data * np.log(pred_clipped)) / pred.data.shape[0]
       out = Tensor(loss)

       def _backward():
           # Gradient of cross entropy
           if pred.requires_grad:
               pred.grad = (pred_clipped - target.data) / pred.data.shape[0]

       out._backward = _backward
       out._parents = [pred]
       return out
```

The cross-entropy loss for a batch of N samples is computed as:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij}\log(\hat{y}_{ij})$$

where y_{ij} is the true probability (0 or 1) of sample i belonging to class j, and $\hat{y}_{ij}$ is the predicted probability. Several numerical considerations are implemented:

1. Adding a small epsilon (ϵ) to prevent taking log of zero
2. Clipping predictions to the range $[\epsilon, 1-\epsilon]$
3. Averaging the loss over the batch size
4. Storing the backward computation for automatic differentiation

3.5 Training Implementation

The training process is implemented in the Trainer class, which handles the forward pass, backward pass, and parameter updates:

Listing 10: Trainer Implementation

```
class Trainer:
   def __init__(self, model, learning_rate=0.01):
       self.model = model
       self.learning_rate = learning_rate

   def train_epoch(self, train_loader, desc="Training"):
       total_loss = 0
       correct = 0
       n_samples = 0

       for X, y in tqdm(train_loader, desc=desc):
           batch_size = X.shape[0]
           n_samples += batch_size

           # Forward pass through the network
           h1 = X @ self.model.layers[0].weights.data + \
                self.model.layers[0].bias.data
           a1 = np.maximum(0, h1)  # ReLU
           h2 = a1 @ self.model.layers[2].weights.data + \
                self.model.layers[2].bias.data
           a2 = np.maximum(0, h2)  # ReLU
           logits = a2 @ self.model.layers[4].weights.data + \
                    self.model.layers[4].bias.data
           probs = self._softmax(logits)
```

The forward pass computes several intermediate values that are needed for both prediction and gradient computation:

1. **Layer 1**: Input transformation ($\mathbf{h_1} = \mathbf{X}\mathbf{W_1} + \mathbf{b_1}$) followed by ReLU activation
2. **Layer 2**: Hidden layer transformation ($\mathbf{h_2} = \mathbf{a_1}\mathbf{W_2} + \mathbf{b_2}$) with ReLU
3. **Output**: Final transformation ($\mathbf{logits} = \mathbf{a_2}\mathbf{W_3} + \mathbf{b_3}$) with Softmax

3.6 Gradient Computation

The gradient computation follows the chain rule of calculus, implemented in the `_compute_gradients` method:

Listing 11: Gradient Computation

```
def _compute_gradients(self, X, y):
    batch_size = X.shape[0]

    # Forward pass (as shown above)
    ...

    # Backward pass
    grad_y = probs.copy()
    grad_y[range(batch_size), y.argmax(axis=1)] -= 1
    grad_y /= batch_size

    # Last layer
    grad_w3 = a2.T @ grad_y
    grad_b3 = np.sum(grad_y, axis=0)
    grad_a2 = grad_y @ self.model.layers[4].weights.data.T

    # Second layer
    grad_h2 = grad_a2 * (h2 > 0)  # ReLU derivative
    grad_w2 = a1.T @ grad_h2
    grad_b2 = np.sum(grad_h2, axis=0)
    grad_a1 = grad_h2 @ self.model.layers[2].weights.data.T

    # First layer
    grad_h1 = grad_a1 * (h1 > 0)  # ReLU derivative
    grad_w1 = X.T @ grad_h1
    grad_b1 = np.sum(grad_h1, axis=0)

    return [(grad_w1, grad_b1), (grad_w2, grad_b2),
            (grad_w3, grad_b3)]
```

The backpropagation process computes gradients layer by layer, from output to input:

1. **Output Layer Gradients**:
   - Compute gradient of loss with respect to probabilities
   - Calculate gradients for weights and bias using the chain rule
2. **Hidden Layer Gradients**:
   - Propagate gradient through ReLU activation ($\frac{\partial L}{\partial h} = \frac{\partial L}{\partial a} \cdot \mathbb{1}_{h>0}$)
   - Compute weight and bias gradients
   - Propagate gradient to previous layer
3. **Input Layer Gradients**:
   - Similar process as hidden layer
   - Final gradients connect loss to input layer parameters

3.7 Parameter Updates

After computing gradients, the parameters are updated using gradient descent:

Listing 12: Parameter Updates

```
# Update weights
layer_indices = [0, 2, 4]  # Indices of Linear layers
for idx, (grad_w, grad_b) in zip(layer_indices, gradients):
   self.model.layers[idx].weights.data -= \
       self.learning_rate * grad_w
   self.model.layers[idx].bias.data -= \
       self.learning_rate * grad_b
```

The update rule follows the standard gradient descent formula:

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

where α is the learning rate and $\nabla_\theta \mathcal{L}$ represents the computed gradients.

3.8 Implementation Optimizations and Practical Considerations

3.8.1 Memory Management

Our implementation includes several memory optimization strategies:

Listing 13: Memory Management in Training

```
def zero_grad(self):
   """Clear gradients before each backward pass"""
   for p in self.parameters():
       if p.grad is not None:
           p.grad = np.zeros_like(p.data)
```

Key memory management features:

1. **Gradient Clearing**: Gradients are cleared before each backward pass to prevent accumulation:

```
model.zero_grad()  # Clear gradients before backward pass
```

2. **Intermediate Value Management**: Intermediate values are stored efficiently during the forward pass for use in backpropagation:

```
# Store activations for backward pass
h1 = X @ self.model.layers[0].weights.data + self.model.layers[0].bias.data
a1 = np.maximum(0, h1)  # ReLU activation stored
```

3. **Memory Reuse**: The implementation reuses memory where possible by updating existing arrays rather than creating new ones:

```
# Update weights in-place
self.model.layers[idx].weights.data -= self.learning_rate * grad_w
```

3.9 Performance Optimizations

Our implementation incorporates several crucial performance optimizations that balance computational efficiency with memory usage.

3.9.1 Vectorized Operations

We leverage NumPy's efficient array operations for computations across batches:

Listing 14: Vectorized Computations

```
def _compute_gradients(self, X, y):
    batch_size = X.shape[0]

    # Vectorized forward pass
    h1 = X @ self.model.layers[0].weights.data + self.model.layers[0].bias.data
    grad_h2 = grad_a2 * (h2 > 0)  # Vectorized ReLU gradient

    # Vectorized gradient computation
    grad_w1 = X.T @ grad_h1
    grad_b1 = np.sum(grad_h1, axis=0)
```

3.9.2 Numerical Stability

To ensure stable training, we implement several numerical safeguards:

Listing 15: Numerical Stability Implementations

```
def softmax(x):
    # Subtract max for numerical stability
    exp_x = np.exp(x - np.max(x, axis=1, keepdims=True))
    return exp_x / np.sum(exp_x, axis=1, keepdims=True)

def cross_entropy_loss(pred, target):
    # Add epsilon to prevent log(0)
    eps = 1e-12
    pred_clipped = np.clip(pred.shape[0], eps, 1 - eps)
    return -np.sum(target * np.log(pred_clipped)) / pred.shape[0]
```

These stability measures prevent common numerical issues:

1. Overflow in exponential computations
2. Underflow in logarithmic computations
3. Division by zero in normalization operations

3.10 Training Loop Optimization

The training process is optimized through efficient batch processing and progress monitoring:

Listing 16: Optimized Training Loop

```
def train_epoch(self, train_loader, desc="Training"):
    total_loss = 0
    correct = 0
    n_samples = 0

    for X, y in tqdm(train_loader, desc=desc):
        batch_size = X.shape[0]
        n_samples += batch_size
```

```
            # Efficient forward and backward pass
            outputs = self.model.forward(X)
            loss = self.criterion(outputs, y)

            # Gradient computation and update
            self.model.zero_grad()
            gradients = self._compute_gradients(X, y)
            self._update_parameters(gradients)

            # Track metrics
            total_loss += loss * batch_size
            predictions = np.argmax(outputs, axis=1)
            correct += np.sum(predictions == np.argmax(y, axis=1))

    return total_loss / n_samples, correct / n_samples
```

3.11 Final Considerations

The effectiveness of our implementation relies on several key factors:

1. **Batch Size Selection**:
   - Large enough for efficient matrix operations
   - Small enough to fit in memory
   - Typically 32 or 64 for MNIST training
2. **Learning Rate Tuning**:
   - Initial rate of 0.01 works well for this architecture
   - May need adjustment based on convergence behavior
   - Can be reduced during training for fine-tuning
3. **Gradient Management**:
   - Regular gradient clearing prevents accumulation errors
   - Efficient computation through vectorized operations
   - Proper handling of backpropagation through the network

These optimizations and considerations together create an efficient and stable implementation that serves both educational purposes and practical usage for the MNIST classification task.