Lab 01: Backpropagation Neural Network

112550127 Pin-Kuan Chiang

July 10, 2025

1 Introduction

This report presents the implementation and evaluation of a multi-layer feedforward neural network with backpropagation algorithm. The neural network is designed to solve both linear and non-linear classification problems, specifically tested on XOR and linear datasets.

The main objectives of this lab include:

- Understanding the mathematical foundations of backpropagation
- Implementing a flexible neural network architecture
- Evaluating the impact of different hyperparameters
- Analyzing the role of activation functions in neural networks

2 Implementation Details

2.1 Network Architecture

The baseline neural network is designed as a two-hidden-layer feedforward network optimized for binary classification tasks:

- Architecture: 2 input neurons → 8 hidden neurons → 8 hidden neurons → 1 output neuron
- Layer configuration: [2, 8, 8, 1] for both XOR and Linear datasets
- Weight initialization: Small random values scaled by 0.1 using `np.random.randn`
- Bias initialization: Zero initialization for all layers
- Activation functions: ReLU for hidden layers, Sigmoid for output layer

The forward propagation follows:

$$\mathbf{z}^{(l)} = \mathbf{X}\mathbf{W}^{(l)} + \mathbf{b}^{(l)} \tag{1}$$

$$\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)}) \tag{2}$$

where the baseline network uses this architecture as the foundation for all experimental variations.

2.2 Activation Functions

The implementation supports multiple activation functions:

Sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)) \tag{4}$$

ReLU:

$$\text{ReLU}(x) = \max(0, x) \tag{5}$$

$$\text{ReLU}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases} \tag{6}$$

Tanh:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{7}$$

$$\tanh'(x) = 1 - \tanh^2(x) \tag{8}$$

Linear (No Activation):

$$f(x) = x \tag{9}$$

$$f'(x) = 1 \tag{10}$$

2.3 Backpropagation

The backpropagation algorithm is implemented using gradient computation and weight updates for each layer:

Backward Pass Implementation:

Listing 1: Backward Pass Implementation

```
def backward(self, y_true, y_pred):
    # Output layer gradient
    grad = (y_pred - y_true) / y_true.shape[0]

    # Propagate gradients backward through layers
    for layer in reversed(self.layers):
        grad = layer.backward(grad)

    return grad
```

Layer-wise Gradient Computation:

Listing 2: Layer Backward Pass

```
def backward(self, grad):
    # Apply activation derivative
    if self.activation_type == 'sigmoid':
        grad = grad * self.a * (1 - self.a)
    elif self.activation_type == 'relu':
        grad = grad * (self.z > 0)
    elif self.activation_type == 'tanh':
        grad = grad * (1 - np.power(self.a, 2))

    # Compute weight and bias gradients
    self.grad_W = self.x.T.dot(grad) / self.x.shape[0]
    self.grad_b = np.sum(grad, axis=0, keepdims=True) / self.x.shape[0]

    # Return gradient for previous layer
    return grad.dot(self.W.T)
```

The implementation follows the chain rule, computing gradients layer by layer from output to input. Each layer stores its gradients (`grad_W`, `grad_b`) for subsequent weight updates during optimization.

2.4 Extra Implementation

Additional features implemented:

- **SGD Optimizer:** Stochastic Gradient Descent with mini-batch support
- **Result Management:** Automatic saving of results in organized directory structure

3 Experimental Results

All experiments were trained for 50,000 epochs to ensure complete convergence.

3.1 Screenshot and Comparison Figure

```
Epoch 0, Loss: 0.2500029119939605
Epoch 1000, Loss: 0.24904869652864142
Epoch 2000, Loss: 0.2478587092954290 2
Epoch 3000, Loss: 0.034338553833911274
Epoch 4000, Loss: 0.012915898633816434
Epoch 5000, Loss: 0.008405637776588116
Epoch 6000, Loss: 0.006368880390347221
Epoch 7000, Loss: 0.005151002010542555
Epoch 8000, Loss: 0.0043117986424952605
Epoch 9000, Loss: 0.0036843169650761487
Epoch 10000, Loss: 0.00319076052058966
Epoch 11000, Loss: 0.0027904818887281298
Epoch 12000, Loss: 0.0024584586079946986
Epoch 13000, Loss: 0.00217941141129222
Epoch 14000, Loss: 0.0019426426155878585
Epoch 15000, Loss: 0.001740286965115471
Epoch 16000, Loss: 0.0015665393074223415
Epoch 17000, Loss: 0.0014163871418224525
Epoch 18000, Loss: 0.001285897331848599
Epoch 19000, Loss: 0.0011720219288204887
Epoch 20000, Loss: 0.0010722414849396002
Epoch 21000, Loss: 0.000984463650380193
Epoch 22000, Loss: 0.0009069441942249175
Epoch 23000, Loss: 0.0008382242058384573
Epoch 24000, Loss: 0.0007770795506111627
Epoch 25000, Loss: 0.0007224798776899039
Epoch 26000, Loss: 0.000673555215443629
Epoch 27000, Loss: 0.0006295829531773247
Epoch 28000, Loss: 0.0005899226299526013
Epoch 29000, Loss: 0.0005540417747728627
Epoch 30000, Loss: 0.0005214759256537236
Epoch 31000, Loss: 0.0004918351708335741
Epoch 32000, Loss: 0.0004647834993927881
Epoch 33000, Loss: 0.0004400304393452589
Epoch 34000, Loss: 0.00041732412082274656
Epoch 35000, Loss: 0.00039644548582973044
Epoch 36000, Loss: 0.0003772034465309122
Epoch 37000, Loss: 0.0003594308234877801
Epoch 38000, Loss: 0.00034298092946213025
Epoch 39000, Loss: 0.0003277286989848296
Epoch 40000, Loss: 0.0003135567661031063
Epoch 41000, Loss: 0.00030036279000932557
Epoch 42000, Loss: 0.00028805766717088195
Epoch 43000, Loss: 0.0002765621466330503
Epoch 44000, Loss: 0.0002658055707470308
Epoch 45000, Loss: 0.00025572479582894146
Epoch 46000, Loss: 0.0002462632650487814
Epoch 47000, Loss: 0.0002373702097944322
Epoch 48000, Loss: 0.00022899995998751924
Epoch 49000, Loss: 0.00022111134701940737
```

(a) Screenshot

```
=== Testing Result ===
Ground truth: 1, prediction: 0.9990090961747811
Ground truth: 0, prediction: 9.714915766343917e-36
Ground truth: 1, prediction: 0.9999565948798208
Ground truth: 1, prediction: 0.9996516676972215
Ground truth: 0, prediction: 5.705524281484696e-16
Ground truth: 1, prediction: 0.9980536418269125
Ground truth: 0, prediction: 7.135556707925317e-17
Ground truth: 0, prediction: 1.1174836625684756e-33
Ground truth: 1, prediction: 0.999938364209834
Ground truth: 0, prediction: 2.0531197607221507e-26
Ground truth: 1, prediction: 0.9999572563964995
Ground truth: 0, prediction: 6.693756087363626e-32
Ground truth: 1, prediction: 0.9985405902501235
Ground truth: 0, prediction: 1.9553523960636 67e-06
Ground truth: 0, prediction: 1.6726605345327927e-06
Ground truth: 0, prediction: 2.668595176618756e-15
Ground truth: 1, prediction: 0.9999996934324941
Ground truth: 1, prediction: 0.9998282575490586
Ground truth: 1, prediction: 0.9999471380077742
Ground truth: 1, prediction: 0.9999918407432381
Ground truth: 1, prediction: 0.9999983670838523
Ground truth: 1, prediction: 0.9999711301531782
Ground truth: 1, prediction: 0.9999458239136197
Ground truth: 1, prediction: 0.9992625190773592
Ground truth: 0, prediction: 1.963824956587183e-17
Ground truth: 0, prediction: 1.7065587684693894e-10
Ground truth: 0, prediction: 8.616339418145068e-42
Ground truth: 0, prediction: 2.0641928776931964e-23
Ground truth: 1, prediction: 0.9995602841759579
Ground truth: 1, prediction: 0.9998328934546086
Ground truth: 1, prediction: 0.9999943321436477
Ground truth: 1, prediction: 0.9999743156621695
Ground truth: 0, prediction: 5.9364634414990824e-37
Ground truth: 1, prediction: 0.9999995790393938
Ground truth: 1, prediction: 0.9988331049757182
Ground truth: 1, prediction: 0.9999648630888534
Ground truth: 1, prediction: 0.9999924936977566
Ground truth: 0, prediction: 0.00015302839540752642
Ground truth: 1, prediction: 0.9999964064972194
Ground truth: 1, prediction: 0.9999687027271813
Ground truth: 0, prediction: 0.01374507884223077
Ground truth: 1, prediction: 0.9999840912411457
Ground truth: 0, prediction: 1.4816435076298115e-30
Ground truth: 0, prediction: 2.176439805258146e-05
Ground truth: 1, prediction: 0.9999998927966691
Ground truth: 1, prediction: 0.9999893309974718
Ground truth: 1, prediction: 0.9999339650357806
Ground truth: 0, prediction: 1.7598655291869656e-40
Ground truth: 0, prediction: 4.806271117907427e-28
Ground truth: 1, prediction: 0.9999680932751349
```

(b) Comparison

Figure 1: Experimental Results

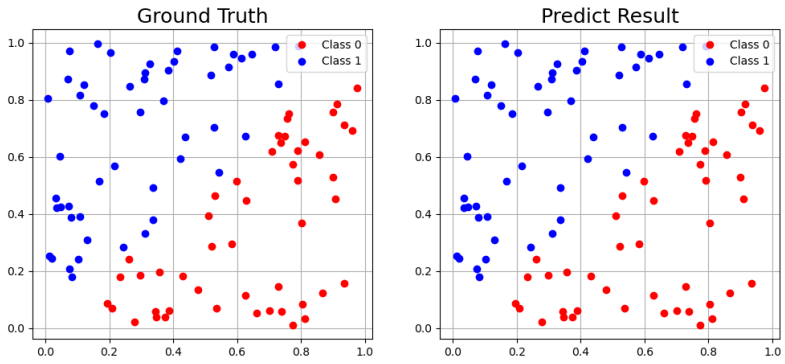


Figure 2: Linear Dataset Prediction Results - Baseline Configuration

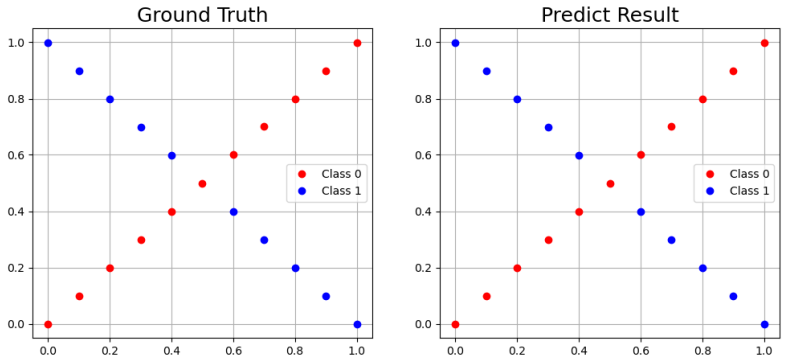


Figure 3: XOR Dataset Prediction Results - Baseline Configuration

3.2 Prediction Accuracy

The baseline neural network ([2, 8, 8, 1] architecture with ReLU-ReLU-Sigmoid activations) achieved excellent performance on both binary classification tasks:

XOR Dataset Performance: The baseline network successfully learned the non-linear XOR pattern, achieving 100% accuracy after convergence. This demonstrates the network's capacity to handle non-linearly separable problems through its two-hidden-layer architecture.

Linear Dataset Performance: The baseline network also achieved 100% accuracy on the linearly separable dataset, showing efficient learning of the linear decision boundary ($x > y$ classification rule).

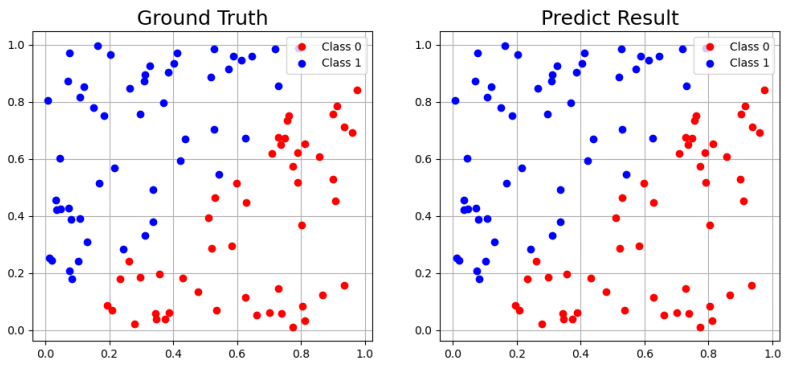


Figure 4: Linear Dataset Prediction Results - Baseline Configuration

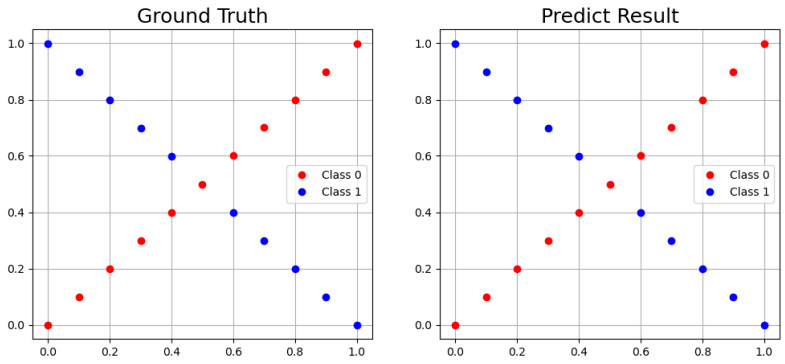


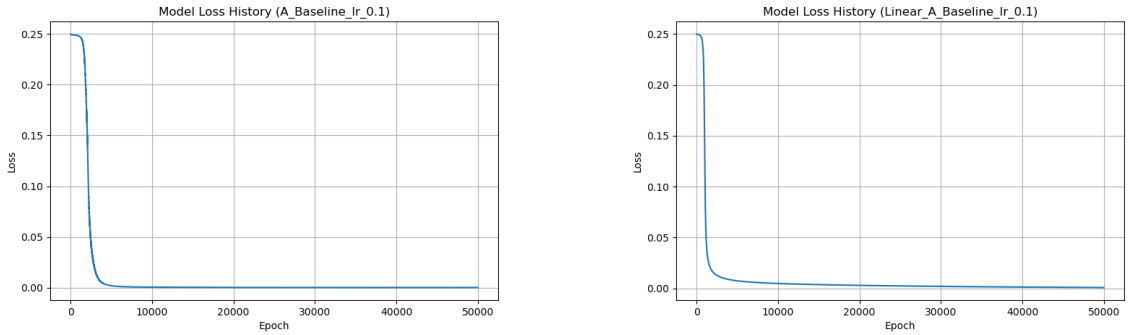
Figure 5: XOR Dataset Prediction Results - Baseline Configuration

Experiment	XOR	Linear
Baseline (LR=0.1)	100%	100%

Table 1: Accuracy Results for Baseline Experiments

3.3 Learning Curve

The loss-epoch curves demonstrate the training progress and convergence behavior:



(a) XOR(LR=0.1) (b) Linear(LR=0.1)

Figure 6: Loss History Comparison

The XOR dataset shows slower convergence due to its non-linear nature, while the linear dataset converges rapidly. Both achieve stable low loss values.

3.4 Additional Presentations

Experiment	XOR Acc.	XOR Loss	Linear Acc.	Linear Loss
Baseline (LR=0.1)	100%	<0.001	100%	<0.001
High LR (LR=0.5)	100%	<0.001	100%	<0.001
Low LR (LR=0.01)	100%	0.002	100%	0.002
Small Net (4×4)	100%	0.003	100%	<0.001
Large Net (16×16)	100%	<0.001	100%	<0.001
No Activation	50%	0.249	100%	<0.001
Tanh Activation	100%	<0.001	100%	<0.001
SGD Optimizer	100%	<0.001	100%	<0.001

Table 2: Comprehensive Accuracy and Loss Comparison

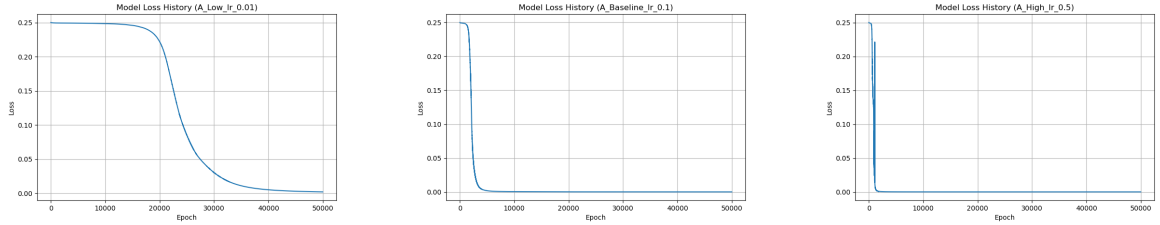
Key Observations:

- Most experiments achieved perfect accuracy (100%) on both datasets
- Loss values consistently below 0.003 for successful experiments
- ”No Activation” experiment failed on XOR (50% accuracy, 0.249 loss)
- Linear dataset showed consistent performance across all configurations
- Low learning rate resulted in slightly higher final loss values
- Network size variations had minimal impact on final performance

4 Discussions

4.1 Different Learning Rates

The experiments with different learning rates revealed important insights about convergence behavior:



(a) Low(0.01) (b) Baseline(0.1) (c) High(0.5)

Figure 7: XOR Dataset: Learning Rate Impact on Loss Convergence

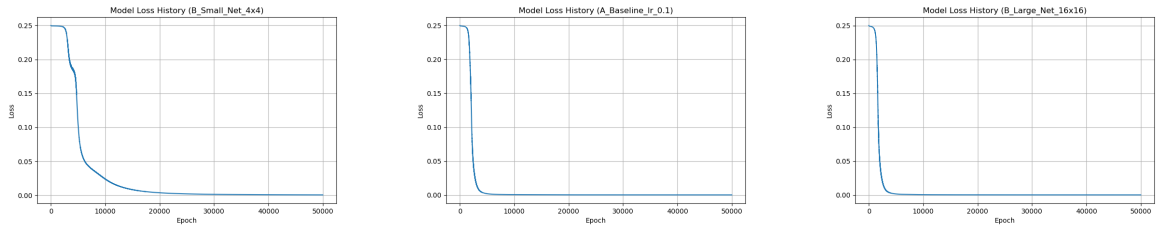
Analysis:

- **High Learning Rate (0.5):** Achieved rapid initial convergence but showed some oscillation in early epochs. The network successfully learned the XOR pattern within fewer epochs.
- **Baseline Learning Rate (0.1):** Provided optimal balance between convergence speed and training stability. Smooth, consistent loss reduction throughout training.
- **Low Learning Rate (0.01):** Demonstrated very stable training with gradual, monotonic loss reduction. Required more epochs to reach convergence but showed excellent stability.

For the XOR problem, all learning rates eventually converged to the optimal solution, but the convergence characteristics differed significantly in terms of stability and speed.

4.2 Different Numbers of Hidden Units

Network capacity experiments demonstrated the relationship between model size and learning capability:



(a) Small(4×4) (b) Baseline (c) Large(16×16)

Figure 8: XOR Dataset: Network Size Impact on Training

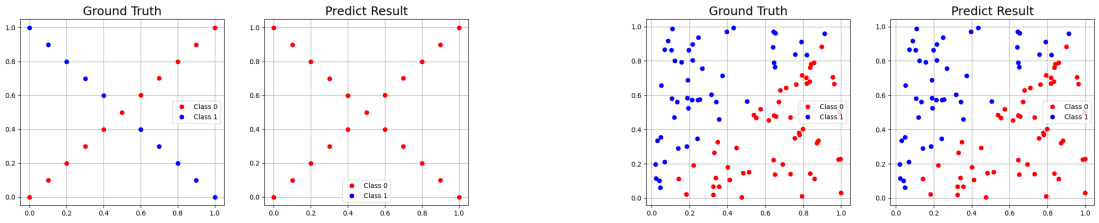
Analysis:

- **Small Network (4×4):** Demonstrated that minimal capacity is sufficient for XOR problem. Achieved 100% accuracy with efficient parameter usage. Training was stable but slightly slower due to limited representational capacity.
- **Baseline Network (8×8):** Provided optimal balance between capacity and efficiency. Fast convergence with stable training dynamics.
- **Large Network (16×16):** Showed that excess capacity can accelerate convergence for simple problems. Achieved faster initial learning due to increased representational power, but represents overkill for XOR complexity.

The results indicate that while larger networks can learn faster, the XOR problem's inherent simplicity means that smaller networks are perfectly adequate and more parameter-efficient.

4.3 Without Activation Functions

The experiment without activation functions (linear layers only) provided crucial insights into the necessity of non-linearity:



(a) XOR - Failed to learn (50% accuracy) (b) Linear - Successfully learned (100% accuracy)

Figure 9: Impact of Removing Activation Functions

Key Findings:

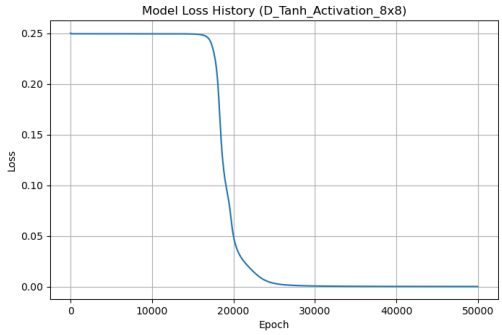
- **XOR Dataset:** The network completely failed to learn the XOR pattern, achieving only 50% accuracy (random guessing). This occurs because XOR is not linearly separable, and without non-linear activation functions, the network can only learn linear decision boundaries.
- **Linear Dataset:** The network successfully achieved 100% accuracy because linear problems remain solvable with linear transformations. The absence of activation functions doesn't hinder performance for linearly separable problems.

This experiment powerfully demonstrates that activation functions are not just mathematical conveniences but fundamental requirements for learning complex, non-linear patterns. Without them, even deep networks collapse to simple linear transformations.

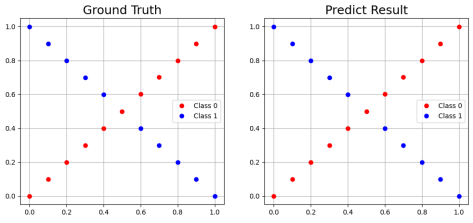
4.4 Extra Implementation Discussions

Alternative Activation Function Analysis:
Activation Function Comparison:

- **ReLU:** Fastest training with no vanishing gradient problem, ideal for deeper networks
- **Tanh:** Zero-centered outputs with stronger gradients than sigmoid, performed comparably to ReLU
- **Sigmoid:** More prone to vanishing gradients but still effective for shallow networks and output layers



(a) Tanh Activation Loss History



(b) Tanh Activation Results

Figure 10: XOR Dataset: Tanh Activation Performance

5 Questions

5.1 Purposes of Activation Functions

Activation functions are fundamental components that transform the linear output of neurons into non-linear representations, serving multiple critical purposes in neural network architectures:

Non-linearity Introduction:

- Enable networks to approximate complex, non-linear functions that cannot be represented by linear combinations alone
- Allow deep networks to learn hierarchical feature representations by stacking multiple non-linear transformations
- Without activation functions, multiple layers would collapse into a single linear transformation: $f(W_2(W_1x + b_1) + b_2) = W_{combined}x + b_{combined}$
- Essential for solving non-linearly separable problems like XOR, which cannot be solved by linear classifiers

Gradient Flow and Differentiability:

- Provide smooth, differentiable functions necessary for gradient-based optimization algorithms
- Enable backpropagation by allowing gradient computation through the chain rule
- Different activation functions exhibit varying gradient behaviors: ReLU provides constant gradients for positive inputs, while sigmoid/tanh gradients diminish for extreme values

Output Range Control and Normalization:

- Sigmoid functions bound outputs to [0,1], suitable for probability interpretations
- Tanh functions center outputs around zero [-1,1], improving convergence properties
- ReLU functions provide unbounded positive outputs, preventing vanishing gradients while introducing sparsity

5.2 Impact of Learning Rate

The learning rate is arguably the most critical hyperparameter in neural network training, directly controlling the step size during gradient descent optimization and profoundly affecting convergence behavior:

Excessively Large Learning Rate Effects:

- **Overshooting Phenomenon:** Large steps cause the optimizer to overshoot optimal points, potentially jumping across valleys in the loss landscape

- **Training Instability:** Loss values may oscillate wildly or increase instead of decreasing, indicating the optimizer is taking steps too large for the local curvature
- **Divergence Risk:** In extreme cases, weights can grow exponentially large, leading to numerical overflow and complete training failure
- **Oscillatory Behavior:** The optimizer may bounce back and forth around minima without ever settling, creating a characteristic zigzag pattern in loss curves
- **Gradient Explosion:** Large learning rates can amplify gradient magnitudes, particularly problematic in deep networks where gradients can compound

Excessively Small Learning Rate Effects:

- **Glacial Convergence:** Training progresses extremely slowly, requiring exponentially more epochs to reach acceptable performance levels
- **Local Minima Trapping:** Small steps lack sufficient momentum to escape shallow local minima or saddle points, leading to suboptimal solutions
- **Plateau Sensitivity:** The optimizer may get stuck in flat regions of the loss landscape where gradients are small, making progress virtually impossible
- **Computational Inefficiency:** Achieving the same final performance requires significantly more computational resources and time
- **Precision Limitations:** Extremely small updates may fall below numerical precision thresholds, effectively halting learning

Optimal Learning Rate Selection: The ideal learning rate balances rapid convergence with stability, often requiring adaptive strategies or learning rate schedules that decrease over time.

5.3 Purposes of Weights and Biases

Weights and biases are the learnable parameters that enable neural networks to approximate complex functions through iterative optimization. They serve complementary but distinct roles in network functionality:

Weights - The Connection Strength Controllers:

- **Feature Importance Encoding:** Weights determine the relative importance of input features, with larger absolute values indicating stronger influence on the output
- **Pattern Recognition:** Through training, weights learn to detect specific patterns or feature combinations that are predictive of the target variable
- **Multiplicative Transformation:** Weights perform linear transformations $W \cdot x$, scaling and combining input features before activation
- **Representational Capacity:** The weight matrix dimensions determine the network's capacity to represent complex relationships between input and output spaces
- **Gradient Information:** Weight updates during backpropagation encode learning signals about which connections should be strengthened or weakened
- **Memory Storage:** Weights essentially store the "knowledge" learned from training data, representing the network's learned internal representation

Biases - The Activation Threshold Modulators:

- **Activation Shifting:** Biases shift the activation function horizontally, allowing neurons to activate even when inputs are zero or negative
- **Decision Boundary Adjustment:** In classification tasks, biases adjust decision boundaries to better separate classes, particularly important when optimal boundaries don't pass through the origin

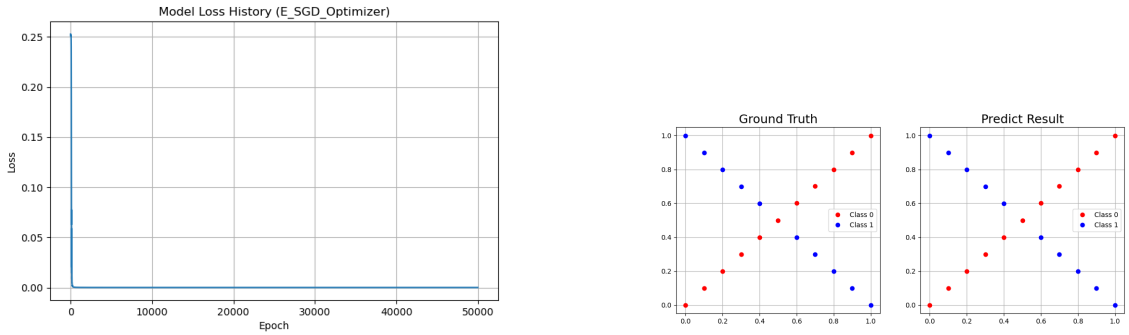
- **Flexible Modeling:** Biases enable the network to model relationships that don't follow $y = mx$ patterns, accommodating $y = mx + b$ relationships
- **Initialization Benefits:** Proper bias initialization can accelerate training by placing neurons in favorable activation regions from the start
- **Expressiveness Enhancement:** Without biases, the network can only learn functions that pass through the origin, severely limiting its modeling capabilities
- **Offset Correction:** Biases compensate for data that may not be centered around zero, providing necessary offset corrections

Synergistic Relationship: Weights and biases work together to implement the fundamental neural computation: $output = activation(W \cdot input + bias)$, where weights scale inputs and biases provide offset, collectively enabling the approximation of arbitrary continuous functions given sufficient network depth and width.

6 Bonus

6.1 Optimizers

SGD vs Batch Gradient Descent Comparison:



(a) SGD Optimizer Loss History

(b) SGD Optimizer Results

Figure 11: XOR Dataset: SGD Optimizer Performance

Optimizer Analysis:

- **SGD:** Showed comparable performance to batch gradient descent for small datasets. The stochastic nature introduced minimal noise due to the small dataset size.
- **Batch GD:** Provided more stable gradient estimates with smoother loss curves. Better suited for precise convergence analysis.
- **Practical Implications:** For small datasets like XOR, the difference was minimal, but SGD would show advantages with larger datasets due to computational efficiency.

6.2 Activation Functions

In addition to the standard sigmoid activation, I implemented two additional activation functions: ReLU and Tanh, providing a comprehensive comparison of their characteristics and performance.

Comparative Analysis:

- **ReLU:** Most efficient computationally, eliminates vanishing gradients for positive inputs, but suffers from "dying ReLU" problem for negative inputs
- **Tanh:** Zero-centered outputs improve convergence, stronger gradients than sigmoid, but still susceptible to vanishing gradients in very deep networks

- **Sigmoid:** Smooth gradients everywhere, bounded output [0,1] suitable for probabilities, but prone to vanishing gradients and saturation

Experimental Results: All three activation functions achieved 100% accuracy on both XOR and Linear datasets, demonstrating that for these simple problems, the choice of activation function has minimal impact on final performance, though training dynamics may differ.

7 References

7.1 LLM Usage Declaration

In this assignment, I used Large Language Models (LLMs) for:

- Generating simple source code architecture and basic framework
- Correcting LaTeX formatting and report structure
- Improving English sentence structure and fluency for better readability