Deep Learning for Imaging (EE5179): Assignment 1

Siddhant Erande - ED22B048

September 11, 2025

1 Introduction

The Multi-Layer Perceptron (MLP) is a fundamental feedforward artificial neural network consisting of multiple layers of nodes with nonlinear activation functions. This study implements and analyzes MLP networks for the classic problem of handwritten digit recognition using the MNIST dataset.

The MNIST dataset consists of 60,000 training images and 10,000 test images of handwritten digits (0-9), each of size 28×28 pixels. The objective is to classify these images into their respective digit classes using MLP architectures.

Dataset Statistics:

- Training samples: 60,000
- Test samples: 10,000
- Image dimensions: 28×28 (flattened to 784 features)
- Number of classes: 10

2 Methodology

2.1 Model Architecture

The MLP architecture used throughout this study consists of:

- **Input layer:** 784 neurons (flattened 28×28 images)
- **Hidden layers:** Three hidden layers with 500, 250, and 100 neurons respectively
- **Output layer:** 10 neurons (one for each digit class)
- **Weight initialization:** Glorot uniform initialization
- **Bias initialization:** Zero initialization

2.2 Mathematical Formulation

For each layer i, the forward pass is computed as:

$$\mathbf{z}_i = \mathbf{W}_i \mathbf{a}_{i-1} + \mathbf{b}_i \tag{1}$$

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

Where $\mathbf{W}_i$ and $\mathbf{b}_i$ are the weights and biases for layer i, $\mathbf{a}_i$ is the activation output of layer i, and $f(\cdot)$ is the activation function. The final layer uses softmax activation:

$$\text{softmax}(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{3}$$

The loss function used is categorical cross-entropy:

$$L = -\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{K} y_{ij} \log(\hat{y}_{ij}) \tag{4}$$

3 From-Scratch Implementation

3.1 Implementation Details

The MLP was implemented from scratch using NumPy with the following key components:

```python
class MLP(nn.Module):
    def __init__(self, input_dim=784, hidden_dims=[500, 250, 100], output_dim
    =10):
        layer_sizes = [input_dim] + hidden_dims + [output_dim]

        # Glorot initialization
        self.weights = []
        self.biases = []
        for i in range(len(layer_sizes) - 1):
            ip_layer_size = layer_sizes[i]
            op_layer_size = layer_sizes[i + 1]
            limit = np.sqrt(6 / (ip_layer_size + op_layer_size))
            W = np.random.uniform(-limit, limit,
                                  size=(ip_layer_size, op_layer_size))
            b = np.zeros((1, op_layer_size))
            self.weights.append(W)
            self.biases.append(b)
```

Listing 1: MLP Class Initialization with Glorot Weight Initialization

3.2 Activation Functions

Three activation functions were implemented and compared:

3.2.1 Sigmoid Activation

```python
def sigmoid(self, x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(self, x):
    return x * (1 - x)
```

Listing 2: Sigmoid Activation Implementation

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

Results:

- Test Accuracy: 82.30%
- Training Loss (15 epochs): 0.6783

3.2.2 ReLU Activation

```python
def relu(self, x):
    return np.maximum(0, x)

def relu_derivative(self, x):
    return (x > 0).astype(float)
```

Listing 3: ReLU Activation Implementation

Mathematical form: $\mathrm{ReLU}(x) = \max(0, x)$

Results:

- Test Accuracy: 97.12%
- Training Loss (15 epochs): 0.0753

3.2.3 Tanh Activation

```python
def tanh(self, x):
    return np.tanh(x)

def tanh_derivative(self, x):
    return 1.0 - np.tanh(x)**2
```

Listing 4: Tanh Activation Implementation

Mathematical form: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

Results:

- Test Accuracy: 95.47%
- Training Loss (15 epochs): 0.1501

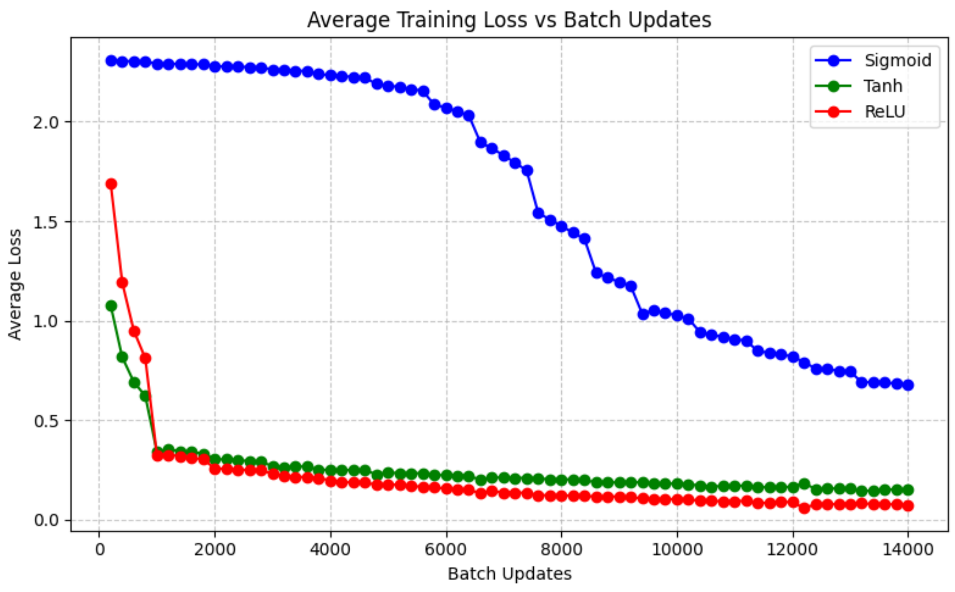

Figure 1: Training loss comparison for different activation functions

3.3 Learning Rate Analysis

The impact of different learning rates was analyzed using the sigmoid activation function:

Table 1: Learning Rate Impact on Sigmoid Activation Performance

Learning Rate	Test Accuracy (%)	Convergence Behavior
0.001	11.56	Too slow, insufficient learning
0.01	82.30	Moderate, stable convergence
0.1	94.25	Fast, some instability

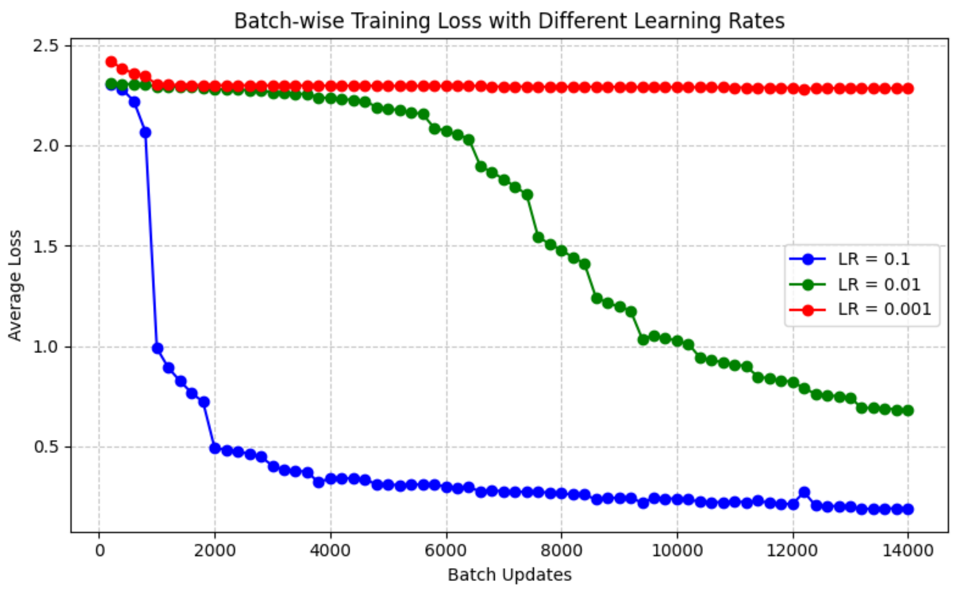

Figure 2: Training loss curves for different learning rates

3.4 Training Duration Analysis

The effect of training duration was studied with sigmoid activation:

Table 2: Impact of Training Epochs on Model Performance

Epochs	Test Accuracy (%)	Final Training Loss
10	69.76	1.0842
15	82.30	0.6783
20	85.39	0.5247
25	87.75	0.4352

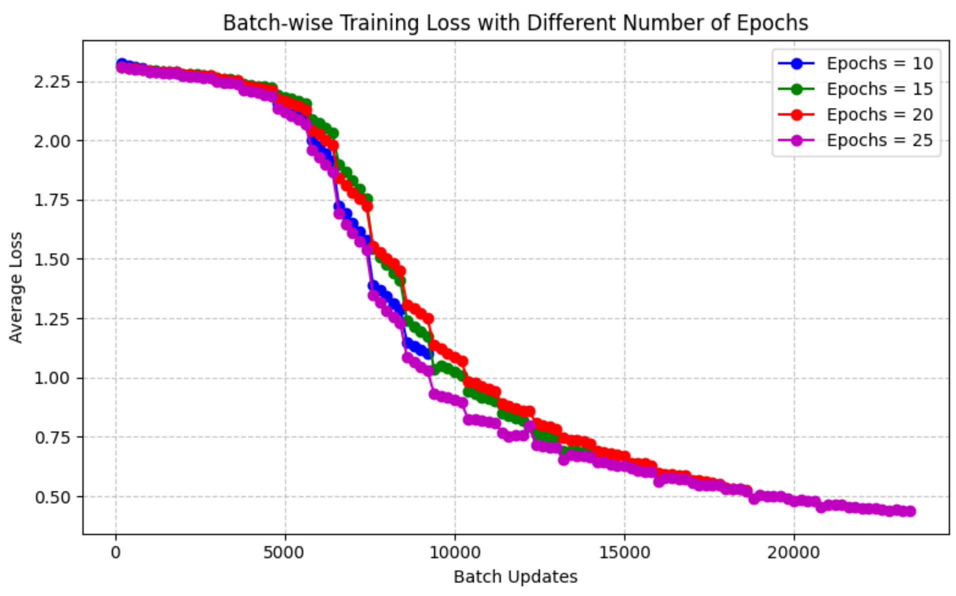

Figure 3: Training loss curves for different number of epochs

4 PyTorch Implementation

4.1 Model Implementation

The PyTorch implementation leveraged automatic differentiation and optimized operations:

```
class MLP(nn.Module):
    def __init__(self, input_dim=784, hidden_dim1=500,
                 hidden_dim2=250, hidden_dim3=100, output_dim=10):
        super(MLP, self).__init__()
        self.layer1 = nn.Linear(input_dim, hidden_dim1)
        self.layer2 = nn.Linear(hidden_dim1, hidden_dim2)
        self.layer3 = nn.Linear(hidden_dim2, hidden_dim3)
        self.output_layer = nn.Linear(hidden_dim3, output_dim)
        self.ReLu = nn.ReLU()

    def forward(self, x):
        out_1 = self.ReLu(self.layer1(x))
        out_2 = self.ReLu(self.layer2(out_1))
        out_3 = self.ReLu(self.layer3(out_2))
        out = self.output_layer(out_3)
        return out
```

Listing 5: PyTorch MLP Implementation

4.2 L2 Regularization Analysis

The effect of L2 regularization (weight decay) was examined using different values of the regularization parameter α:

```
# No regularization
optimizer = optim.Adam(model.parameters(), lr=0.01, weight_decay=0.0)

# Light regularization
optimizer = optim.Adam(model.parameters(), lr=0.01, weight_decay=1e-4)

# Heavy regularization
optimizer = optim.Adam(model.parameters(), lr=0.01, weight_decay=0.01)
```

Listing 6: Optimizer Configuration with Weight Decay

Table 3: L2 Regularization Impact on Model Performance

Weight Decay (α)	Test Accuracy (%)	Training Loss	Regularization Effect
0.0	97.12	0.0728	No regularization
0.0001	95.94	0.1324	Balanced performance
0.01	91.65	0.3140	Over-regularized

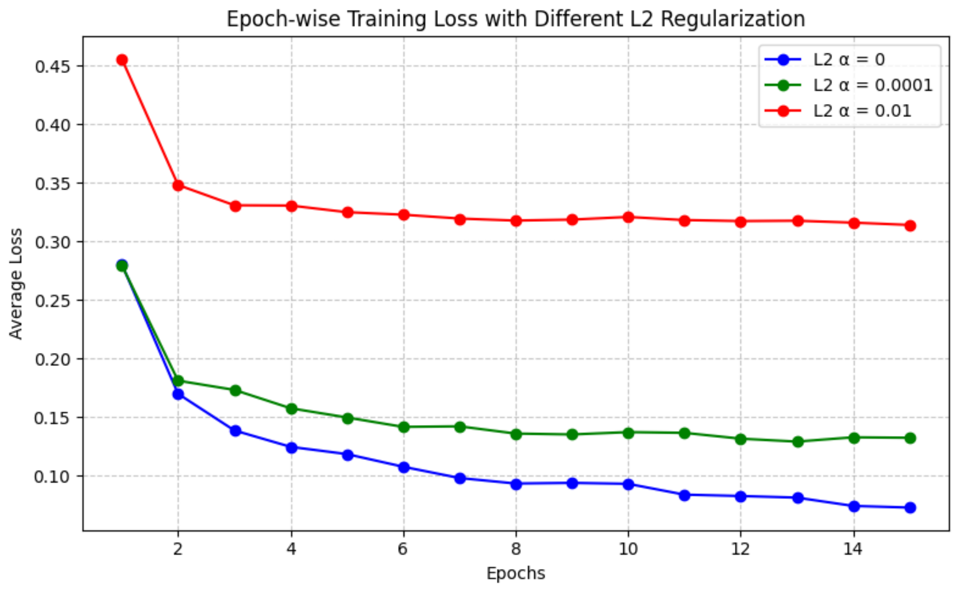

Figure 4: Training loss curves for different L2 regularization strengths

5 Results and Analysis

5.1 Activation Function Performance

Table 4: Comprehensive Comparison of Activation Functions

Activation	Test Accuracy (%)	Training Loss	Key Characteristics
Sigmoid	82.30	0.6783	Vanishing gradients, slow convergence, bounded output
ReLU	97.12	0.0753	Fast convergence, best performance, sparse activation
Tanh	95.47	0.1501	Good performance, zero-centered, smoother than ReLU

The superior performance of ReLU can be attributed to:

- Absence of vanishing gradient problem

- Sparse activation leading to efficient computation
- Better gradient flow during backpropagation
- Computational simplicity

6 Discussion

6.1 Key Findings

1. **Activation Function Superiority:** ReLU significantly outperformed sigmoid and tanh activations, achieving 97.12% vs 82.30% and 95.47% respectively. This demonstrates the importance of activation function choice in deep learning.
2. **Learning Rate Sensitivity:** The sigmoid activation function showed high sensitivity to learning rate, with performance improving dramatically from 82.30% to 94.25% when increasing the learning rate from 0.01 to 0.1.
3. **Regularization Trade-offs:** L2 regularization effectively reduced overfitting but at the cost of slightly lower test accuracy. The optimal weight decay factor was approximately 0.0001.
4. **Training Duration:** Extended training beyond 15-20 epochs showed diminishing returns, suggesting efficient convergence with the chosen architecture and hyperparameters.

7 Conclusion

The key conclusions are:

1. ReLU activation functions significantly outperform traditional sigmoid and tanh functions
2. Proper learning rate selection is crucial for optimal convergence
3. Light regularization provides the best balance between performance and generalization
4. Both implementation approaches (from-scratch and framework-based) achieved comparable results, validating the mathematical correctness

References

- https://elcaiseri.medium.com/building-a-multi-layer-perceptron-from-scratch-with-numpy-e4cee82ab06d
- https://medium.com/@hirok4/building-a-multi-layer-perceptron-from-scratch-c9679752cf48
- https://www.projectpro.io/article/exploring-mnist-dataset-using-pytorch-to-train-an-mlp/408
- https://www.geeksforgeeks.org/deep-learning/tanh-vs-sigmoid-vs-relu/
- https://www.aitude.com/comparison-of-sigmoid-tanh-and-relu-activation-functions/
- https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/