Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

*Digital Object Identifier 10.1109/ACCESS.2023.0322000*

# Perceptron Training for Handwritten Digit Classification using MNIST Dataset

**KHADKA PILOT[1*] AND POUDEL KSHITIZ[2*]**

[1]Institute of Engineering, Thapathali Campus, Thapathali, Nepal (e-mail: pilot.076bct025@tcioe.edu.np)
[2]Institute of Engineering, Thapathali Campus, Thapathali, Nepal (e-mail: kshitizpoudel18@gmail.com)

Corresponding author: Khadka Pilot (e-mail: pilot.076bct025@tcioe.edu.np).

* Authors contributed equally

**ABSTRACT** Artificial Neural Network are computational system inspired from human brain to learn human like abilities. By harnessing the power of the concepts we can design any learning systems advanced weight initialization techniques like Xavier random and He initialization, we elevate recognition precision. Moreover, we systematically investigate diverse neural network architectures, manipulating neuron quantities and hidden layers with the MNIST dataset. This empirical exploration highlights the significant influence of architecture on accuracy and efficiency. Additionally, our study delves into the impact of various activation functions on model performance, aiming to optimize results.

**INDEX TERMS** ANN, MNIST dataset

## I. INTRODUCTION

AN Artificial Neural Network (ANN) is a computational model inspired by the structure and functioning of the human brain's interconnected neurons. It's widely used in machine learning for tasks like pattern recognition, classification, regression, and more. The basic building block of an ANN is a neuron, which takes in inputs, performs computations, and produces an output. This project focuses on understanding artificial neural networks, which are important tools in computing and learning. These networks can find complex patterns in data. The main goal is to grasp how they work internally, including their forward and backward processes. These processes help improve the networks over time.

In today's machine learning world, neural networks are crucial. They're used in tasks like recognizing images and understanding language. This project aims to show how neural networks work, especially how they process data and get better with practice. We'll change settings, use different functions, and start from different points to see how they affect network performance. Knowing about neural networks is valuable as our world relies on data. Businesses and schools use data to learn things, and understanding neural networks gives us an important skill. This project wants to uncover how neural networks turn raw data into smart predictions. By learning theory and doing experiments, we can add to what people already know about these networks. In this experiment we will build a neural network from scratch by mathematically formulating the process and then experiment with different hyperparameters to observe their effect on final result. It will also help us to understand the mechanism of ANNs in more detail.The main goal is to make neural networks easier to understand and get ideas to make better models and choices.

## II. METHODOLOGY

### A. THE NEURAL SYSTEM: RECEPTORS, NEURONS, AND EFFECTORS

The human nervous system can be conceptualized as a triad of components: Receptors, Neuronal Network, and Effectors. The journey begins with Receptors, which transmute both external and internal stimuli into electrical impulses, signaling towards the central hub of the nervous system, the brain. Here, the Neural Network comes into play, adeptly processing and decoding the received information, subsequently formulating an appropriate response. Finally, the Effectors translates the brain's electrical instructions into tangible output responses.

Central to this system are neurons, the building blocks of the brain. These neurons interact through specialized units known as synapses, which act as bridges between them. Notably, Chemical synapses, the most prevalent type, convert electrical signals into chemical messengers and then reconvert them back into electrical impulses.

The neurons themselves encode their outputs in the form of voltage pulses, famously termed "action potentials," which traverse the neuron's anatomy with constant velocity and amplitude.

### B. ANATOMY OF A NEURON

At the core of neuronal operation lie three fundamental components:

- Synaptic Assemblies: These intricate ensembles represents an array of synapses, where each synapse exhibits a quantifiable weight or synaptic strength. The input signal xi, arriving at synapse j and connected to neuron k transforms via multiplication by the corresponding weight wkj.
- Summative Integration: The adder serves as an essential component responsible for the aggregation of input signals, each multiplicatively weighted by the corresponding input signals.
- Activation Function: It provides gating mechanism in regulating the neuron's output.The neuron's behavioral output is governed by an activation function, a nonlinear operator that limits the magnitude of the response. By introducing a thresholding effect on the cumulative input, it controls response to varying levels of input stimulation.

### C. FORWARD PROPAGATION

Forward propagation involves calculating the output for given inputs. For the first pass, the weights within the network can be zero, but it is not a very good idea. Therefore, there are many initialization techniques, the most basic being random initialization where every weight from one neuron in layer $L$ to another neuron in layer $L+1$ is randomly chosen from between $-1$ to 1 or another range of values.

After the weights are initialized, we need to compute the weighted sum coming to each neuron in the first hidden layer. Each neuron in the hidden layer has a weight associated with every neuron in the input layer. The forward propagation process is done in batches, where each batch consists of a number of training/test instances. Therefore, it is easier and faster to compute the weighted sum in matrix form as:

$$Z^{(1)} = W^{(1)T} \cdot A^{(0)} + B^{(1)} \tag{1}$$

Where,

Z(̂n) represents the weighted sum going to the nth layer

A(̂n) represents the activation of the nth layer the 0th layer corresponds to the input layer

W(̂n)T represents the weight matrix, where the jth column represents the weights for the jth neuron in the hidden layer

B(̂n) represents the bias of the nth layer

The weighted sum of the activation of the previous layer and the weights is then given as input to an activation function:

$$A^{(1)} = \text{activation}\{Z^{(1)}\} \tag{2}$$

The forward propagation continues until we obtain the activations for the output layer. Those activations represent the probabilities of the given instances belonging to each of the 10 classes. In other words, if $n$ hidden layers are used, then the output vector is given by:

$$\text{Output vector} = \text{Activation}(Z^{(n)}) = \text{softmax}(W^{(n)T} \cdot A^{(n-1)} + B^{(n)}) \tag{3}$$

### D. BACKWARD PROPAGATION

After the forward pass is made, we obtain predictions for the classes. Now, the weights need to be updated for the learning process to occur. The goal of learning is to minimize the difference between the predicted output and the actual class label. Thus, a loss function is required to measure this difference.

Mean Squared Error (MSE) loss works well when there are multiple classes of output labels. Our goal is to find the combination of parameters (weights + bias) that minimizes the MSE loss. The MSE loss is given as:

$$MSE = \frac{1}{2n} \sum_{i=1}^{n} (Y_i - \hat{Y}_i)^2$$

Where, $Y_i$ is the ground truth and $\hat{Y}_i$ is the predicted value, and $n$ is the total number of training examples.

Now we need to calculate the derivative of the loss function with respect to every weight. This is known as the gradient. Moving opposite to the direction of the gradient will result in a local minimum or global minimum. A local minimum is the point at which the loss is the lowest. It defines the combination of parameters for which the loss is the lowest.

$$\frac{\partial E}{\partial w_{ij}^{(k)}} = \frac{\partial E}{\partial a_j^{(k)}} \cdot \frac{\partial a_j^{(k)}}{\partial w_{ij}^{(k)}} \tag{4}$$

$$\frac{\partial a_j^{(k)}}{\partial w_{ij}^{(k)}} = \frac{\partial}{\partial w_{ij}^{(k)}} \left( \sum_{i'=0}^{r_{k-1}-1} w_{lj}^{(k)} o_{i'}^{(k-1)} \right) = o_i^{(k-1)} \tag{5}$$

Therefore,

$$\frac{\partial E}{\partial w_{ij}^{(k)}} = \frac{\partial E}{\partial a_j^{(k)}} o_i^{(k-1)} \tag{6}$$

## III. ALGORITHM

**Algorithm 1** Perceptron Training using Training Data

**Require:** Training Data: $(\mathbf{x}_i, y_i); \forall i \in \{1, 2, \ldots, N\}$, Learning Rate: $\eta$
**Ensure:** Separating Hyper-plane coefficients: $\mathbf{w}^*$

1: Initialize $\mathbf{w} \leftarrow \mathbf{0}$
2: **repeat**
3: Get example $(\mathbf{x}_i, y_i)$
4: $\hat{y}_i \leftarrow \mathbf{w}^T \mathbf{x}_i$
5: **if** $\hat{y}_i y_i \leq 0$ **then**
6: $\mathbf{w} \leftarrow \mathbf{w} + \eta y_i \mathbf{x}_i$
7: **end if**
8: **until** convergence

**Algorithm 2** Gradient Descent for Training a Linear Unit

**Require:** Training Data: $(\mathbf{x}_i, y_i); \forall i \in \{1, 2, \ldots, N\}$, Learning Rate: $\eta$

**Ensure:** Optimal Hyper-plane coefficients based on squared loss: $\mathbf{w}^*$

1: Initialize $\mathbf{w} \leftarrow$ random weights
2: **repeat**
3: Calculate $\nabla E$
4: $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E$
5: **until** convergence

### A. SYSTEM BLOCK DIAGRAM

Appendix

### B. INSTRUMENTATION

Here are some key NumPy functions utilized in our neural network implementation:

- **np.random**: This function is employed to generate random values sampled from a uniform or normal distribution. In our context, it's utilized for initializing weights and biases with random values.
- **np.dot**: The `np.dot` function performs matrix multiplication between the weight matrix and input matrix. It's a fundamental operation in neural network feedforward computations.
- **np.sum**: In various instances, `np.sum` is used to compute the sum over derivatives or values within specific layers of the network. It plays a role in backpropagation and loss calculation.
- **np.exp**: The `np.exp` function is employed for softmax calculations, which are crucial for obtaining probability distributions over multiple classes.
- **np.arange**: `np.arange` is used for creating a one-hot encoded representation of the labels. It helps in converting categorical labels into a binary format suitable for neural network training.
- **np.mean**: This function is used to calculate the mean value, often across all samples in a dataset. It might be employed, for instance, to calculate the mean loss over a batch of training examples.
- **np.argmax**: The `np.argmax` function helps in obtaining the index of the maximum value in a column. It's frequently used when making predictions or evaluating model outputs.
- **np.sum**: Another use of `np.sum` is to calculate the sum of values that match a specific target. This operation is valuable when computing the count of correct predictions in classification tasks.
- **plt.plot**: The `plt.plot` function from the `matplotlib.pyplot` library is employed for visualizing the accuracy and losses of the neural network during training. It aids in understanding the learning progress and potential issues.

## IV. WORKING PRINCIPLE

### A. DATASET OVERVIEW

The MNIST dataset, which stands for the Modified National Institute of Standards and Technology dataset, is a widely used benchmark in machine learning and computer vision. It consists of handwritten digit images along with corresponding labels, making it suitable for tasks like image classification and digit recognition. The MNIST dataset has played a significant role in advancing these fields and bench-marking various algorithms and models.

The MNIST dataset comprises a total of 70,000 gray-scale images depicting handwritten digits. Typically, this dataset is segregated into a training set, encompassing 60,000 images, and a distinct testing set, comprising 10,000 images. However, for our specific utilization, we've opted to work with a subset of the MNIST dataset, containing 42,000 instances.

### B. DATASET CONTENTS

#### 1) Images as csv

The images in the MNIST dataset are single-channel (grayscale) images. Each pixel's intensity value ranges from 0 to 255, representing the darkness of the pixel. The images contain handwritten digits from 0 to 9.

Each image is associated with a label corresponding to the digit it represents. The labels are integers ranging from 0 to 9, indicating the actual value of the digit in the image.

### C. DATASET PREPARATION

Preparing the MNIST dataset for machine learning involves several key steps:

#### 1) Data Loading Process

The data loading process involved utilizing the 'read_csv' function to import the CSV file. Within this dataset, there are a total of 785 columns. The initial column serves as the label, while the subsequent columns correspond to each individual pixel within a 28x28 image. Notably, frameworks such as TensorFlow and PyTorch commonly offer convenient tools for both acquiring and loading such datasets.

#### 2) Data Preprocessing

Normalize the raw pixel values in the images to a range between 0 and 1 for faster convergence during training. This is achieved by dividing all pixel values by 255. Additionally, reshape the 28x28 images into a flattened vector of length 784 to serve as input to the neural network.In order to send it into a neural network, each image is transformed into a 1-D vector, which can be fed as input into the network. Moreover, for convenience, the vectorized image instances are transposed, effectively arranging each instance as a column in the dataset.

#### 3) Division into Training and Testing Sets

To ensure the model's ability to generalize to unseen data, the dataset is partitioned into training and testing subsets. This segregation enables the model to learn and then evaluate its performance on previously unseen examples. For this purpose, we opt for a ratio of 80% for the training set and 20% for the validation set, effectively maintaining an 8:2 distribution.

#### 4) One-Hot Encoding

In tasks such as digit recognition using the MNIST dataset, the labels represent categorical values (ranging from 0 to 9), signifying the respective class of the image (i.e., the digit it portrays).

During the training of a classification model, a loss function quantifies the dissimilarity between predicted and actual labels. The use of one-hot encoded labels allows the calculation of these losses. This convenience arises from the fact that each class is distinctly represented by an individual binary element within the encoded vector. This process involves the conversion of a single label (e.g., 3) into a binary vector (e.g., [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]), clearly indicating the class membership.

### D. MODEL ARCHITECTURE

The basic architecture of model consists of input layer few hidden layers and 10 output layers. Since there are 784 pixels , where each pixel acts as a feature , the input layer always has 784 neurons and 1 extra bias term. The number of hidden layers and number of neurons in each layer is a hyper-parameter which can be experimented . Finally the output layer contains 10 neurons one for each class.

Besides that each neuron must contain an activation function to introduce non linearity. If there were no any activation then the entire network becomes a linear system and is just equivalent to a single neuron no matter how deep the network is. By adding activation function are trying to separate the classes with non linear decision boundaries. Some choices of activation functions we experimented with are: A) Sigmoid activation B) Relu Activation C) Tanh activation

## V. RESULT ANALYSIS

For initializing initial values for the neural network's weights and biases, our approach involved drawing values from a uniform distribution spanning between -0.5 and 0.5.

### A. WEIGHT INITIALIZATION

**TABLE 1.** Performance for Different weight initialization

| Activation | Initialization method | Accuracy | Loss |
|---|---|---|---|
| Relu | Random uniform | 0.7977 | 0.8490 |
| Relu | He | 0.8512 | 0.6239 |
| Relu | Xavier | 0.8516 | 0.6234 |
| Tanh | Random uniform | 0.7461 | 0.8548 |
| Tanh | He | 0.8594 | 0.5792 |
| Tanh | Xavier | 0.8572 | 0.5879 |
| Sigmoid | Random uniform | 0.6290 | 1.1680 |
| Sigmoid | He | 0.7400 | 1.6020 |
| Sigmoid | Xavier | 0.7287 | 0.6668 |

During the initial phases of training, the random weight initialization yielded higher loss values in the initial iterations, accompanied by diminished accuracy. However, the accuracy exhibited an upward trend as training progressed, and the loss gradually diminished. Interestingly, it was observed that the randomly initialized model achieved a high final accuracy and lower loss.

Comparing the outcomes of random initialization to those of Xavier and He initialization revealed notable differences. Both the Xavier and He initialization methods showcased substantially lower loss values during the initial iterations, contributing to an accelerated convergence rate of the model. In particular, for a model featuring a single hidden layer comprising 256 nodes, the loss for random initialization was approximately 0.84, whereas Xavier and He initialization yielded losses around 0.62.

Moreover, the weight initialization techniques were applied across various activation functions, including ReLU, tanh, and sigmoid. Remarkably, this effect persisted consistently across all three activation functions.

This underlines the crucial role that weight initialization plays in the model's convergence and overall performance, with Xavier and He initialization providing a beneficial head start in terms of convergence speed and loss reduction.

### B. DETERMINING THE NUMBER OF NEURONS IN A HIDDEN LAYER

Determining the number of nodes in each layer of the neural network involves considering various factors. The input layer's size should correspond to the input dimensions, which are 784 in this case. Additionally, the output layer needs a node for every potential output label. With 10 labels (0-9 numbers) in this scenario, the output layer will encompass 10 neurons.

Nonetheless, the quantity of neurons within the hidden layers remains a matter of design. The network's complexity relies on considerations like:

- **Data Complexity:** Complex datasets often require larger networks. If the data exhibits intricate patterns or relationships, a larger network may be necessary.
- **Overfitting:** Too many neurons can lead to overfitting, where the model becomes overly specialized in training data and doesn't generalize to new data. Regularization techniques can help prevent this.
- **Computational Resources:** Larger networks demand more computational power and memory, influencing the network's size.
- **Empirical Experimentation:** Experimentation is crucial. Starting with a small network, evaluating its performance, and incrementally increasing its size helps identify optimal points between diminishing returns and overfitting.
- **Preceding Architectures:** Existing architectures or state-of-the-art models in your domain can guide the range of neurons used effectively.

To determine a suitable count, we initiated a neural network with xavier weight initialization. First, we implemented a network with:

- 784 input neurons
- variable hidden layer 1 neurons
- 128 hidden layers 2 neurons
- 10 output neurons

For that we calculated the accuracy and loss, and increased the number of hidden neurons by increments of 5. in our observation we saw that the hidden layer containing nodes less than 50 had significantly lower accuracy and higher losses than the accuracy and loss with higher neuron.

Interestingly, our investigation revealed a point of saturation in performance enhancement at approximately 125 neurons within the hidden layer. As seen in the figure (5), this plateau persisted until reaching the 300-neuron layer. The rationale behind this phenomenon can be attributed to the fact that around 125 neurons, the network possesses adequate capacity to capture the intrinsic patterns within the MNIST dataset. Beyond this threshold, further escalation in neuron count fails to yield corresponding accuracy improvements. The phenomenon points to the presence of diminishing returns, where excessive neuron augmentation does not proportionally enhance the model's predictive capabilities.

### C. SINGLE VS MULTIPLE HIDDEN LAYERS

We conducted a performance analysis by comparing two distinct neural network architectures using the MNIST dataset. The first architecture featured a single hidden layer containing 128 neurons, while the second configuration comprised multiple hidden layers with 512, 224, and 128 neurons. Both models underwent training for 100 epochs, employing Xavier weight initialization to ensure a balanced starting point.

**TABLE 2.** Performance for Different Network Architectures

| Architecture | Epoch | Accuracy | Loss |
|---|---|---|---|
| [784, 512, 224, 128, 10] | 100 | 0.8968 | 0.3833 |
| [784, 128, 10] | 100 | 0.8949 | 0.3922 |

Surprisingly, our results revealed minimal disparity in performance between the two architectures. The observed similarity in performance prompts us to consider potential reasons behind this outcome. One plausible explanation is rooted in the nature of the MNIST dataset itself. Given the dataset's relative simplicity and the prominence of its inherent patterns, it is conceivable that these patterns were adequately captured by the single hidden layer network we employed initially. While we cannot definitively conclude this to be the sole reason, it remains a valid hypothesis.

It's worth highlighting that the complexity of the dataset plays a pivotal role in determining the efficacy of different network architectures. While our results may not starkly differentiate the two architectures on the MNIST dataset, this should not overshadow the potential advantages of multi-layer networks in more intricate datasets. These deep architectures have a unique capability to discern intricate hierarchical features that might elude a shallower, single-layer network.

However, it's essential to exercise caution in interpreting these findings. While we anticipate discernible differences on more complex datasets, the interplay of various factors—such as activation functions, optimization techniques, and overall architecture capacity—can confound straightforward expectations. Therefore, future work should encompass a broader spectrum of datasets and rigorous cross-validation to comprehensively evaluate how different architectures perform across diverse scenarios.

### D. ACTIVATION FUNCTIONS

We tested different activation functions in the hidden layer, departing from our previous use of the ReLU activation function. To expedite convergence, we employed Xavier weight initialization without dropout for all three experiments, resulting in notable speed enhancements in learning.

## VI. NETWORK ARCHITECTURE

**TABLE 3.** Performance for Different Network Architectures and Activation Functions

| Architecture | Activation Function | Accuracy | Loss |
|---|---|---|---|
| [784, 128, 10] | Relu | 0.87 | 0.50 |
| [784, 128, 10] | Tanh | 0.87 | 0.51 |
| [784, 128, 10] | Sigmoid | 0.73 | 1.45 |

With the ReLU activation function, we observed faster convergence. By the 10th epoch, accuracy had already reached 65.18%, climbing to 87.25% by the 80th epoch, when our training concluded.

Switching to the tanh activation function further accelerated convergence, surpassing the progress achieved by ReLU. Accuracy stood at 70.77% by the 10th epoch and peaked at 86.86% by the end of the 80th epoch.

However, the sigmoid activation function demonstrated a slower trajectory among the three. Its accuracy was a mere 34.01% in the 10th epoch, gradually increasing to 73.97% by the 80th epoch.

The variation in performance across activation functions (ReLU, tanh, sigmoid) can be attributed to their distinct traits and their fit with the dataset and learning process.

**ReLU:** ReLU's simplicity and ability to tackle the vanishing gradient issue make it effective. It allows faster learning for positive inputs, which aligns well with the image data in MNIST. The dataset's clear pixel value transitions suit ReLU's activation patterns, aiding in capturing these patterns effectively.In the context of the MNIST dataset, where images represent handwritten digits, these transitions could represent edges, boundaries, or distinct features within the digit images.'

**tanh:** tanh's range from -1 to 1 suits datasets with zero-centered data like normalized MNIST pixels. Despite its vanishing gradient issue for extreme inputs, tanh can model both positive and negative data relationships due to its symmetry around zero.

**Sigmoid:** Sigmoid's (0, 1) range and sensitivity around zero could explain its performance. Yet, its rapid gradient saturation for strong inputs hampers learning, especially in deeper networks. For MNIST's pattern-centric data, sigmoid's saturation for intense inputs can limit its pattern recognition ability.

**Weight Initialization:** Xavier initialization's role in gradient normalization helps counter the vanishing gradient problem, benefiting ReLU and tanh activations.

Activation function performance depends on their characteristics, compatibility with data distribution, and interaction with weight initialization. The discrepancies observed in accuracy and convergence stem from these factors, highlighting how certain activations align better with MNIST's intrinsic attributes.

### A. DROPOUT

Dropout is a regularization technique used in neural networks to prevent overfitting and improve generalization performance. Overfitting occurs when a neural network learns to perform exceptionally well on the training data but struggles to perform well on unseen data (validation or test data). Dropout is a simple yet effective way to combat overfitting by introducing randomness during training.

We used neuron deactivation for our dropout. To experiment with dropout, we masked a fraction of neurons in the layers randomly by passing a parameter `dropout_prob`. This creates a dropout mask so that during the training phase, the output of masked neurons is set to zero.

First we trained the models for 50 epoch, the table below shows the result. But we realized the network could further be trained. So, for the second run, we a model 512 hidden neurons for 150 epochs. We trained three single hidden layer neural networks with 0%, 10%, and 20% of the neurons deactivated during training. we observed a training accuracy of 89% for the models with no dropout and 10% dropout. For the model with 20% dropout, we observed the accuracy to be 88%. Moving to test accuracy and loss, as seen in the figure 11, the accuracy was almost the same in all three models. However, the loss slightly increased as the dropout increased.

**TABLE 4.** Effect of Dropout Probability on Network Performance

| Network | epoch | Dropout Probability | Accuracy | Loss |
|---|---|---|---|---|
| [784, 224, 10] | 50 | 0 | 0.85 | 0.63 |
| [784, 224, 10] | 50 | 0.1 | 0.83 | 0.67 |
| [784, 224, 10] | 50 | 0.2 | 0.81 | 0.75 |
| [784, 224, 10] | 50 | 0.5 | 0.70 | 1.03 |
| [784, 224, 10] | 50 | 0.8 | 0.44 | 1.67 |

**TABLE 5.** Effect of Dropout Probability on Network Performance

| Network | Epoch | Dropout Probability | Accuracy | Loss |
|---|---|---|---|---|
| [784, 512, 10] | 150 | 0 | 0.89 | 0.3778 |
| [784, 512, 10] | 150 | 0.1 | 0.89 | 0.3978 |
| [784, 512, 10] | 150 | 0.2 | 0.88 | 0.4243 |

It's important to consider a few potential reasons for this result:

- **Dataset Size and Complexity:** The MNIST dataset is relatively small and contains well-defined patterns. In such cases, the benefits of dropout regularization might not be as pronounced as in larger and more complex datasets. Dropout is particularly effective when dealing with larger and more intricate datasets, where overfitting is a more significant concern.
- **Model Architecture:** The architecture of neural network can also influence the effectiveness of dropout. If your network is already relatively simple or shallow, overfitting might not be a major issue even without dropout. Dropout tends to provide more pronounced benefits in deeper networks.

## VII. CONCLUSION

This project aimed to enhance the performance of a Multi-Layer Perceptron (MLP) on the MNIST dataset by systematically exploring various factors that influence its training process and final accuracy. Through the meticulous design of experiments, we investigated the impact of different weight initialization techniques, activation functions, and dropout regularization on the model's performance.

Our findings revealed that weight initialization plays a crucial role in the convergence speed and overall performance of the MLP. Certain techniques, such as Xavier/Glorot initialization, demonstrated a clear advantage in facilitating faster convergence and higher accuracy. Moreover, the choice of activation functions showcased distinct effects on the model's ability to capture complex patterns within the dataset. We observed that ReLU and its variants tend to outperform others by mitigating the vanishing gradient problem and enabling more effective learning.

Incorporating dropout regularization during training can show positive effect on the model's generalization ability. This technique effectively reduced overfitting and improved the MLP's performance on unseen data, leading to a more robust and reliable model.

As machine learning practitioners, understanding the nuances of these factors is pivotal in designing effective neural network architectures. This project underscores the importance of a systematic approach to hyperparameter tuning and experimentation. By optimizing these key components—weight initialization, activation functions, and dropout regularization—we not only enhanced the accuracy of our MLP on the MNIST dataset but also gained valuable insights into the inner workings of neural networks.

In the ever-evolving field of deep learning, this project serves as a testament to the significance of thorough experimentation and analysis. As we continue to push the boundaries of AI research, the knowledge gained from this project will undoubtedly contribute to the development of more efficient and accurate neural network models for a wide range of applications, ultimately advancing the field as a whole.

## REFERENCES


[1] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *"Deep Learning."* MIT Press.
[2] Daniel Jurafsky & James H. Martin. *Speech and Language Processing*.
[3] M. Narasimha Murty, V. Susheela Devi. *Pattern Recognition: An Algorithmic Approach*.


**PILOT KHADKA** is a student at Institute of Engineering, Thapathali Campus. He is expected to graduate in Bachelor of Computer Engineering in 2024. During his time at Thapathali Campus, Pilot has actively engaged in various academic and extracurricular activities. He has participated in coding competitions, collaborated on software development projects, and demonstrated a keen interest in exploring new technologies.

**KSHITIZ POUDEL** is a student at Institute of Engineering, Thapathali Campus. He is expected to graduate in Bachelor of Computer Engineering in 2024. His relentless pursuit of knowledge and dedication to uplifting and empowering others make him an exceptional contributor and an invaluable asset to the academic community.

# VIII. APPENDIX

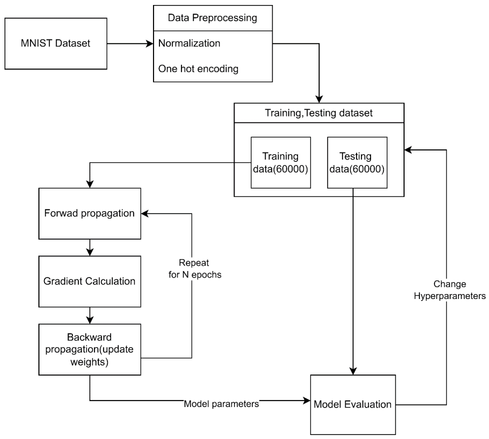


**FIGURE 1.** System Block Diagram

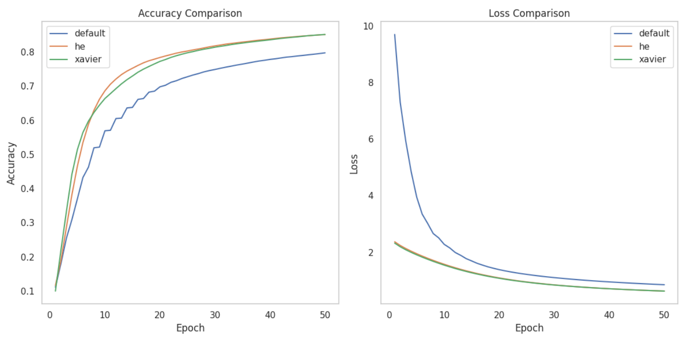


**FIGURE 2.** Weight initialization for Relu activation

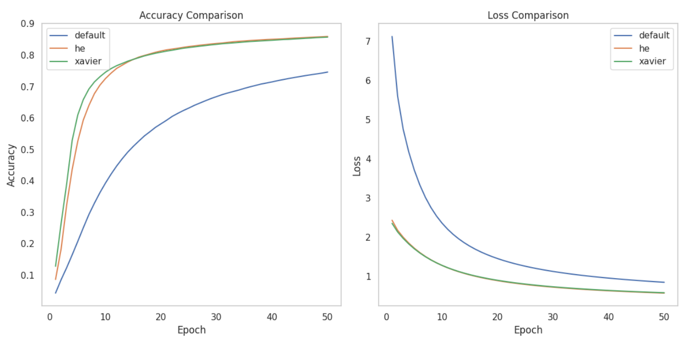


**FIGURE 3.** Weight initialization for Tanh activation

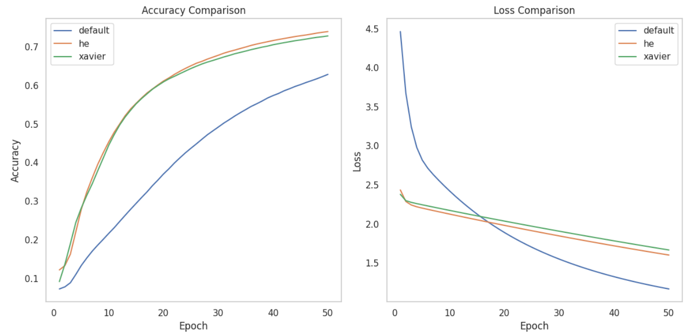


**FIGURE 4.** Weight initialization for sigmoid activation

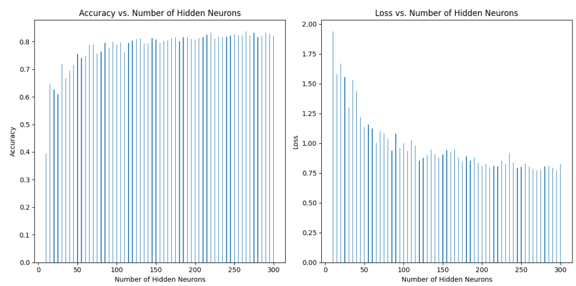


**FIGURE 5.** System Block Diagram

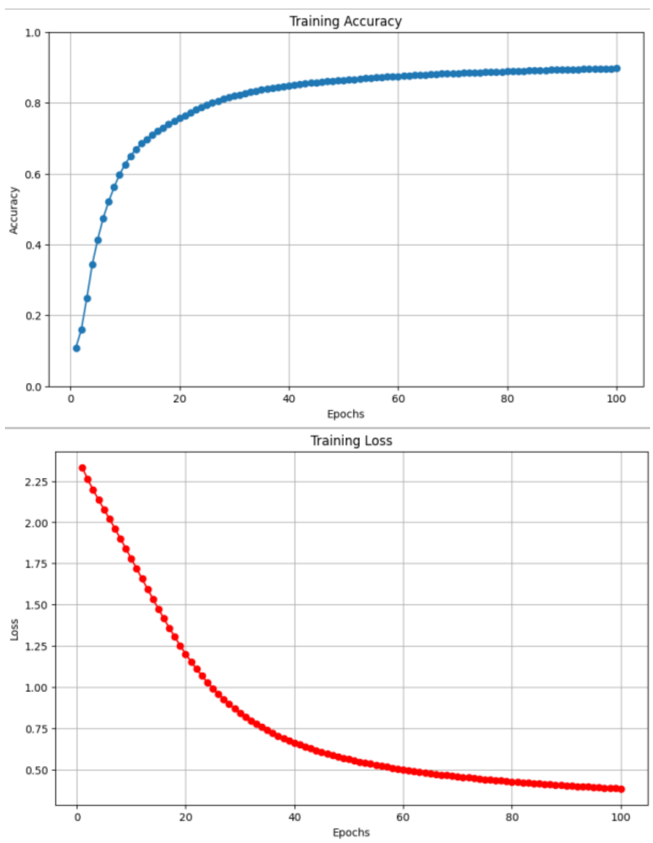


**FIGURE 6.** Accuracy/Loss curve of Deeper Network

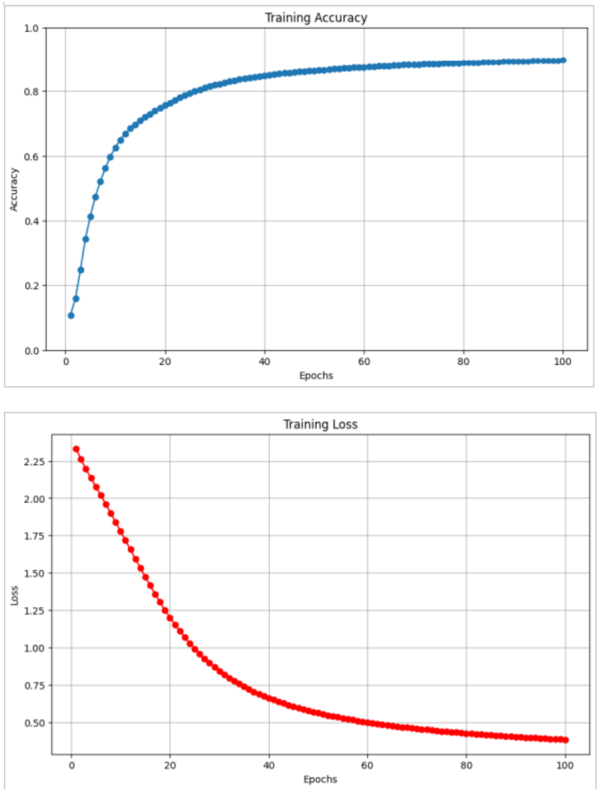


**FIGURE 7. Accuracy/Loss curve of Shallow Network**

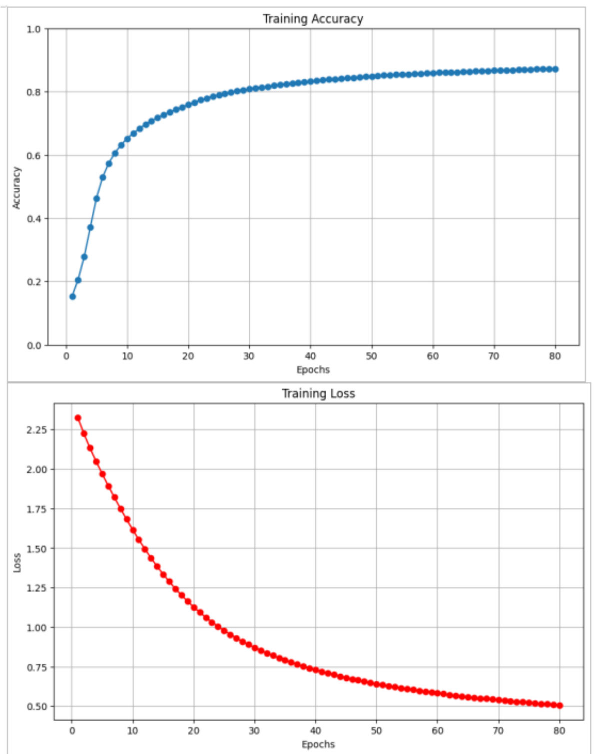


**FIGURE 8. Accuracy/Loss curve of Network with relu**

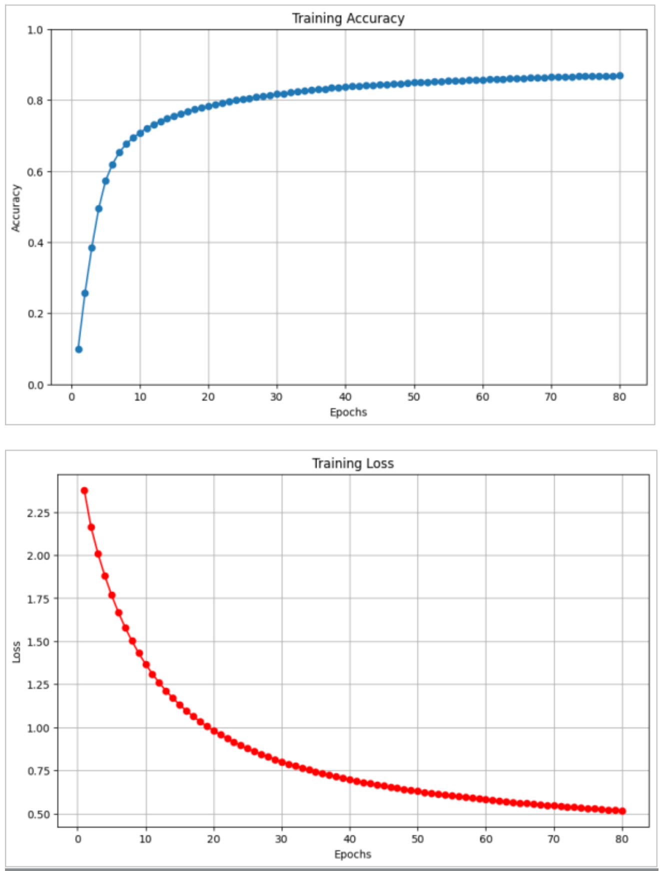


**FIGURE 9. Accuracy/Loss curve of network with tanh**

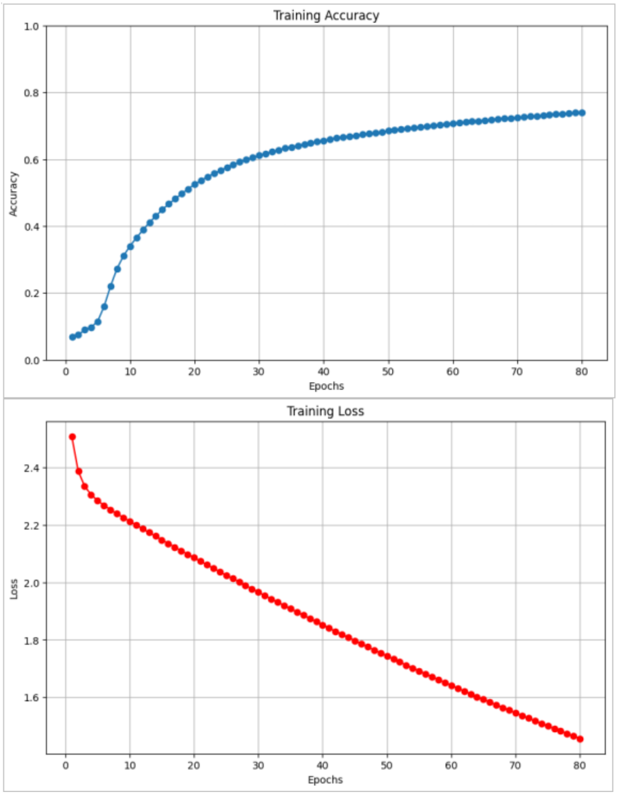


FIGURE 10. Accuracy/Loss curve of network with sigmoid

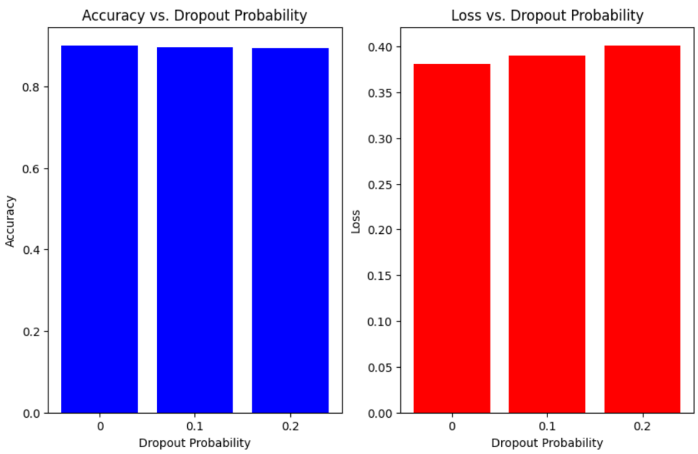


FIGURE 12. Accuracy/loss comparision with dropouts 0,0.1,0.2

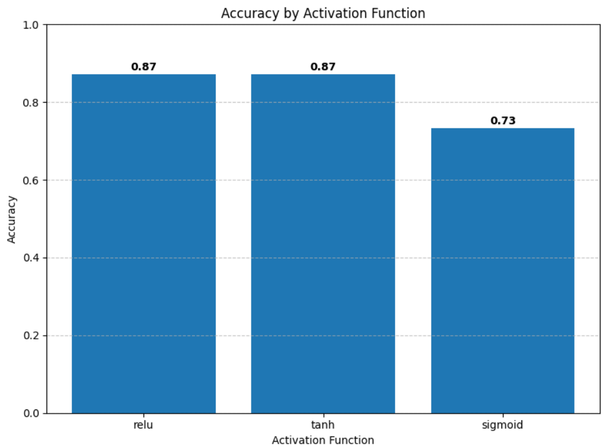


FIGURE 11. accuracy comparision of three activations

## IX. CODE

```python


import numpy as np
import matplotlib.pyplot as plt

def init_params(layers_dims, initialization="default"):
    """
    Initializes the parameters (weights and biases) for a neural network.

    Parameters:
    layers_dims (list): List of integers representing the dimensions of each layer in the neural network.
    initialization (str): Initialization method to use for parameter initialization.
        - "default": Random initialization using a uniform distribution between -0.5 and 0.5.
        - "xavier": Xavier/Glorot initialization for better training convergence.
        - "he": He initialization for training deeper networks.

    Returns:
    params (dict): A dictionary containing the initialized parameters for each layer.
    """
    params = {}

    for layer in range(1, len(layers_dims)):
        input_dim = layers_dims[layer - 1]
        output_dim = layers_dims[layer]

        # Xavier weight initialization
        if initialization == "xavier":
            params["W" + str(layer)] = np.random.uniform(
                -np.sqrt(6 / (input_dim + output_dim)),
                np.sqrt(6 / (input_dim + output_dim)),
                (output_dim, input_dim),
            )
            params["b" + str(layer)] = np.random.uniform(
                -np.sqrt(6 / (input_dim + output_dim)),
                np.sqrt(6 / (input_dim + output_dim)),
                (output_dim, 1),
            )
        # He weight initialization
        elif initialization == "he":
            params["W" + str(layer)] = np.random.normal(
                0, np.sqrt(2 / input_dim), (output_dim, input_dim)
            )
            params["b" + str(layer)] = np.random.normal(
                0, np.sqrt(2 / input_dim), (output_dim, 1)
            )

        # Default initialization uniform distribution
        else:
            params["W" + str(layer)] = np.random.uniform(
                -0.5, 0.5, (output_dim, input_dim)
            )
            params["b" + str(layer)] = np.random.uniform(-0.5, 0.5, (output_dim, 1))
    return params

def forward_prop(X, params,activation="relu", dropout_prob=0):
    """
    Perform forward propagation through the neural network to compute activations.

    Parameters:
        X (numpy.ndarray): Input data of shape (input_size, m), where input_size is the number of features and m is the number of examples.
        params (dict): Dictionary containing network parameters including weights (W) and biases (b) for each layer.
        dropout_prob (float): Dropout probability for applying dropout regularization to hidden layers (default 0).

    Returns:
        activations (dict): Dictionary containing computed activations for each layer.
        dropout_masks (dict): Dictionary containing dropout masks applied to hidden layers.
    """
    # Get the number of layers directly from the length of W parameters
    L = len(params) // 2
    activations = {}
    activations["A0"] = X # Input activations
    dropout_masks = {} # Dictionary to store dropout masks

    for l in range(1, L):
        # Calculate Z and A for intermediate layers using ReLU activation
        activations["Z" + str(l)] = np.dot(params["W" + str(l)], activations["A" + str(l - 1)]) + params["b" + str(l)]

        if activation == 'tanh':
            activations["A" + str(l)] = _tanh(activations["Z" + str(l)])  # Tanh activation
        elif activation == 'sigmoid':
            activations["A" + str(l)] = 1 / (1 + np.exp(-activations["Z" + str(l)]))  # Sigmoid activation
        else :
            activations["A" + str(l)] = _relu(activations["Z" + str(l)])  # ReLU activation

        if l < L:
            dropout_mask = np.random.rand(*activations["A" + str(l)].shape) < (1 - dropout_prob)  # Inverted dropout mask
            activations["A" + str(l)] *= dropout_mask
            activations["A" + str(l)] /= (1 - dropout_prob)  # Scale to maintain expected value
            dropout_masks["D" + str(l)] = dropout_mask

    # Calculate Z and A for the output layer using softmax activation
    activations["Z" + str(L)] = np.dot(params["W" + str(L)], activations["A" + str(L - 1)]) + params["b" + str(L)]
    exp_scores = np.exp(activations["Z" + str(L)])
    activations["A" + str(L)] = exp_scores / np.sum(exp_scores, axis=0, keepdims=True)  #
```

```python
    Softmax activation

    return activations,dropout_masks


def back_prop(activations, params, Y,dropout_masks,activation="relu"):
    """
    Perform backpropagation to compute gradients of the loss with respect to parameters.

    Parameters:
        activations (dict): Dictionary containing computed activations for each layer during forward propagation.
        params (dict): Dictionary containing network parameters including weights (W) and biases (b) for each layer.
        Y (numpy.ndarray): True labels (ground truth) of shape (1, m), where m is the number of examples.
        dropout_masks (dict): Dictionary containing dropout masks applied to hidden layers.

    Returns:
        grads (dict): Dictionary containing computed gradients of the loss with respect to parameters.
    """
    L = len(params) // 2
    one_hot_Y = one_hot_encode(Y)
    m = one_hot_Y.shape[1]

    derivatives = {}
    grads = {}

    # for layer L
    derivatives["dZ" + str(L)] = activations["A" + str(L)] - one_hot_Y
    grads["dW" + str(L)] = (
        1 / m * np.dot(derivatives["dZ" + str(L)], activations["A" + str(L - 1)].T)
    )
    grads["db" + str(L)] = 1 / m * np.sum(derivatives["dZ" + str(L)])

    # for layers L-1 to 1
    for l in reversed(range(1, L)):
        if activation == 'relu':
            _activation_derivative = _relu_derivative
        elif activation == 'tanh':
            _activation_derivative = _tanh_derivative
        elif activation == 'sigmoid':
            _activation_derivative = _sigmoid_derivative

        derivatives["dZ" + str(l)] = np.dot(
            params["W" + str(l + 1)].T, derivatives["dZ" + str(l + 1)]
        ) * _activation_derivative(activations["Z" + str(l)])

        # apply dropout mask
        if l < L:
            derivatives["dZ" + str(l)] *= dropout_masks["D" + str(l)]

        grads["dW" + str(l)] = (
            1 / m * np.dot(derivatives["dZ" + str(l)], activations["A" + str(l - 1)].T)
        )
        grads["db" + str(l)] = (
            1 / m * np.sum(derivatives["dZ" + str(l)], axis=1, keepdims=True)
        )

    return grads


def update_params(params, grads, alpha):
    """
    Update network parameters using gradient descent optimization.

    Parameters:
        params (dict): Dictionary containing network parameters including weights (W) and biases (b) for each layer.
        grads (dict): Dictionary containing gradients of the loss with respect to parameters.
        alpha (float): Learning rate, controlling the step size of parameter updates.

    Returns:
        params_updated (dict): Dictionary containing updated network parameters.
    """
    # number of layers
    L = len(params) // 2

    params_updated = {}
    for l in range(1, L + 1):
        params_updated["W" + str(l)] = (
            params["W" + str(l)] - alpha * grads["dW" + str(l)]
        )
        params_updated["b" + str(l)] = (
            params["b" + str(l)] - alpha * grads["db" + str(l)]
        )

    return params_updated

def train(X, Y, params, max_iter=10, learning_rate=0.1,dropout_prob=0,activation="relu"):
    """
    Trains a neural network model using gradient descent optimization.

    Parameters:
    X (numpy.ndarray): Input data of shape (input_size, num_samples).
    Y (numpy.ndarray): Ground truth labels of shape (1, num_samples).
    params (dict): Dictionary containing the parameters of the neural network.
                   Keys are "W1", "b1", ..., "WL", "bL" where L is the number of layers.
    max_iter (int, optional): Maximum number of training iterations. Default is 10.
    learning_rate (float, optional): Learning rate for gradient descent. Default is 0.1.
    dropout_prob (float, optional): Dropout probability for regularization. Default is 0.
    activation (str, optional): Activation function to use in hidden layers. Default is "relu".

    Returns:
    tuple: A tuple containing updated parameters, list of accuracies per iteration, and list of losses per iteration.
    """
    # Initialize parameters Wl, bl for layers l
```

```python
=1,...,L
    L = len(params) // 2
    accuracies = []
    losses = []

    for iteration in range(1, max_iter + 1):
        # Forward propagation
        activations,dropout_mask = forward_prop(X,
     params,activation,dropout_prob)

        # Make predictions
        Y_hat = get_predictions(activations["A" +
    str(L)])

        # Compute accuracy
        accuracy = get_accuracy(Y_hat, Y)
        accuracies.append(accuracy)

        # Compute loss (cross-entropy)
        loss = cross_entropy(one_hot_encode(Y),
    activations["A" + str(L)])
        losses.append(loss)

        # Backpropagation
        gradients = back_prop(activations, params,
     Y,dropout_mask,activation)

        # Update parameters
        params = update_params(params, gradients,
    learning_rate)

        # Print progress
        if iteration ==1 or (iteration%5) == 0:
            print("Iteration {}: Accuracy = {},
    Loss = {}".format(iteration, accuracy, loss))

    return params,accuracies,losses

def test(X_test, Y_test, params,activation):
    """
    Evaluates the trained neural network model on
    test data.

    Parameters:
    X_test (numpy.ndarray): Test input data of
    shape (input_size, num_samples).
    Y_test (numpy.ndarray): Ground truth labels
    for test data of shape (1, num_samples).
    params (dict): Dictionary containing the
    parameters of the neural network.
                   Keys are "W1", "b1", ..., "WL",
     "bL" where L is the number of layers.
    activation (str): Activation function used in
    the hidden layers during forward propagation.

    Returns:
    tuple: A tuple containing test accuracy and
    loss.
    """
    L = len(params) // 2

    # Forward propagation
    activations,_ = forward_prop(X_test, params,
    activation,0)

    # Make predictions
    Y_hat = get_predictions(activations["A" + str(
    L)])

    # Compute test accuracy
    test_accuracy = get_accuracy(Y_hat, Y_test)
    loss = cross_entropy(one_hot_encode(Y_test),
    activations["A" + str(L)])

    return test_accuracy,loss

def one_hot_encode(Y):
    Y_one_hot = np.zeros((Y.shape[0], Y.max() + 1)
    )
    # set to 1 the corret indices
    Y_one_hot[np.arange(Y.shape[0]), Y] = 1
    # transpose
    Y_one_hot = Y_one_hot.T
    return Y_one_hot

def cross_entropy(Y_one_hot, Y_hat, epsilon=1e-10)
    :
    # clip predictions to avoid values of 0 and 1
    Y_hat = np.clip(Y_hat, epsilon, 1.0 - epsilon)
    # sum on the columns of Y_hat * np.log(Y),
    then take the mean
    # between the m samples
    cross_entropy = -np.mean(np.sum(Y_one_hot * np
    .log(Y_hat), axis=0))
    return cross_entropy

def shuffle_rows(data):
    # Convert input dataframe to ndarray
    data = np.array(data)
    np.random.shuffle(data)
    return data


def normalize_pixels(data):
    return data / 255.0

def _relu(Z):
    return np.maximum(Z, 0)


def _softmax(Z):
    A = np.exp(Z) / sum(np.exp(Z))
    return A


def _relu_derivative(Z):
    return Z > 0

def _softmax_derivative(Z):
    dZ = np.exp(Z) / sum(np.exp(Z)) * (1.0 - np.
    exp(Z) / sum(np.exp(Z)))
    return dZ

def _tanh(x):
    return (np.exp(x) - np.exp(-x)) / (np.exp(x) +
     np.exp(-x))

def _tanh_derivative(x):
    return 1 - np.tanh(x) ** 2

def _sigmoid_derivative(x):
    sigmoid = 1 / (1 + np.exp(-x))
    return sigmoid * (1 - sigmoid)



def get_predictions(AL):
    # get the max index by the columns
    return np.argmax(AL, axis=0)


def get_accuracy(Y_hat, Y):
    return np.sum(Y_hat == Y) / Y.size

def plot_accuracy_and_loss(accuracies, losses,
    max_iter):
    # Plot training accuracy
    plt.figure(figsize=(10, 6))
```

```
    plt.plot(range(1, max_iter + 1), accuracies, marker='o')
    plt.title("Training Accuracy")
    plt.xlabel("Epochs")
    plt.ylim(0,1)
    plt.ylabel("Accuracy")
    plt.grid()
    plt.show()

    # Plot training loss
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, max_iter + 1), losses, marker='o', color='r')
    plt.title("Training Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.grid()
    plt.show()

# load training data
import pandas as pd
df_train = pd.read_csv("mnist_train.csv")

# shuffle the data
df_train = shuffle_rows(df_train)

# split train and validation set
train_val_split = 0.8
train_size = round(df_train.shape[0] * train_val_split)
data_train = df_train[:train_size, :].T
data_val = df_train[train_size:, :].T

# divide input features and target feature
X_train = data_train[1:]
y_train = data_train[0]
X_test = data_val[1:]
y_test = data_val[0]

# normalize training and val sets
X_train = normalize_pixels(X_train)
X_test = normalize_pixels(X_test)

print(X_test.shape)
print(y_test.shape)

# set network and optimizer parameters
layers_dims = [784,128, 10]
# layers_dims = [784, 10, 10]
max_iter = 80
alpha = 0.1
dropout_prob = 0
accuracies =[]
losses =[]

activation=["relu","tanh","sigmoid"]
for a in activation:
    params = init_params(layers_dims,'xavier')


    # train the network
    trained_params,train_acc,train_loss = train(
        X_train, y_train,params, max_iter, alpha, dropout_prob,a
    )
    plot_accuracy_and_loss(train_acc,train_loss, max_iter)
    accuracy,loss = test(X_test,y_test, trained_params,a)
    accuracies.append(accuracy)
    losses.append(loss)

import seaborn as sns

plt.figure(figsize=(8, 6))  # Adjust the figure size
# Create the bar plot
plt.bar(activation, accuracies)

# Adding title and labels
plt.title('Accuracy by Activation Function')
plt.xlabel('Activation Function')
plt.ylabel('Accuracy')

# Adding data labels on top of the bars
for i, acc in enumerate(accuracies):
    plt.text(i, acc + 0.01, f'{acc:.2f}', ha='center', color='black', fontweight='bold')

plt.ylim(0, 1)  # Set y-axis limits
plt.grid(axis='y', linestyle='--', alpha=0.7)  # Add horizontal grid lines

plt.tight_layout()  # Adjust spacing

plt.show()  # Display the plot

test_accuracy,test_loss = test(X_test, y_test, trained_params)
print("Test Accuracy: {:.2f}, Test loss: {:.2f}".format(test_accuracy * 100, test_loss))

hidden_neurons_range = range(10, 201, 5)
accuracies = []
losses = []

for hidden_neurons in hidden_neurons_range:
    layers_dims = [784, hidden_neurons, 128, 10]  # Update hidden layer neurons
    params = init_params(layers_dims,'xavier')  # Reinitialize parameters

    # Train the network
    trained_params, _,__ = train(X_train, y_train, params, max_iter, alpha, dropout_prob)

    accuracy,loss = test(X_test,y_test, trained_params)
    accuracies.append(accuracy)
    losses.append(loss)

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.bar(hidden_neurons_range, accuracies)
plt.xlabel('Number of Hidden Neurons')
plt.ylabel('Accuracy')
plt.title('Accuracy vs. Number of Hidden Neurons')

plt.subplot(1, 2, 2)
plt.bar(hidden_neurons_range, losses)
plt.xlabel('Number of Hidden Neurons')
plt.ylabel('Loss')
plt.title('Loss vs. Number of Hidden Neurons')

plt.tight_layout()
plt.show()

# Deeper networks

m_layers_dims = [784, 512,224,128, 10]  # Update hidden layer neurons
m_params = init_params(layers_dims,'xavier')  # Reinitialize parameters
max_iter=100
# Train the network
m_trained_params, m_train_accuracy,m_train_loss = train(X_train, y_train, params, max_iter,
```

```
    alpha, dropout_prob)

m_accuracy,m_loss = test(X_test,y_test, trained_params)

plot_accuracy_and_loss(m_train_accuracy, m_train_loss, max_iter)

s_layers_dims = [784,128, 10]  # Update hidden layer neurons
s_params = init_params(layers_dims,'xavier')  # Reinitialize parameters
max_iter=100
# Train the network
s_trained_params, s_train_accuracy,s_train_loss = train(X_train, y_train, params, max_iter, alpha, dropout_prob)

s_accuracy,s_loss = test(X_test,y_test, trained_params)

plot_accuracy_and_loss(s_train_accuracy, s_train_loss,max_iter)

#Compare initialization methods

import matplotlib.pyplot as plt
# set network and optimizer parameters
layers_dims = [784, 256, 10]
# layers_dims = [784, 10, 10]
max_iter = 50
alpha = 0.1
dropout_prob= 00

# Define the initialization methods you want to compare
initialization_methods = ['default', 'he', 'xavier']
activation = ['default','relu','sigmoid']
# Initialize an empty dictionary to store accuracies and losses for each method
results = {}

# Loop through each initialization method
for a in activation:
    for method in initialization_methods:
        # Initialize parameters using the current method
        params = init_params(layers_dims, method)

        # Train the network and collect accuracies and losses
        updated_params,accuracies, losses = train(X_train, y_train, params, max_iter, alpha, dropout_prob,a)

        # Store the results for the current method
        results[method] = {'accuracies': accuracies, 'losses': losses}

    # Plotting the results
    plt.figure(figsize=(12, 6))

    # Plot accuracy for each method
    plt.subplot(1, 2, 1)
    for method, data in results.items():
        plt.plot(range(1, max_iter + 1), data['accuracies'], label=method)
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Accuracy Comparison')
    plt.grid()
    plt.legend()

    # Plot loss for each method
    plt.subplot(1, 2, 2)
    for method, data in results.items():
        plt.plot(range(1, max_iter + 1), data['losses'], label=method)
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Loss Comparison')
    plt.grid()
    plt.legend()

    plt.tight_layout()
    plt.show()

dropout_prob = [num * 0.1 for num in range(0, int(1 / 0.1) + 1)]
for alpha in dropout_prob:
    print(alpha)

# Dropout during training
# set network and optimizer parameters
layers_dims = [784,224, 10]
max_iter = 50
alpha = 0.1

dropout_prob =[0,0.1,0.2,0.5,0.8]
accuracies =[]
losses =[]

for d in dropout_prob:
    params = init_params(layers_dims,'xavier')
    # train the network
    trained_params,train_acc,train_loss = train(
        X_train, y_train,params, max_iter, alpha,d,"relu"
    )
    accuracy,loss = test(X_test,y_test, trained_params,a)
    plot_accuracy_and_loss(train_acc,train_loss, max_iter)
    accuracies.append(accuracy)
    losses.append(loss)

# Dropout during training
# set network and optimizer parameters
layers_dims = [784,512, 10]
max_iter = 150
alpha = 0.1

dropout_prob =[0,0.1,0.2]
accuracies =[]
losses =[]

for d in dropout_prob:
    params = init_params(layers_dims,'xavier')
    # train the network
    trained_params,train_acc,train_loss = train(
        X_train, y_train,params, max_iter, alpha,d,"relu"
    )
    accuracy,loss = test(X_test,y_test, trained_params,"relu")
    plot_accuracy_and_loss(train_acc,train_loss, max_iter)
    accuracies.append(accuracy)
    losses.append(loss)

print(accuracies)

print(losses)

print(dropout_prob)

plt.figure(figsize=(10, 6))
```

```python

# Bar plot for accuracies
plt.subplot(1, 2, 1)
plt.bar(['0','0.1','0.2'], accuracies, color='blue')
plt.title('Accuracy vs. Dropout Probability')
plt.xlabel('Dropout Probability')
plt.ylabel('Accuracy')

# Bar plot for losses
plt.subplot(1, 2, 2)
plt.bar(['0','0.1','0.2'], losses, color='red')
plt.title('Loss vs. Dropout Probability')
plt.xlabel('Dropout Probability')
plt.ylabel('Loss')

plt.show()

# Define hyperparameter grid
layer_options = [
    [784,256,10],
    [784,256,256,10],
    [784, 256, 128, 64, 10],
]
alpha=0.1
init_methods = ['default', 'xavier', 'he']
max_iter=50
best_accuracy = 0
best_params = None

for layers_dims in layer_options:
    for initialization_method in init_methods:
        # Initialize Neural network
        params = init_params(layers_dims, initialization_method)

        # Train the neural network
        updated_param, accuracy,loss = train(X_train, y_train, params, max_iter, alpha)

        # Check if this set of hyperparameters is the best so far
        if accuracy > best_accuracy:
            best_accuracy = accuracy
            best_params = {
                'layers_dims': layers_dims,
                'alpha': alpha,
                'initialization_method': initialization_method
            }

print("Best hyperparameters:", best_params)
print("Best validation accuracy:", best_accuracy)

# Test best model on test data
test_accuracy = evaluate_model(model, X_test, y_test)
print("Test accuracy:", test_accuracy)
```