Date of Submission August 2024

*Digital Object Identifier Not Applicable*

# A Study on the Implementation of Artificial Neural Network

**NIMESH GOPAL PRADHAN[1], RAMAN BHATTARAI[1]**
[1]Department of Electronics and Computer Engineering, Thapathali Campus, Tribhuvan University, Kathmandu, Nepal

**ABSTRACT** Artificial Neural Networks (ANNs) are models inspired by the human brain that learn from data and solve complex problems by recognizing patterns. These networks are made up of interconnected nodes called neurons, which are arranged in layers. Each neuron processes input data and uses an activation function to produce an output. ANNs are widely used in various applications, including image recognition, natural language processing, and predictive analytics, because of their ability to handle complex data and capture non-linear relationships. To demonstrate how ANNs function, the MNIST dataset, which is commonly used for recognizing handwritten digits, is used in this paper.

**INDEX TERMS** Activation function, artificial neural network (ANN), natural language processing, predictive analytics

## I. INTRODUCTION

Artificial Neural Networks (ANNs) are computer models inspired by how the human brain functions. The concept behind ANNs started in the 1940s when researchers like Warren McCulloch and Walter Pitts developed a simple model of a biological neuron. This early model could perform basic logical functions and set the foundation for creating more advanced neural networks that can learn from data, recognize patterns, and make decisions. ANNs are made up of connected nodes, called neurons, organized into layers. Each neuron takes in input, processes it, and sends out an output to the next layer, allowing the network to understand complex relationships in the data.

An ANN typically has three main types of layers: the input layer, one or more hidden layers, and the output layer. The input layer takes in the raw data, such as numbers or images. This data then passes through one or more hidden layers, where the actual learning happens and the network learns and discovers patterns. Neurons in the hidden layers apply activation functions to the inputs, making the network capable of understanding complex, non-linear relationships in the data. The final output layer gives the network's prediction or classification based on the task. ANNs learn by adjusting the connections between neurons through a process called backpropagation, which reduces the difference between the predicted and actual outputs. Over time, this process helps the network improve its accuracy.

ANNs are versatile and powerful tools used in various fields due to their ability to handle complex tasks. One of their main uses is in image recognition, where they help identify and classify objects, faces, and patterns in images. This capability has led to advancements in areas like medical imaging, self-driving cars, and facial recognition technology. ANNs are also widely used in natural language processing (NLP), which allows machines to understand and work with human language. This includes applications like speech recognition, language translation, and analyzing the sentiment of text. Additionally, ANNs play a key role in predictive analytics, where they analyze large datasets to predict trends, assess risks, and support decision-making in areas like finance, healthcare, and marketing. The ability of ANNs to learn from examples and handle imperfect data makes them valuable for solving a range of complex real-world problems.

A common way to demonstrate how ANNs work is by using the digit dataset, which contains 40,000 grayscale images of handwritten digits, each 28x28 pixels in size. The dataset is commonly used to test and compare different machine learning models,

including ANNs. In this case, the digit dataset is used to train an ANN to recognize and classify handwritten digits accurately. The input layer of the network takes in the pixel values from each image, while the hidden layers learn to identify features such as like edges and shapes. The output layer then predicts the digit represented by the image. By training the network over multiple rounds and adjusting the weights between neurons, the ANN learns to correctly classify new images it hasn't seen before. The digit dataset serves as a benchmark to illustrate how effectively ANNs can recognize patterns and their potential for wider applications in image processing and other areas.

The learning process in ANNs is a key factor in their ability to model and understand complex data. This process involves training the network using a set of labeled examples, where the network adjusts the weights of connections between neurons to minimize the error in its predictions. This adjustment is typically done through an algorithm called backpropagation, which works by calculating the gradient of the loss function with respect to each weight and updating the weights in the direction that reduces the error. However, training ANNs can be challenging due to issues like overfitting, where the network becomes too tailored to the training data and performs poorly on new, unseen data. To address such challenges, techniques such as regularization, dropout, and early stopping are used to improve the network's ability to generalize to new data. Additionally, the choice of hyperparameters, like the learning rate, the number of hidden layers, and the size of the network, can significantly impact the performance of an ANN, making the design and tuning of these networks a complex and iterative process. Despite these challenges, the ability of ANNs to learn from data and improve their performance over time is one of their greatest strengths.

## II. RELATED WORKS

Neural networks are mathematical models inspired by biological neurons, designed to process information through interconnected nodes. Their complex structure simplifies problem-solving and is utilized in various applications, particularly in computer science and electronics. Researchers use artificial neural networks (ANNs) to address challenges such as pattern recognition, optimization, and prediction. This paper [1] provides foundational knowledge about ANNs, including their architecture, models, and real-life applications across different sectors.

This book 'Artificial Neural Networks: A Practical Course' [2] offers a thorough examination of neural networks, covering their evolution, structure, and applications. The first half focuses on theoretical aspects and key architectures applicable to various scenarios. The second half provides practical guidance on using neural networks to address real-world problems and details the implementation processes used to achieve results. This comprehensive approach aids in refining experimental techniques and selecting suitable neural network architectures for specific applications.

This paper [3] reviews recent developments in Artificial Neural Networks (ANNs) and explores new methodologies and applications. The focus is on advancing research by analyzing recent achievements and introducing innovative approaches in the field.

Artificial neural networks are crucial to the emerging field of artificial intelligence. This paper [4] reviews the development and key characteristics of artificial neural networks, including their nonlinear, non-limitative, non-qualitative, and nonconvex properties. It also explores their applications across various fields such as information technology, medicine, economics, control systems, transportation, and psychology, and looks ahead to future developments in the field.

This web article [5], part of the second chapter on Deep Learning, focuses on Artificial Neural Networks (ANNs). It covers essential topics including the types of ANNs, such as FeedForward and Feedback topologies, and discusses various activation functions like Sigmoid, Tanh, and ReLU. The chapter also explains how backpropagation works and provides practical implementation examples using Keras and TensorFlow, including a case study on churn modeling for a bank, where ANNs are used to predict customer departure based on various data.

Backpropagation is a widely used method in pattern recognition and fault diagnosis, with its basic equations and applications discussed, including those for dynamic systems and control. The method's extensions address systems beyond neural networks, including simultaneous equations and recurrent networks, with pseudocode provided for clarity. The paper [6] also briefly covers the chain rule for ordered derivatives, which underpins backpropagation, focusing on creating a simpler version that can be easily implemented in computer code for neural network users.

Training deep multilayer neural networks was challenging, but new algorithms have demonstrated that deeper networks outperform shallower ones. Research highlights that traditional gradient descent from random initialization struggles with deep networks due to issues with activation functions like the logistic sigmoid, which can lead to

saturation in hidden layers. This study [7] proposes a new initialization scheme and explores how less saturating activation functions can improve training efficiency and convergence speed.

The author of this web article [8] used the MNIST dataset for ANN, which consists of 70,000 handwritten images, each 28x28 pixels, with each pixel representing an intensity value from 0 (white) to 255 (black). The dataset is split into 60,000 training images and 10,000 testing images. The process includes importing libraries, loading and preprocessing data, creating and compiling the model, training, and evaluating it, ultimately achieving successful value prediction.

## III. METHODOLOGY

### A. DATASET INFORMATION

The dataset utilized in this study is a subset of the MNIST handwritten digits dataset, comprising 42,000 instances. Each instance is represented by 785 columns, where 784 columns correspond to the pixel values of a $28 \times 28$ grayscale image, and the remaining column indicates the class label. This dataset includes 10 classes, representing the digits from 0 through 9.

The distribution of digit images in the dataset is illustrated in Figure 3. Among the digits, digit 1 has the highest number of images, totaling 4,684, while digit 5 has the fewest, with 3,795 images. The other digits each have approximately 4,100 images.

### B. PREPROCESSING OF DATASET

Preprocessing the dataset is crucial to ensure effective model performance. In this study, several preprocessing steps were performed.

- **Handling Missing Values**: No missing values were found in the dataset, eliminating the need for imputation techniques.
- **Handling Categorical Features**: There were no any categorical features in the dataset.
- **Normalizing the data**: As the dataset contains pixel values, the data were normalized by dividing data by 255.
- **Dataset Splitting**: The dataset was split into training and testing sets with 1000 data in testing dataset. This split ensures that a majority of the data was used for training the model and a portion of the data is used to evaluate the model.

### C. WORKING PRINCIPLE

#### 1) Neurons

Neurons serve as the core components of an ANN, inspired by the biological neurons found in the human brain that handle and transmit information. In an ANN, each neuron takes in input data, performs a mathematical computation, and generates an output. This output is then forwarded to the subsequent layer of neurons within the network.

Every neuron is associated with weights for its inputs, which signify the significance of each input. During the training process, these weights are adjusted to reduce the prediction error of the network. Additionally, neurons use an activation function to introduce non-linearity into the model, enabling the network to identify more complex patterns. Common activation functions include ReLU (Rectified Linear Unit), Sigmoid, and Tanh (Hyperbolic Tangent).

#### 2) Layers

In an ANN, layers are collections of neurons arranged to process information in a sequential manner. Each layer modifies the data it receives and passes the transformed result to the subsequent layer. The primary types of layers in an ANN are:

- **Input Layer**: This is the initial layer that receives raw data, such as images or text, and forwards it to the next layer without altering it. Each neuron in this layer represents a feature of the input; for example, in a digit dataset, each neuron might correspond to a pixel value.
- **Hidden Layer**: Positioned between the input and output layers, hidden layers perform the core processing tasks. They convert the input data into a form that the network can use for decision-making or predictions. Each hidden layer contains numerous neurons, each of which applies a mathematical function to its input. Multiple hidden layers enable the network to learn more intricate patterns.
- **Output Layer**: This is the final layer that generates the network's output. It produces the end result based on the data processed through the hidden layers.

#### 3) Weights and Biases

Weights and biases are essential parameters in an ANN that enable the network to learn and make precise predictions. Weights determine the significance of each input a neuron receives. During training, the network modifies these weights to reduce prediction errors. Biases, on the other hand, are additional values added to a neuron's input, allowing for adjustments in the activation function to better align with the data.

Think of weights as the controls on a stereo system that adjust the volume of different sound frequencies. Modifying these controls changes the

sound output. Similarly, adjusting weights in a neural network alters how the input data is processed and affects the final output. Biases function like baseline adjustments, ensuring that the network can produce accurate outputs even when the input is zero, thus providing greater flexibility.

### 4) Activation Functions

An activation function is a vital element in a neural networks, essential for their learning and operation. It introduces non-linearity into the network, enabling it to model and learn from complex data patterns. Without activation functions, a neural network would essentially only perform linear transformations, greatly limiting its capacity to capture intricate relationships in the data. Common activation functions include:

- **Sigmoid**: Maps input values to a range between 0 and 1.
- **Hyperbolic Tangent (tanh)**: Maps inputs to a range between -1 and 1.
- **Rectified Linear Unit (ReLU)**: Outputs the input directly if positive and zero otherwise.

The choice of activation function can significantly affect the performance and convergence of a neural network, making it a critical design decision in deep learning.

- **Softmax Function**: Commonly used in the output layer for classification tasks, particularly for multi-class problems. It converts raw output scores (logits) into probabilities that sum to 1, making them interpretable as class probabilities. It is defined by Equation 11. The class with the highest probability is typically chosen as the predicted class. This function aids in understanding the model's confidence in its predictions and supports decision-making.
- **Rectified Linear Unit (ReLU)**: A widely used activation function in neural networks, especially in deep learning models. Defined as Equation 3, it outputs the input directly if positive, and zero otherwise. This simplicity makes ReLU computationally efficient, facilitating faster training compared to other activation functions like sigmoid or tanh. ReLU helps avoid the vanishing gradient problem by not saturating for positive values, although it can encounter issues such as "dying ReLUs," where neurons become inactive due to negative weights.
- **Sigmoid Function**: Particularly useful for binary classification, this function maps any real-valued input to a value between 0 and 1, which is ideal for producing probabilistic outputs. Its smooth gradient assists in optimization during backpropagation, though it can face issues such as the vanishing gradient problem, which may slow down training. The Equation 1 is mathematical expression of sigmoid function.
- **Hyperbolic Tangent (tanh)**: Commonly used in hidden layers, this function maps inputs to a range between -1 and 1 and is defined by Equation 5. Its symmetric output around zero can improve learning efficiency by balancing positive and negative inputs, often resulting in better performance than sigmoid in hidden layers. However, like sigmoid, tanh can also experience the vanishing gradient problem.

### 5) Xavier Initialization

Xavier initialization, also known as Glorot initialization, is a technique introduced by Xavier Glorot and Yoshua Bengio in 2010 for setting the initial weights of a neural network to enhance training convergence. The method seeks to maintain the gradient scale uniformly across all network layers, addressing problems like vanishing or exploding gradients.

The core principle of Xavier initialization is to initialize weights so that the variance of activations and gradients stays consistent throughout the network. This is accomplished by initializing the weights based on a specific distribution. By employing Xavier initialization, networks, especially deep ones, are more likely to converge more quickly and perform more effectively due to the balanced gradient scaling. The Xavier initialization formula is provided in Equation 10.

### 6) Kaiming Initialization

Kaiming initialization, also known as He initialization, is a technique introduced by Kaiming He in 2015. It was specifically designed for neural networks that use the ReLU (Rectified Linear Unit) activation function. The aim of this initialization method is to address issues related to vanishing or exploding gradients, particularly in deep networks, by preserving the variance of activations across layers.

Kaiming initialization accomplishes this by adjusting the initial weights based on the number of input units to each neuron, thereby maintaining a consistent variance of activations throughout the network. This approach helps to ensure that signals are neither amplified nor diminished as they propagate through the layers, which is critical for effective training. The Kaiming initialization formula is provided in Equation 9.

### 7) Forward Propagation

Forward propagation in an ANN refers to the method of passing input data through the network to produce an output. This process starts when input features are introduced to the input layer. These inputs are then adjusted by weights and biases, and passed through activation functions in each layer. The output from each layer becomes the input for the next layer, continuing this way until the final layer generates the network's prediction or result. This output is then compared to the actual target value to calculate the error or loss. Forward propagation is essential for evaluating the network's performance and helps guide the learning process through backward propagation, where the error is used to adjust weights and biases to enhance accuracy.

### 8) Backward Propagation

Backward propagation, or backpropagation, is a crucial process in training an Artificial Neural Network (ANN) to minimize the difference between the predicted output and the actual target. This process begins after forward propagation and involves calculating how much each weight and bias in the network contributed to the error. This is done by applying the chain rule of calculus to trace the error back through the network.

In backward propagation, the error is first calculated at the output layer by comparing the predicted result to the true target. This error is then passed backward through each layer of the network. For each layer, the gradients of the loss function with respect to the weights and biases are computed. These gradients show how changes to each weight and bias will impact the overall error.

The computed gradients are used to adjust the weights and biases using an optimization algorithm, such as Gradient Descent. This iterative process updates the parameters to reduce the error and improve the network's performance. Backward propagation continues until the network achieves satisfactory performance or meets convergence criteria.

### 9) Training loss

In an ANN, training loss or error is a critical measure used to evaluate the network's performance during the training process. It quantifies the discrepancy between the network's predictions and the actual target values for a given set of training data. The training loss is calculated by applying a loss function, which varies depending on the type of problem—such as mean squared error for regression tasks or cross-entropy loss for classification problems.

The primary goal of training is to minimize this loss, effectively making the network's predictions as close to the true values as possible. During training, the network's parameters—weights and biases—are adjusted iteratively through a process called optimization, typically using algorithms like Gradient Descent. These adjustments are guided by the gradients of the loss function, which indicate how each parameter should be modified to reduce the overall error.

Monitoring the training loss provides insights into how well the network is learning and whether it is improving over time. A decreasing training loss generally indicates that the network is learning to make better predictions. However, it is important to balance minimizing training loss with avoiding overfitting, where the network becomes too specialized to the training data and performs poorly on unseen data. Thus, evaluating training loss alongside other metrics and validation performance helps ensure that the network generalizes well to new, unseen examples.

## *D. SOME PROBLEMS OF DEEP NEURAL NETWORKS*

Deep neural networks, while powerful, are prone to several inherent challenges that can impede their performance. Three of the most common issues encountered are the vanishing gradient problem, underfitting, and overfitting.

### 1) Vanishing Gradient Problem

The vanishing gradient problem is a significant challenge in training deep neural networks, particularly when using gradient-based optimization methods like backpropagation. As the gradients are propagated backward through the network to update the weights, they can diminish exponentially, especially in networks with many layers. This phenomenon occurs because each layer's gradient is multiplied by the derivative of the activation function of the previous layer. If the activation function's derivative is small, the gradients shrink as they move toward the input layers.

When the gradients become too small, the weights of the earlier layers in the network receive negligible updates during training. This causes these layers to learn very slowly, or not at all, effectively preventing the network from converging to a good solution. The vanishing gradient problem is particularly prevalent in deep networks that use activation functions like the sigmoid or tanh, which have derivatives less than 1 for most of their input range.

### 2) Underfitting

Underfitting occurs when a neural network is too simple to capture the underlying patterns in the data. This can happen when the model lacks sufficient capacity, such as having too few layers or neurons, or when the training process is insufficient. An underfit model performs poorly on both the training and test datasets because it fails to learn the relevant features, leading to high bias and low variance.

### 3) Overfitting

Overfitting occurs when a neural network becomes too complex and starts to capture noise and irrelevant patterns in the training data. An overfit model performs exceptionally well on the training data but fails to generalize to new, unseen data, resulting in high variance and low bias.

## *E. MITIGATING PROBLEMS IN DEEP NEURAL NETWORKS*

Overcoming the challenges of deep neural networks, such as vanishing gradients, underfitting, and overfitting, is crucial for building effective models. Various strategies have been developed to address these issues, enabling more reliable and accurate learning.

### 1) Addressing the Vanishing Gradient Problem

To mitigate the vanishing gradient problem, several techniques can be employed:

- **ReLU Activation Function:** ReLU (Rectified Linear Unit) and its variants (e.g., Leaky ReLU, Parametric ReLU) are commonly used because they do not saturate like the sigmoid or tanh functions, helping to preserve gradients during backpropagation.
- **Xavier and Kaiming Initialization:** These initialization methods are designed to set the initial weights of the network such that the variance of activations and gradients is maintained throughout the network, reducing the likelihood of vanishing gradients.
- **Batch Normalization:** This technique normalizes the input to each layer, stabilizing and accelerating the training process. By maintaining a consistent scale of inputs, batch normalization helps to prevent gradients from vanishing or exploding.
- **Gradient Clipping:** This involves capping the gradients to a maximum value to prevent them from becoming too small or too large. While more commonly used to address exploding gradients, it can also help mitigate issues with vanishing gradients.

### 2) Overcoming Underfitting

Underfitting can be mitigated by increasing the model's complexity or improving the training process:

- **Increasing Model Complexity:** Adding more layers or neurons can give the model greater capacity to learn complex patterns in the data.
- **Extended Training:** Allowing the model to train for more epochs gives it more time to learn from the data. However, this should be done with care to avoid overfitting.
- **Reducing Regularization:** While regularization helps prevent overfitting, too much can lead to underfitting. Reducing the strength of L1 or L2 regularization might improve the model's performance.
- **Feature Engineering:** Improving the quality and relevance of input features can help the model better capture underlying patterns in the data.

### 3) Preventing Overfitting

To prevent overfitting, several regularization techniques can be employed:

- **Dropout:** Dropout layers randomly deactivate a portion of neurons during training, preventing the model from becoming overly reliant on specific neurons and promoting generalization.
- **Early Stopping:** Monitoring the model's performance on a validation set and halting training when performance begins to degrade helps avoid overfitting to the training data.
- **Regularization (L1/L2):** Regularization adds a penalty to the loss function based on the magnitude of the weights, discouraging the model from developing overly complex or large weights.
- **Data Augmentation:** Techniques such as rotation, flipping, and scaling artificially increase the size and diversity of the training set, helping the model to generalize better to new data.

## *F. ALGORITHM OF ANN*

**Step 1:** Import the dataset.
**Step 2:** Preprocess the data (e.g., normalization, handling missing values).
**Step 3:** Split the dataset into training and testing sets.
**Step 4:** Construct the neural network.

**4.1** Select the number of hidden layers and the number of nodes in each hidden layer.
**4.2** Define the activation functions for the hidden layers and the output layer.

**Step 5:** Initialize the weights and biases for the input and hidden layers.

**Step 6: Forward Propagation:**

**6.1** Feed the training data into the network.
**6.2** Compute the output of the network.
**6.3** Calculate the error between the predicted output and the ground truth.

**Step 7: Backpropagation (if error has not converged):**

**7.1** Compute the partial derivatives of the loss with respect to the weights and biases.
**7.2** Update the weights and biases using the computed derivatives.

**Step 8:** Repeat Steps 6 and 7 until the error converges or the maximum number of iterations is reached.
**Step 9:** Stop the training process.
**Step 10:** Test the neural network using the testing dataset.
**Step 11:** Calculate and report the accuracy of the network.

### G. BLOCK DIAGRAM

The Figure 2 illustrates the block diagram of an Artificial Neural Network (ANN). This network is composed of an input layer, one or more hidden layers, and an output layer, each containing a different number of nodes. Nodes in each layer are interconnected, with the outputs from one layer serving as inputs to the next. The output and hidden layers utilize activation functions, which process the outputs of their respective layers.

Initially, the weights and biases for the input and hidden layers are randomly initialized. During forward propagation, the input data is passed through the network, layer by layer, until the final output is produced by the output layer. This output is then compared to the ground truth value to calculate the training loss (error). If the error has not converged to a satisfactory level, backpropagation is initiated. During backpropagation, the partial derivatives of the error with respect to each weight and bias are computed, and these values are used to update the weights and biases. This process is repeated until the error converges or the maximum number of iterations is reached, at which point the training process stops.

After training, the neural network is tested using a different dataset, and the accuracy of the network is calculated.

### H. FLOWCHART

The flowchart of the Artificial Neural Network (ANN) is presented in Figure 1 The process begins with importing the dataset, followed by preprocessing the data and splitting it into training and testing sets. The neural network is then constructed, where the number of hidden layers and the number of nodes within each layer are specified. Next, the activation functions for the hidden layers and the output layer are defined, and the weights and biases for the input and hidden layers are initialized.

During forward propagation, the training data is fed into the network, and the network produces an output. The error between the predicted output and the actual output is calculated. If the error has not converged, backpropagation is performed. In backpropagation, the partial derivatives of the loss with respect to the weights and biases of each layer are computed, and these values are used to update the weights and biases.

This process is repeated until the error converges or the maximum number of iterations is reached, at which point the training process stops. Finally, the trained ANN is tested using the testing dataset, and the accuracy of the network is calculated.

## IV. IMPLEMENTATION DETAILS AND RESULTS

To prepare the data for training, the pixel values were normalized by dividing each by 255, bringing them into the range [0, 1]. The class labels were then one-hot encoded, transforming each label into a binary vector of length 10. For instance, a label of 6 was converted into a vector where the sixth element is 1, and all other elements are 0.

The dataset was shuffled to ensure randomness, after which it was split into a training set and a test set. The first 1,000 instances were reserved for testing, while the remaining 41,000 instances were used for training the model.

We aimed to evaluate the impact of weight and bias initialization on the learning process. To do this, we constructed a simple neural network with an input layer containing 728 neurons, a hidden layer with 10 neurons, and an output layer with 10 neurons. The input and hidden layers utilized ReLU activations, while the output layer employed a Softmax activation function. The cross-entropy loss function was used for training. In this experiment, weights and biases were initialized by sampling from a Weibull distribution with the shape parameter set to 1. The accuracy versus epoch curve for this configuration is shown in Figure 4. Additionally, inference results displaying the images, true labels, predicted labels, prediction probabilities, overall accuracy, and cross-entropy loss on the test set are presented in Figure 5. Upon examining the results, we observed that the network's accuracy after 10 epochs was only 0.22, which is quite poor. This low accuracy was also evident in the model's performance on the test set.

Given the poor accuracy, we hypothesized that the simplicity of the model might be the cause. To

test this, we increased the model's complexity by having 2 hidden layers with 30 neurons in each hidden layer. The corresponding results are shown in Figure 6 and Figure 7. However, even with the increased complexity, the accuracy remained stagnant from epoch 1 through 10, ultimately achieving an accuracy score of just 0.11, which is even worse than the simpler model. This led us to conclude that the model's complexity was not the issue.

Next, we considered whether the activation function might be contributing to the poor performance. We returned to the simpler network with one hidden layer of 10 neurons, but this time, we experimented with Tanh and Sigmoid activations. The corresponding figures are shown in Figure 8, Figure 9, Figure 10, and Figure 11. In both cases, the accuracy remained constant throughout the training process with no improvements, and the models performed poorly on the test set, achieving accuracies of only 0.12 and 0.10 with Tanh and Sigmoid activations, respectively.

Based on these observations, we concluded that the poor performance was most likely due to the weight initialization method.

We then applied Xavier initialization to the simple model architecture, with the results presented in Figure 12 and Figure 13. Immediately, we observed an improvement in the model's performance, with an accuracy of 0.54 achieved by epoch 3, and 0.3 after 10 epochs. This promising outcome encouraged us to explore further modifications to enhance the model's performance.

Next, we increased the model's complexity by adding a second hidden layer with 30 neurons each. The results, shown in Figure 14 and Figure 15, revealed a steep decline in accuracy until epoch 2, after which it plateaued at 0.11, indicating this approach was ineffective. We then opted to increase the network's depth by using a single hidden layer with 128 neurons. The results, depicted in Figure 16 and Figure 17, showed an initial accuracy of 0.84 at epoch 1, which dropped slightly to 0.76 by epoch 10. Despite this decline, the performance was still superior to the previous attempts.

Building on this, we increased both the depth and width of the architecture by employing two hidden layers with 128 neurons each. The corresponding results are shown in Figure 18 and Figure 19, but the accuracy once again remained constant throughout training, achieving a value of just 0.11.

To further investigate, we returned to the previous architecture of one hidden layer with 128 neurons and switched the activation function from ReLU to Tanh. As shown in Figure 20 and Figure 21, this change resulted in better performance on both the training and testing data, with an accuracy of 0.82 on the test set, the highest achieved thus far. Following this, we experimented with a Sigmoid activation function, with results depicted in Figure 22 and Figure 23. Remarkably, this configuration achieved a training accuracy of 1.0 and a test accuracy of 0.98, demonstrating exceptional performance.

Curious about the specific test cases where the model made incorrect predictions, we examined these cases, as shown in Figure 24. Notably, the incorrectly predicted images were challenging even for human interpretation. Given the model's perfect accuracy on the test set, we suspected overfitting and introduced a dropout layer with a rate of 0.5 after the input and hidden layers. The results, displayed in Figure 25 and Figure 26, showed a decrease in accuracy to 0.56 on the training set and 0.58 on the test set. We then adjusted the dropout rate to 0.2, with results shown in Figure 27 and Figure 28, where accuracy improved to 0.85 on both the training and test sets. Since the accuracy before adding the dropout layer was higher, we concluded that the model was not overfitting.

Next, we experimented with Kaiming initialization using the same model architectures and configurations as before. The results for the simple model architecture are presented in Figure 29 and Figure 30. Similar to the results with Xavier initialization, we observed an improvement in the model's performance, achieving an accuracy of 0.45 after 10 epochs. Encouraged by this initial success, we explored different configurations to further optimize performance.

We increased the model's complexity by adding a second hidden layer with 30 neurons each. The corresponding results are shown in Figure 31 and Figure 32. However, the accuracy remained constant at 0.11 throughout the epochs, indicating no significant improvement with the added complexity.

To evaluate the impact of increasing depth, we utilized a single hidden layer with 128 neurons. The results, illustrated in Figure 33 and Figure 34, showed an accuracy of 0.81 on the test set, marking a notable improvement compared to previous configurations using Kaiming initialization.

Next, we experimented with increasing both the depth and width by employing two hidden layers with 128 neurons each. As shown in Figure 35 and Figure 36, the accuracy was not satisfactory, plateauing at 0.11 from epoch 6 through epoch 10.

We then reverted to the simpler architecture with one hidden layer of 128 neurons and switched the activation function from ReLU to Tanh. The results, shown in Figure 37 and Figure 38, demonstrated improved performance with an accuracy of 0.89 on the test set. Finally, we applied the Sigmoid

activation function with the same architecture, as depicted in Figure 39 and Figure 40. This setup achieved perfect training accuracy of 1.0 and a test accuracy of 0.97. We also examined the cases where the model made incorrect predictions, shown in Figure 41. These misclassifications were challenging to distinguish even for humans.

Concerned about potential overfitting, we added a dropout layer with a rate of 0.5 after the input and hidden layers. The results, shown in Figure 42 and Figure 43, revealed a reduction in accuracy to 0.56 on both the training and testing sets. We then adjusted the dropout rate to 0.2, as illustrated in Figure 44 and Figure 45, which improved the accuracy to 0.86 on the training set and 0.84 on the testing set. Since the accuracy was higher before introducing the dropout layer, we concluded that the model was not overfitting.

## V. DISCUSSION AND ANALYSIS

In this section, we analyze the results obtained from the experiments conducted with different weight initialization techniques—Weibull, Xavier, and Kaiming—and the impact of various model architectures and activation functions.

### A. WEIBULL INITIALIZATION

The experiments using Weibull initialization consistently resulted in poor model performance, even when the model architecture was made more complex. The best accuracy achieved was only 0.22 with a simple model, and increasing the number of hidden layers or neurons did not improve performance.

#### 1) Fundamental Issues with Weibull Initialization

The Weibull distribution, while flexible, may not be well-suited for initializing weights in deep neural networks, especially when the shape parameter is set to 1, which corresponds to an exponential distribution. This could lead to weights that are too small or too large, causing gradients to either vanish or explode during backpropagation. Additionally, Weibull initialization may introduce a bias in weight distribution that affects the convergence of the network, leading to the observed poor performance across different architectures and activation functions.

### B. XAVIER AND KAIMING INITIALIZATION

Both Xavier and Kaiming initialization techniques showed significantly better performance compared to Weibull initialization. The key difference between these techniques lies in how they handle the variance of weights depending on the activation function used in the network.

#### 1) Xavier Initialization

Xavier initialization is designed to keep the variance of the activations and gradients roughly the same across layers. It worked well with the ReLU activation in our experiments, as evidenced by the improvement in accuracy to 0.54 after 3 epochs and the eventual stabilization around 0.76 with a deeper network. However, it failed to deliver good results when the network complexity was increased with additional hidden layers, suggesting that it may not be as effective in deeper networks without careful consideration of the architecture.

#### 2) Kaiming Initialization

Kaiming initialization, tailored for ReLU activations, consistently performed well across different configurations, showing an improvement in accuracy to 0.81 with a single hidden layer of 128 neurons. The method's ability to preserve the variance of gradients even in deep networks contributed to its success. However, similar to Xavier, Kaiming initialization struggled when the network was made more complex with two hidden layers, indicating that the initialization alone is not sufficient to ensure good performance in deeper networks.

### C. IMPACT OF NETWORK ARCHITECTURE

The experiments demonstrated that a network with one hidden layer of 128 neurons performed the best, achieving high accuracy with both Xavier and Kaiming initialization. In contrast, networks with two hidden layers, regardless of the number of neurons, performed poorly.

#### 1) Why One Hidden Layer with 128 Neurons Worked Best

A single hidden layer with 128 neurons likely struck the right balance between model complexity and capacity to generalize. This architecture was complex enough to capture the underlying patterns in the data but not so complex that it introduced excessive parameters, leading to overfitting or vanishing gradients. Additionally, this configuration allowed the initialization methods to effectively propagate gradients, facilitating better learning.

#### 2) Poor Performance with Two Hidden Layers

The poor performance of networks with two hidden layers, particularly with 30 or 128 neurons each, can be attributed to the increased complexity, which may have caused vanishing gradients or difficulty in learning the correct patterns due to the initialization method's limitations. The increased depth might have also led to more pronounced issues with weight distribution, causing the network to plateau at a low accuracy level early in training.

### D. EFFECT OF DROPOUT

Introducing dropout layers was intended to prevent overfitting, especially in cases where the model achieved 100% accuracy on the training set.

#### 1) Dropout and Model Overfitting

Surprisingly, the addition of dropout layers led to a decrease in accuracy, both on the training and test sets, suggesting that the model was not overfitting as initially suspected. The reason behind this could be that the high test accuracy was not due to memorization but rather an effective learning of the data distribution. Dropout may have inadvertently disrupted this process by randomly removing important neurons during training, thus reducing the model's capacity to learn effectively.

#### 2) Model Performance with Dropout Rate Adjustment

When the dropout rate was adjusted from 0.5 to 0.2, there was a slight improvement in accuracy, but it still did not reach the level achieved without dropout. This further supports the idea that the model, even with a perfect training accuracy, was not overfitting but was instead robustly learning the data distribution. Thus, dropout may not have been necessary in this specific context.

### E. PERFORMANCE OF ACTIVATION FUNCTIONS

Among the activation functions tested—ReLU, Tanh, and Sigmoid—Sigmoid achieved the best performance, particularly with a perfect training accuracy of 1.0 and a test accuracy of 0.98 and 0.97 when using Xavier and Kaiming initialization respectively.

#### 1) Why Sigmoid Performed Best

Sigmoid activation functions produce outputs in the range (0, 1), which can be beneficial for binary classification and when used with a softmax output layer in a multi-class classification problem. This characteristic helps the model in normalizing the output values and ensuring stable gradients during training, which might have contributed to better performance compared to ReLU and Tanh. ReLU, while effective in avoiding vanishing gradients, can lead to dying neurons if not handled properly, especially in deeper networks where the initialization of weights can significantly impact performance. Tanh, on the other hand, can suffer from vanishing gradients in deeper networks due to its output range of (-1, 1), which can cause issues in maintaining gradient flow during backpropagation.

In the context of Kaiming initialization, which is designed to work well with ReLU activation functions, the Sigmoid function still outperformed ReLU. This could be attributed to the fact that while Kaiming initialization is intended to address the vanishing gradients problem with ReLU, it may not fully mitigate the issue of dying neurons or the saturation of activations. Sigmoid's range of (0, 1) potentially provided more stable and non-saturated gradients, allowing for more effective learning and weight adjustments. This suggests that for this specific task and architecture, Sigmoid's properties aligned better with the overall training dynamics, even when Kaiming initialization was applied.

### F. OVERALL INSIGHTS

The experiments highlighted the importance of choosing the appropriate initialization method, network architecture, and activation functions based on the specific characteristics of the model and dataset. While both Xavier and Kaiming initialization showed promising results, careful tuning of the network's architecture and activation functions is crucial for achieving optimal performance. The simpler architecture with one hidden layer of 128 neurons outperformed more complex configurations, and Sigmoid activation provided the best results due to its suitability for the task at hand. Dropout, though a common regularization technique, did not improve the model's performance and might have disrupted effective learning in this specific scenario.

## VI. CONCLUSION

In this study, we investigated the impact of various weight initialization techniques and activation functions on the performance of neural networks using a subset of the MNIST dataset. Our findings revealed that Weibull initialization consistently led to poor performance across different model architectures and activation functions. The observed low accuracy suggests that Weibull initialization may not be suitable for this type of neural network setup, potentially due to its lack of effective scaling and variance control in the weights.

In contrast, Xavier initialization significantly improved model performance. It showed considerable gains in accuracy, particularly when applied to a network with a single hidden layer of 128 neurons. However, increasing the model complexity by adding more hidden layers or neurons did not consistently enhance performance and sometimes led to performance plateaus. This indicates that the benefits of Xavier initialization are best realized in simpler architectures.

Kaiming initialization, akin to Xavier, also enhanced performance and was notably effective with a single hidden layer of 128 neurons. Despite being

designed primarily for use with ReLU activation, Kaiming initialization also performed well with Sigmoid and Tanh activations. Among these, the Sigmoid activation function achieved the highest accuracy, which underscores its effectiveness in stabilizing gradients and providing precise weight adjustments. The range of outputs (0, 1) from the Sigmoid function likely contributed to its superior performance compared to ReLU and Tanh, which can encounter issues such as vanishing gradients and saturation.

The consistent outperformance of the model with a single hidden layer of 128 neurons compared to models with multiple hidden layers or fewer neurons suggests that increasing network depth or width does not always lead to better results. This finding indicates that the complexity of the network needs to be carefully balanced with the task requirements. Additionally, the introduction of dropout layers, intended to mitigate overfitting, did not significantly improve generalization and sometimes reduced overall accuracy. This implies that the model was not overfitting in the traditional sense and highlights the need for careful tuning of dropout rates based on the specific architecture and task.

Future research may investigate additional methods and strategies to advance these findings. Investigating alternative initialization methods, such as orthogonal initialization, could provide insights into whether these techniques offer improved performance compared to Weibull and existing methods. Additionally, exploring advanced regularization strategies beyond dropout, such as L1/L2 regularization or batch normalization, might help in controlling overfitting and enhancing model generalization. Testing advanced activation functions like Swish or Mish could also be valuable, as they have shown promise in various neural network tasks. Furthermore, examining different network architectures, such as residual networks or attention mechanisms, may offer better performance or address issues observed with deeper networks. Finally, conducting extensive hyperparameter tuning, including learning rate schedules, batch sizes, and optimization algorithms, could help identify configurations that maximize model accuracy and efficiency.

By pursuing these directions, we can deepen our understanding of neural network training dynamics and work towards improving model performance for complex tasks.

# APPENDIX

## A. EQUATIONS

## A. ACTIVATION FUNCTIONS

The sigmoid function is given by:

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

The derivative of the sigmoid function is given by:

$$\sigma'(z) = \frac{e^{-z}}{(1+e^{-z})^2} \tag{2}$$

The ReLU function is defined as:

$$\text{ReLU}(z) = \max(0, z) \tag{3}$$

The derivative of the ReLU function is given by:

$$\text{ReLU}'(z) = \begin{cases} 0 & \text{for } z \leq 0 \\ 1 & \text{for } z > 0 \end{cases} \tag{4}$$

The hyperbolic tangent function is given by:

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{5}$$

The derivative of the hyperbolic tangent function is given by:

$$\tanh'(z) = 1 - \tanh^2(z) \tag{6}$$

The Leaky ReLU function is defined as:

$$\text{LeakyReLU}(z) = \begin{cases} \alpha z & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases} \tag{7}$$

The derivative of the Leaky ReLU function is given by:

$$\text{LeakyReLU}'(z) = \begin{cases} \alpha & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases} \tag{8}$$

where $\alpha$ is a small constant number (e.g., $\alpha = 0.001$).

## B. KAIMING INITIALIZATION

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n}}\right) \tag{9}$$

where:

- $W$ represents the weight matrix.
- $\mathcal{N}\left(0, \sqrt{\frac{2}{n}}\right)$ denotes a normal distribution with a mean of 0 and a standard deviation of $\sqrt{\frac{2}{n}}$.
- $n$ is the number of input units in the weight matrix.

## C. XAVIER INITIALIZATION

For a layer with $n_{\text{in}}$ input neurons and $n_{\text{out}}$ output neurons, the Xavier initialization sets the weights to be drawn from a distribution with zero mean and variance:

$$\text{Var}(W) = \frac{2}{n_{\text{in}} + n_{\text{out}}} \tag{10}$$

Where $W$ represents the weights of the layer. This initialization can be implemented using either a uniform distribution or a normal distribution:

## D. DERIVATIVE OF SOFTMAX

The Softmax function is defined as:

$$f(z^i) = \frac{e^{z^i}}{\sum_{k=1}^{n} e^{z^k}} \tag{11}$$

We have two cases for the partial derivative of $A_i$ with respect to $z^j$:

1. For $i \neq j$:

$$\frac{\partial A_i}{\partial z^j} = \frac{\partial}{\partial z^j}\left(\frac{e^{z^i}}{\sum_{k=1}^{n} e^{z^k}}\right)$$

$$= \frac{e^{z^i} \cdot 0 - e^{z^i} \cdot e^{z^j}}{\left(\sum_{k=1}^{n} e^{z^k}\right)^2}$$

$$= -\frac{e^{z^i} \cdot e^{z^j}}{\left(\sum_{k=1}^{n} e^{z^k}\right)^2}$$

$$= -A_i \times A_j$$

2. For $i = j$:

$$\frac{\partial A_i}{\partial z^i} = \frac{\partial}{\partial z^i}\left(\frac{e^{z^i}}{\sum_{k=1}^{n} e^{z^k}}\right)$$

$$= \frac{e^{z^i} \cdot \left(\sum_{k=1}^{n} e^{z^k}\right) - e^{z^i} \cdot e^{z^i}}{\left(\sum_{k=1}^{n} e^{z^k}\right)^2}$$

$$= \frac{e^{z^i} \cdot \left(e^{z^i} + \sum_{k \neq i} e^{z^k}\right) - e^{z^i} \cdot e^{z^i}}{\left(e^{z^i} + \sum_{k \neq i} e^{z^k}\right)^2}$$

$$= A_i \times (1 - A_i)$$

Hence,

$$\frac{\partial A_i}{\partial z^j} = \begin{cases} -A_i A_j & \text{if } i \neq j \\ A_i(1 - A_i) & \text{if } i = j \end{cases}$$

## MATHEMATICAL DERIVATIONS

1. Considering categorical cross-entropy, the loss function $L$ is given by:

$$L = -\left[ y_i \log(A_i^1) + \sum_{k \neq i} y_k \log(A_k^1) \right] \tag{12}$$

where $y_i$ is the true label for class $i$, and $A_i^1$ is the predicted probability for class $i$.

Differentiate both sides of the categorical cross-entropy loss function $L$ with respect to $Z_1$:

$$\frac{\partial L}{\partial Z_1} = -\frac{\partial}{\partial Z_1} \left[ y_i \log(A_i^1) + \sum_{k \neq i} y_k \log(A_k^1) \right] \tag{13}$$

Expanding the differentiation, we get:

$$\frac{\partial L}{\partial Z_1} = -\left[ y_i \frac{1}{A_i^1} \frac{\partial A_i^1}{\partial Z_1} \right] - \sum_{k \neq i} y_k \frac{1}{A_k^1} \frac{\partial A_k^1}{\partial Z_1} \tag{14}$$

Substitute the derivatives of the predicted probabilities:

$$\frac{\partial A_i^1}{\partial Z_1} = A_i^1 (1 - A_i^1) \tag{15}$$

$$\frac{\partial A_k^1}{\partial Z_1} = -A_k^1 A_i^1 \text{ for } k \neq i \tag{16}$$

So,

$$\frac{\partial L}{\partial Z_1} = -y_i \cdot \frac{1}{A_i^1} \cdot A_i^1 (1 - A_i^1) - \sum_{k \neq i} y_k \cdot \frac{1}{A_k^1} \cdot (-A_k^1 A_i^1) \tag{17}$$

Simplify the expression:

$$\frac{\partial L}{\partial Z_1} = -y_i \cdot (1 - A_i^1) + A_i^1 \sum_{k \neq i} y_k \tag{18}$$

$$\frac{\partial L}{\partial Z_1} = -y_i + y_i \cdot A_i^1 + A_i^1 \sum_{k \neq i} y_k \tag{19}$$

Since $\sum_{k=1}^{2} y_k$ is the sum of probabilities and equals 1:

$$\frac{\partial L}{\partial Z_1} = A_i^1 - y_i \tag{20}$$

Thus, we have:

$$\frac{\partial L}{\partial Z_1} = \Delta Z_1 \tag{21}$$

2. To compute the gradient of the loss function with respect to $w_2$, we start with:

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial Z_2} \cdot \frac{\partial Z_2}{\partial w_2} \tag{22}$$

Given that:

$$\frac{\partial Z_2}{\partial w_2} = A_1 \tag{23}$$

We get:

$$\frac{\partial L}{\partial w_2} = \Delta Z_2 \cdot \frac{\partial (w_2 A_1 + B_2)}{\partial w_2} \tag{24}$$

$$\frac{\partial L}{\partial w_2} = \Delta Z_2 \times A_1 \tag{25}$$

For $m$ observations, the gradient becomes:

$$\frac{\partial L}{\partial w_2} = \frac{1}{m} \Delta Z_2 \times \left( \sum_{i=1}^{m} A_1^i \right)^T \tag{26}$$

where $\sum_{i=1}^{m} A_1^i$ denotes the sum of the activations $A_1$ across all observations.

3. To compute the gradient of the loss function with respect to $b_2$, we start with:

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial Z_2} \cdot \frac{\partial Z_2}{\partial b_2} \tag{27}$$

Since:

$$\frac{\partial Z_2}{\partial b_2} = 1 \tag{28}$$

We get:

$$\frac{\partial L}{\partial b_2} = \Delta Z_2 \cdot \frac{\partial (w_2 A_1 + B_2)}{\partial b_2} \tag{29}$$

$$\frac{\partial L}{\partial b_2} = \Delta Z_2 \times 1 \tag{30}$$

For $m$ observations, the gradient is:

$$\frac{\partial L}{\partial b_2} = \frac{1}{m} \sum_{i=1}^{m} \Delta Z_2^i \tag{31}$$

where $\sum_{i=1}^{m} \Delta Z_2^i$ denotes the sum of the error terms $\Delta Z_2$ across all observations.

4. Gradient of Loss with Respect to $z_1$ To find the gradient of the loss $L$ with respect to $z_1$, we use the chain rule:

$$\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial a_2} \cdot \frac{\partial a_2}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1}$$

$$= \delta z_2 \cdot \frac{\partial}{\partial a_1} (w_2 a_1 + b_2) \cdot \frac{\partial}{\partial z_1} (\text{ReLU}(z_1))$$

$$= \delta z_2 \cdot w_2 \cdot \text{ReLU}'(z_1)$$

For $m$ observations, the gradient is:

$$\Delta z_1 = \frac{\partial L}{\partial z_1} = \delta z_2 \cdot (w_2)^T \cdot \text{ReLU}'(z_1)$$

5. Gradient of Loss with Respect to $w_1$ To find the gradient of the loss $L$ with respect to $w_1$, we again use the chain rule:

$$\begin{aligned}\Delta w_1 &= \frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial w_1} \\ &= \frac{\delta z_2}{\partial w_1} \\ &= \delta z_1 \cdot \frac{\partial}{\partial w_1}(w_1 A_0 + b_1) \\ &= \delta z_1 \cdot A_0\end{aligned}$$

For $m$ observations, the gradient is:

$$\Delta w_1 = \left(\frac{1}{m}\right) \delta z_1 \cdot (A_0)^T$$

6. Gradient of Loss with Respect to $b_1$ To find the gradient of the loss $L$ with respect to $b_1$, we use:

$$\begin{aligned}\Delta b_1 &= \frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial z_2} \cdot \frac{\partial z_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{\partial z_1}{\partial b_1} \\ &= \frac{\delta z_1}{\partial b_1} \\ &= \delta z_1 \cdot \frac{\partial}{\partial b_1}(w_1 A_0 + b_1) \\ &= \delta z_1 \cdot 1\end{aligned}$$

For $m$ observations, the gradient is:

$$\frac{\partial L}{\partial b_1} = \left(\frac{1}{m}\right) \sum (\delta z_1)_i$$

## E. FIGURES

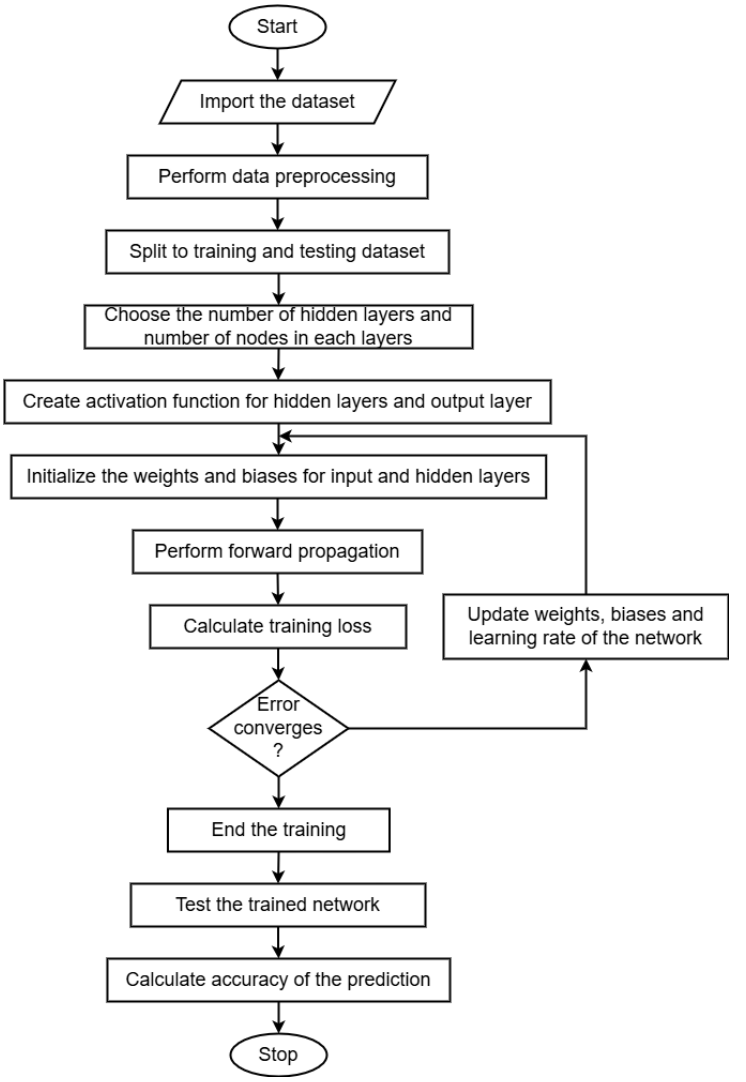


**FIGURE 1. Flowchart**

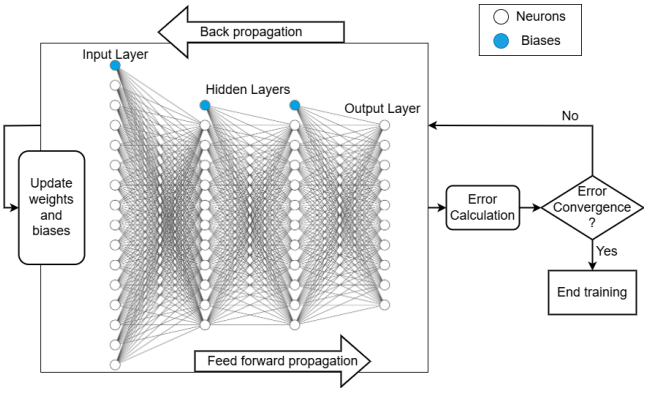


**FIGURE 2. System Block Diagram**

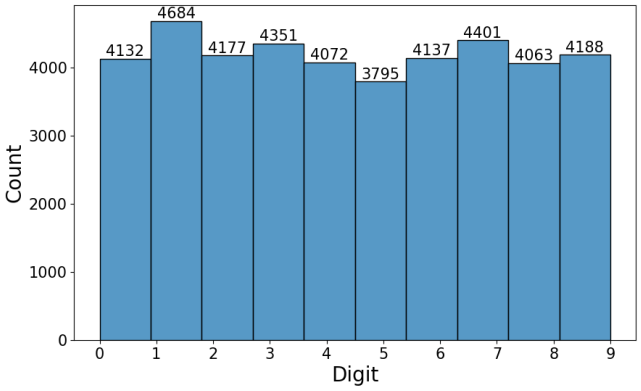


**FIGURE 3. Histogram of Digits**

### 1) Using Weibul Distribution for Weights and Bias Initialization

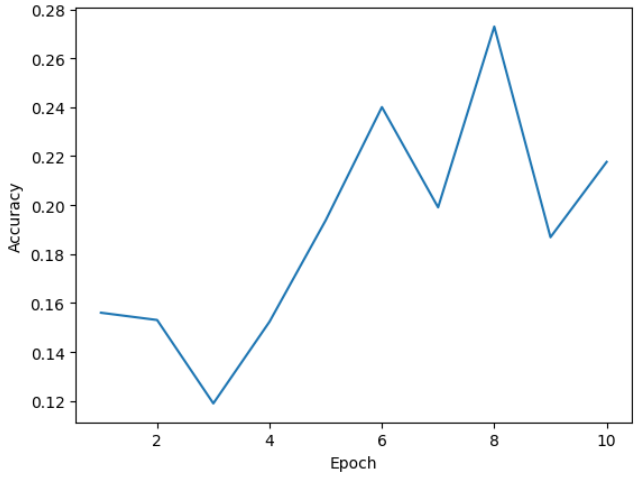


**FIGURE 4. Accuracy vs Epoch using ReLU activation, 1 hidden layer with 10 neuron**

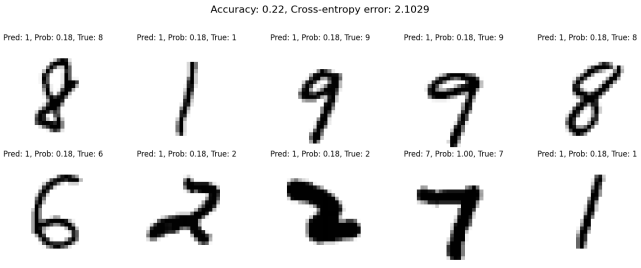


**FIGURE 5. Inference Results using ReLU activation, 1 hidden layer with 10 neuron**

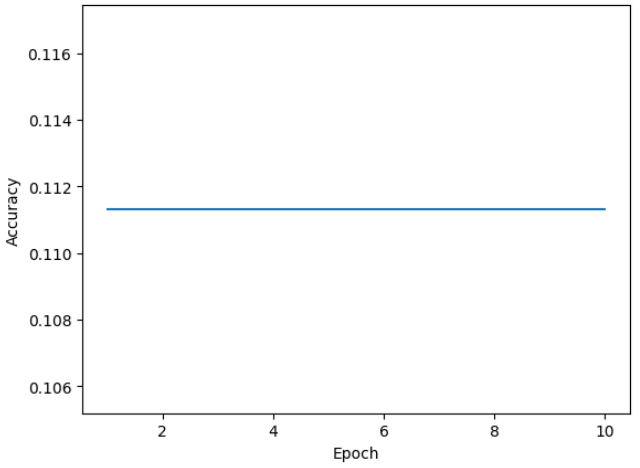


**FIGURE 6. Accuracy vs Epoch curve using ReLU activation, 2 hidden layer with 30 neuron**

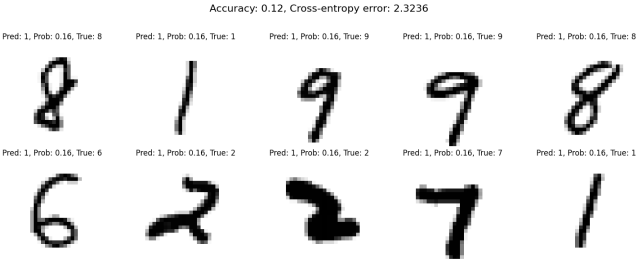


**FIGURE 7. Inference Results using ReLU activation, 1 hidden layer with 10 neuron**

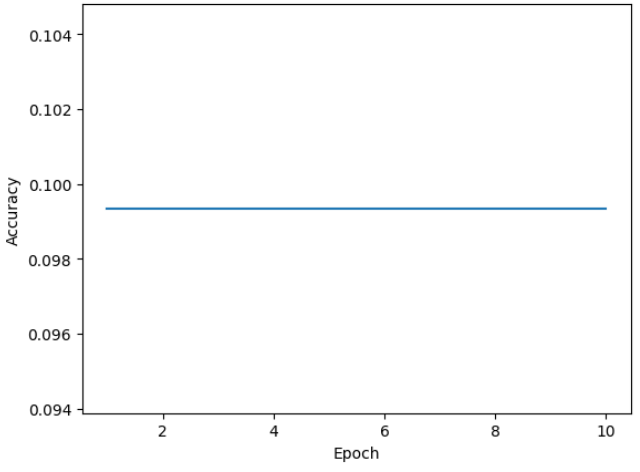


**FIGURE 8. Accuracy vs Epoch curve using tanh activation, 1 hidden layer with 10 neuron**

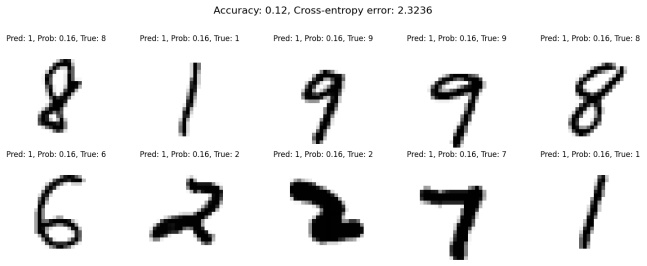


**FIGURE 9.** Inference Results using tanh activation, 1 hidden layer with 10 neuron

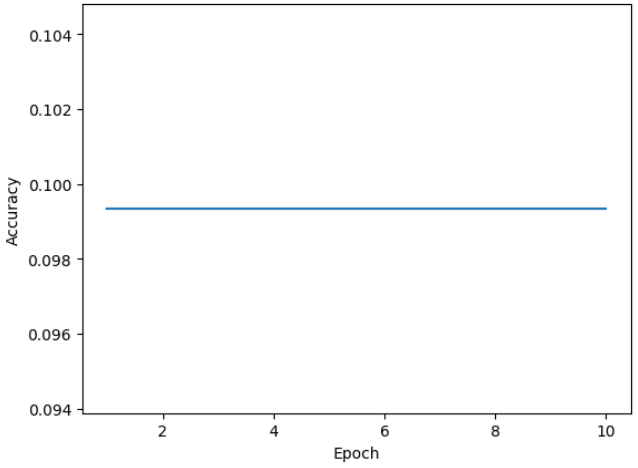


**FIGURE 10.** Accuracy vs Epoch curve using sigmoid activation, 1 hidden layer with 10 neuron

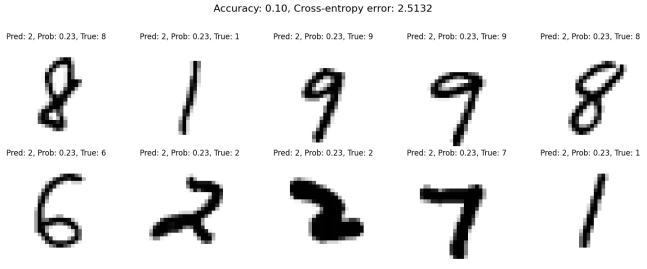


**FIGURE 11.** Inference Results using sigmoid activation, 1 hidden layer with 10 neuron

## *F. USING XAVIER INITIALIZATION FOR WEIGHTS AND BIAS*

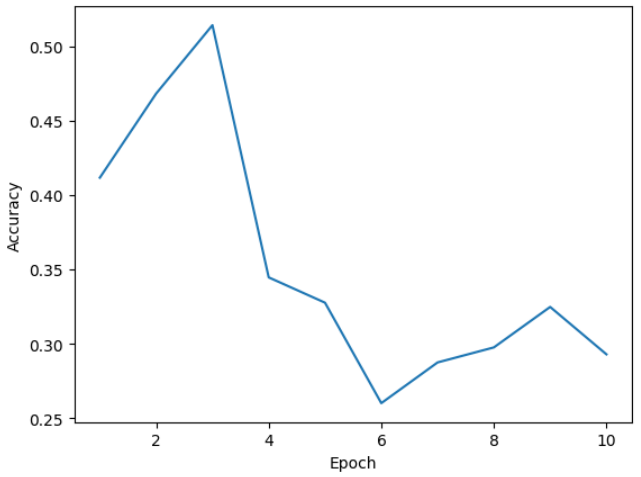


**FIGURE 12.** Accuracy vs Epoch curve using ReLU activation, 1 hidden layer with 10 neuron

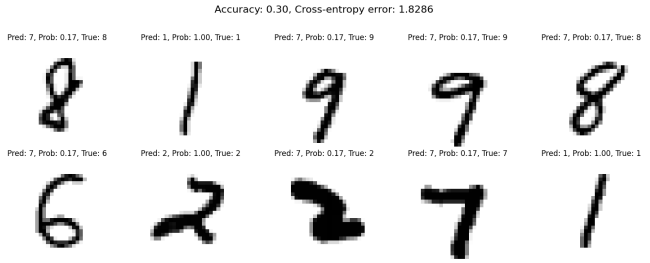


**FIGURE 13.** Inference Results using ReLU activation, 1 hidden layer with 10 neuron

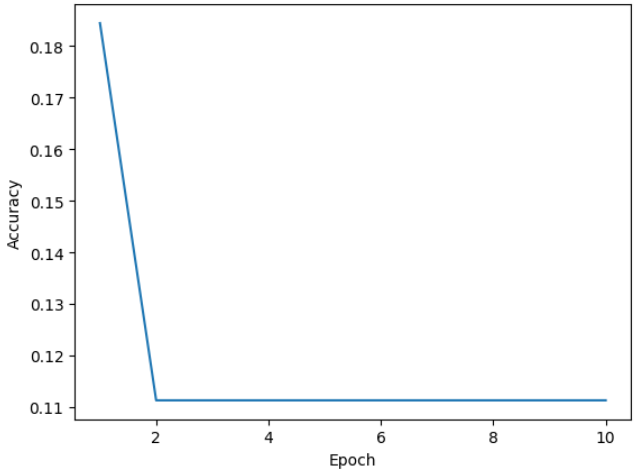


**FIGURE 14.** Accuracy vs Epoch curve using ReLU activation, 2 hidden layer with 30 neuron

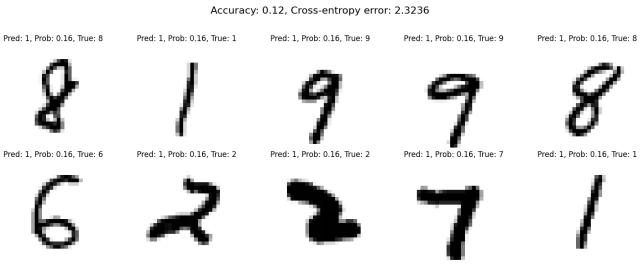


**FIGURE 15.** Inference Results using ReLU activation, 2 hidden layer with 30 neuron

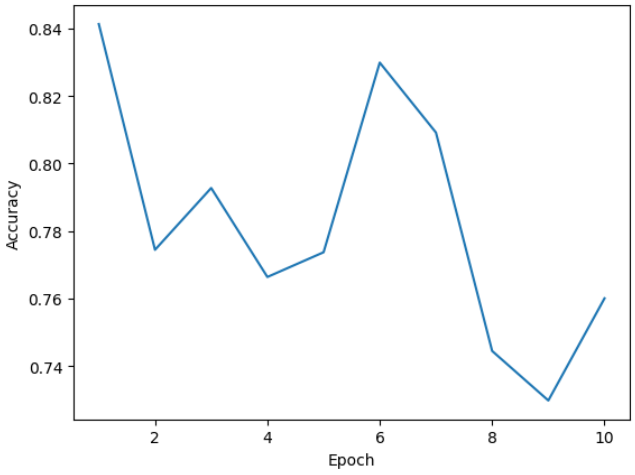


**FIGURE 16.** Accuracy vs Epoch curve using ReLU activation, 1 hidden layer with 128 neuron

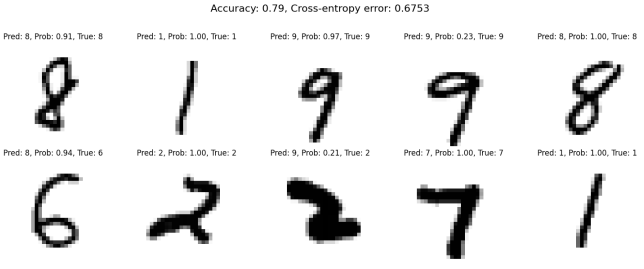


**FIGURE 17.** Inference Results using ReLU activation, 1 hidden layer with 128 neuron

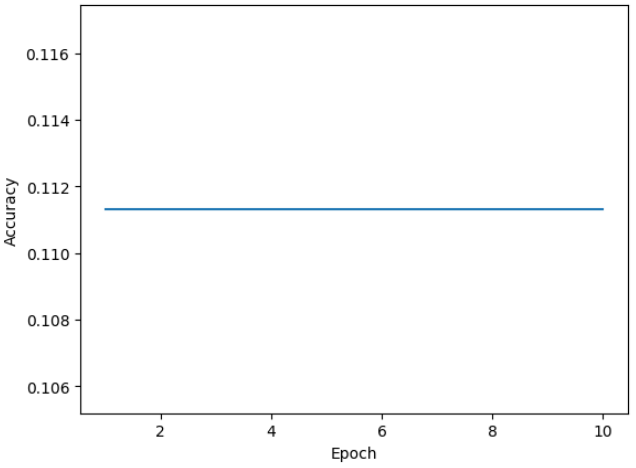


**FIGURE 18.** Accuracy vs Epoch curve using ReLU activation, 2 hidden layer with 128 neuron

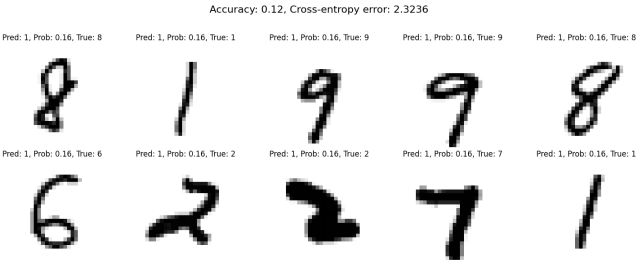


**FIGURE 19.** Inference Results using ReLU activation, 2 hidden layer with 128 neuron

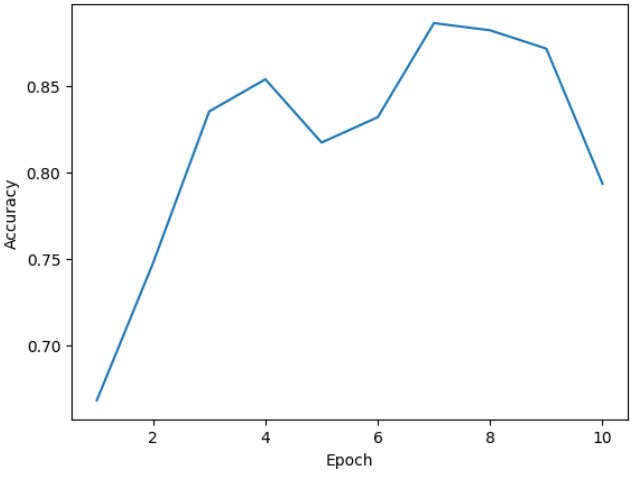


**FIGURE 20.** Accuracy vs Epoch curve using tanh activation, 1 hidden layer with 128 neuron

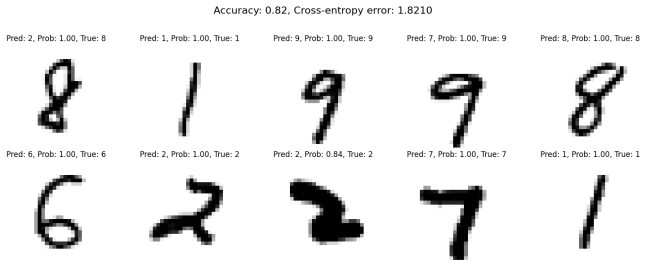


**FIGURE 21.** **Inference Results using tanh activation, 1 hidden layer with 128 neuron**

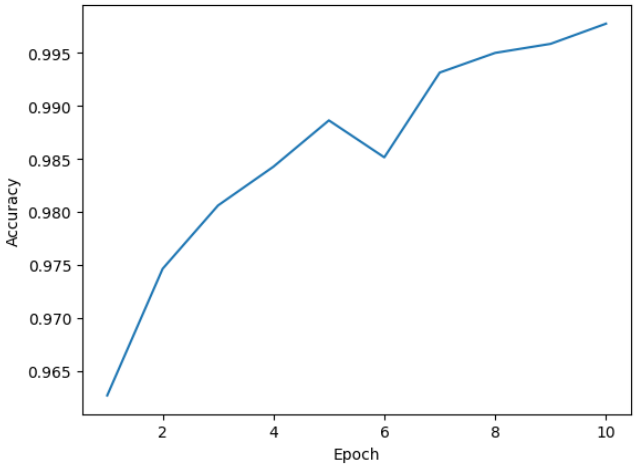


**FIGURE 22.** **Accuracy vs Epoch curve using sigmoid activation, 1 hidden layer with 128 neuron**

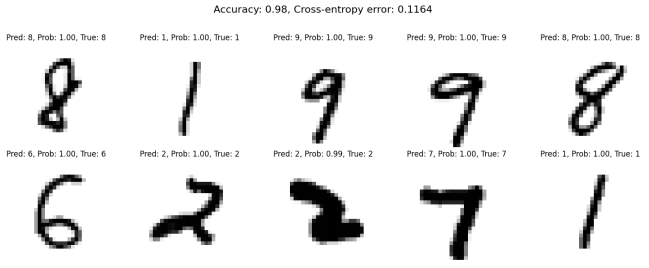


**FIGURE 23.** **Inference Results using sigmoid activation, 1 hidden layer with 128 neuron**


**FIGURE 24.** **Incorrect Prediction using sigmoid activation, 1 hidden layer with 128 neuron**

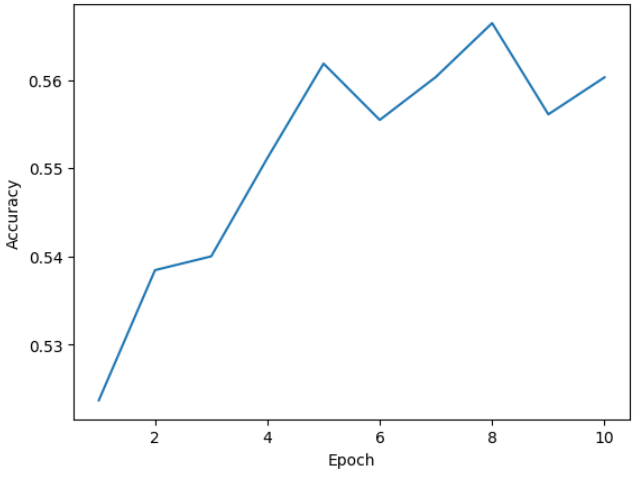


**FIGURE 25.** **Accuracy vs Epoch curve using sigmoid activation, 1 hidden layer with 128 neuron with 0.5 dropout rate**

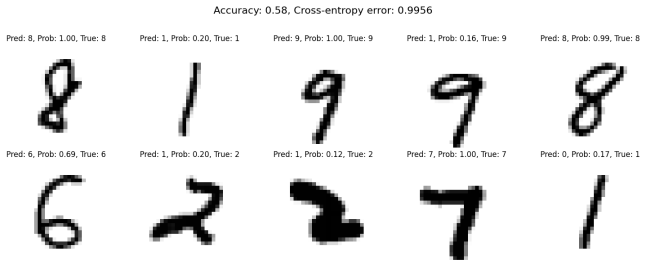


**FIGURE 26.** Inference Results using sigmoid activation, 1 hidden layer with 128 neuron with 0.5 dropout rate

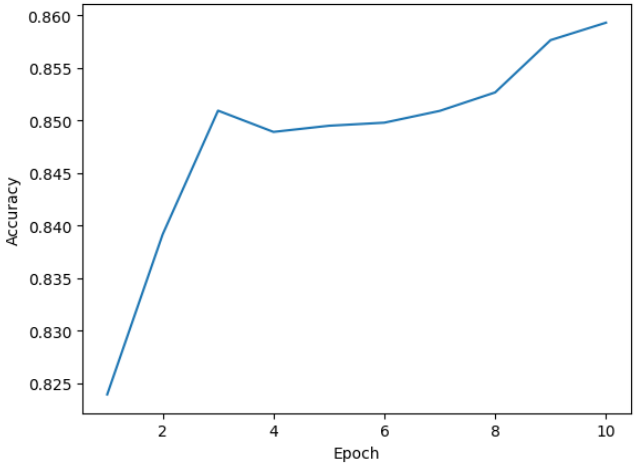


**FIGURE 27.** Accuracy vs Epoch curve using sigmoid activation, 1 hidden layer with 128 neuron with 0.2 dropout rate

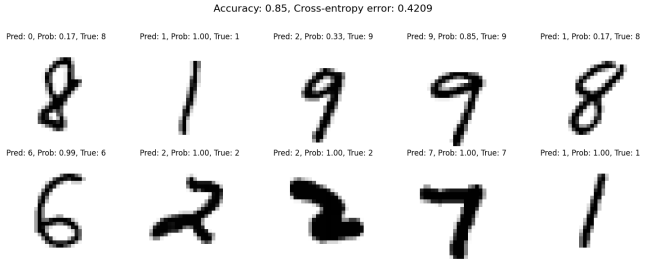


**FIGURE 28.** Inference Results using sigmoid activation, 1 hidden layer with 128 neuron with 0.2 dropout rate

## 1) Using Kaiming Initialization for Weights and Bias

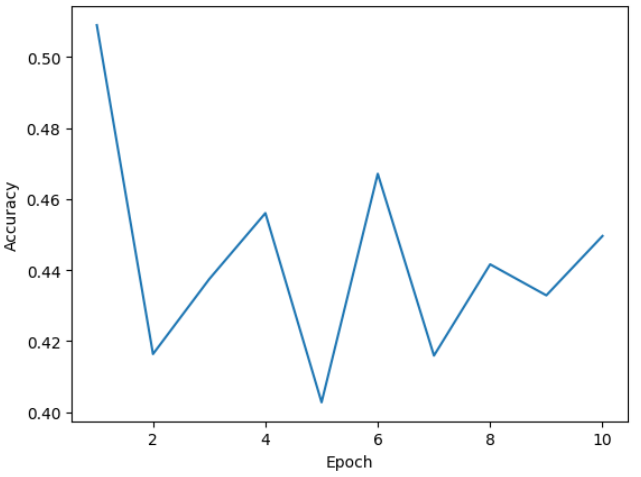


**FIGURE 29.** Accuracy vs Epoch curve using ReLU activation, 1 hidden layer with 10 neuron

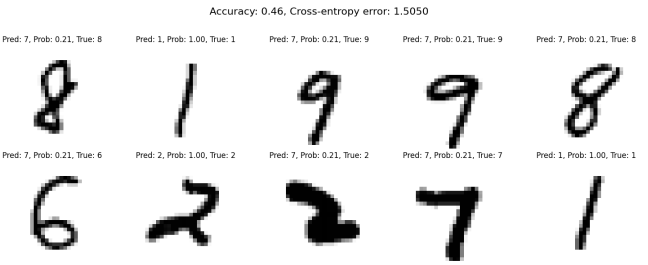


**FIGURE 30.** Inference Results using ReLU activation, 1 hidden layer with 10 neuron

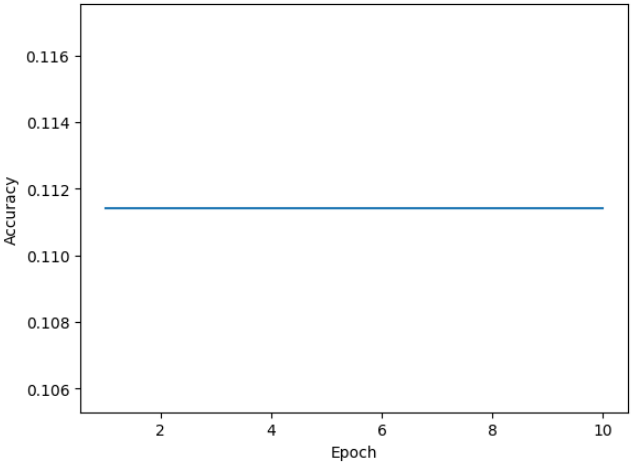


**FIGURE 31.** Accuracy vs Epoch curve using ReLU activation, 2 hidden layer with 30 neuron

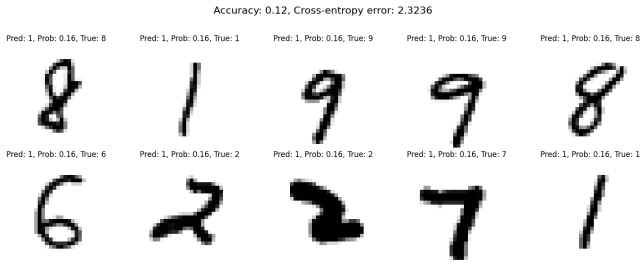


**FIGURE 32.** Inference Results using ReLU activation, 2 hidden layer with 30 neuron

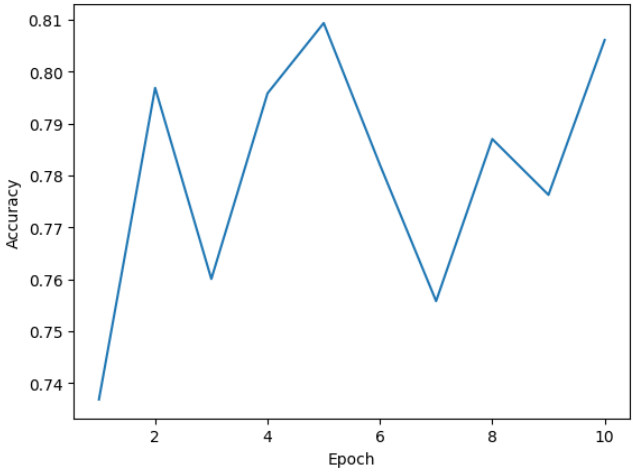


**FIGURE 33.** Accuracy vs Epoch curve using ReLU activation, 1 hidden layer with 128 neuron

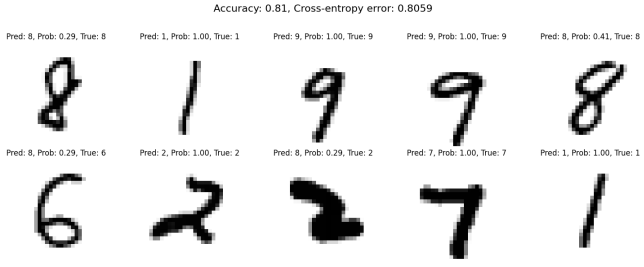


**FIGURE 34.** Inference Results using ReLU activation, 1 hidden layer with 128 neuron

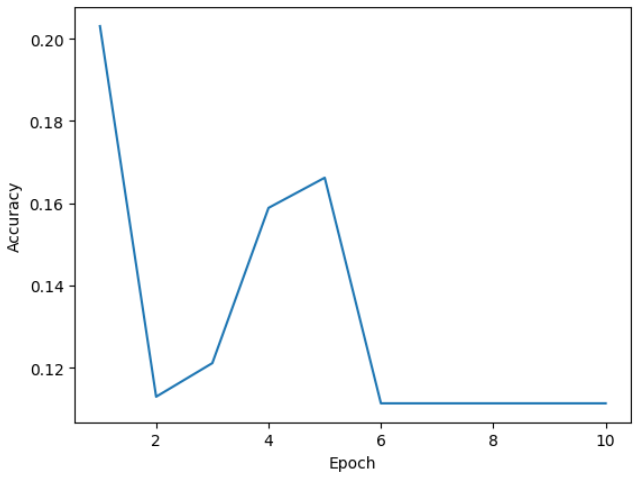


**FIGURE 35.** Accuracy vs Epoch curve using ReLU activation, 2 hidden layer with 128 neuron

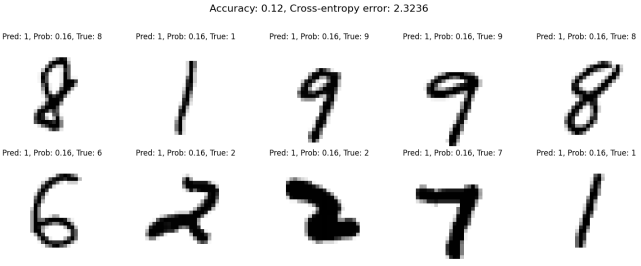


**FIGURE 36.** Inference Results using ReLU activation, 2 hidden layer with 128 neuron

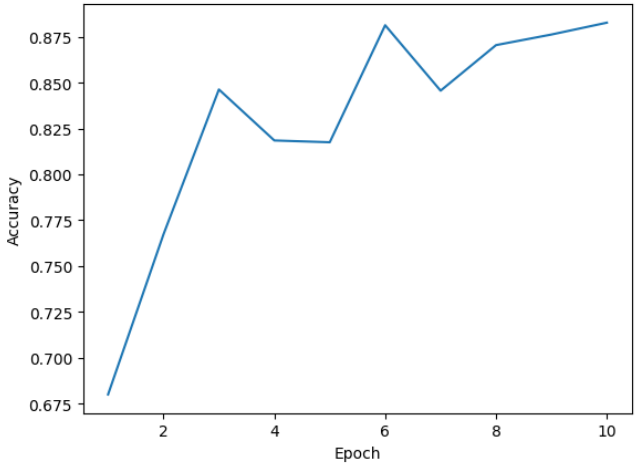


**FIGURE 37.** Accuracy vs Epoch curve using tanh activation, 1 hidden layer with 128 neuron

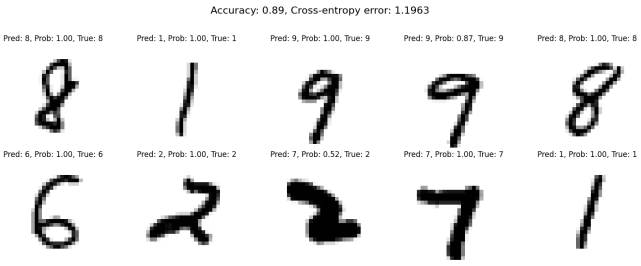


**FIGURE 38.** **Inference Results using tanh activation, 1 hidden layer with 128 neuron**

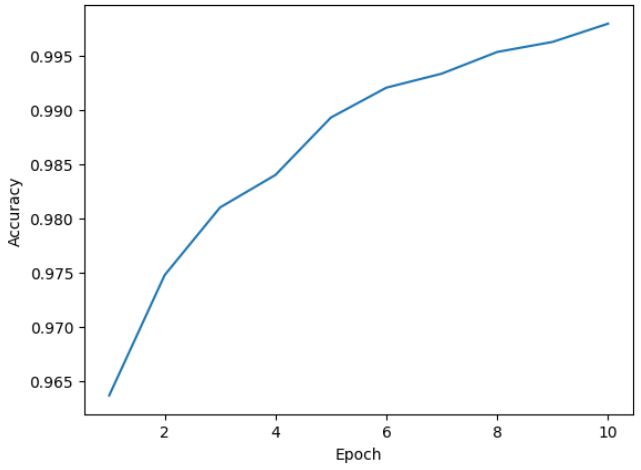


**FIGURE 39.** **Accuracy vs Epoch curve using sigmoid activation, 1 hidden layer with 128 neuron**

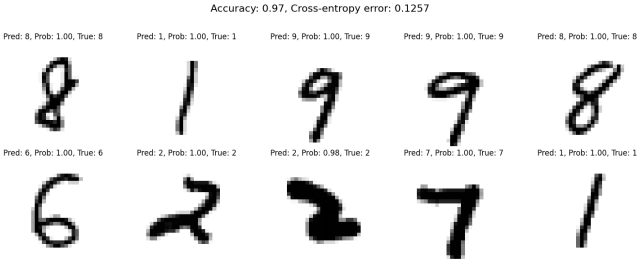


**FIGURE 40.** **Inference Results using sigmoid activation, 1 hidden layer with 128 neuron**

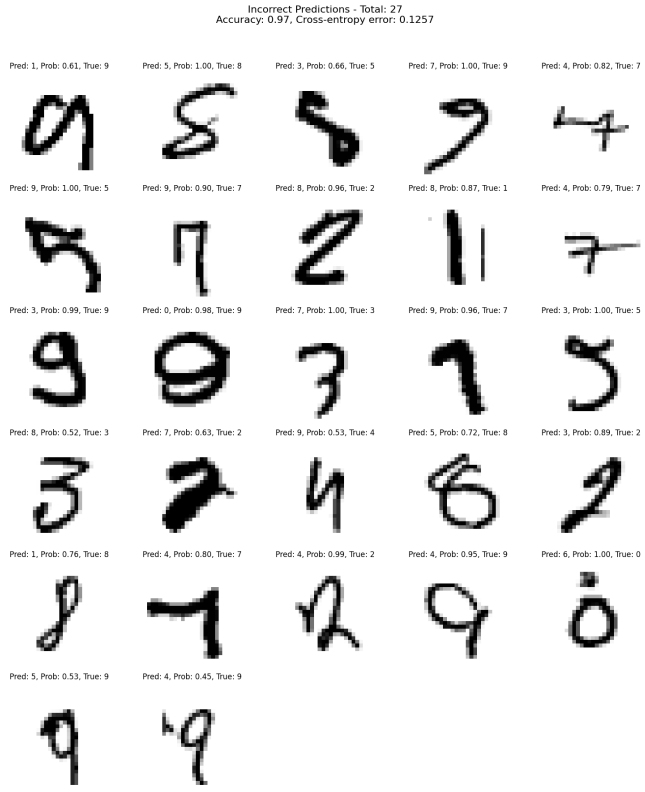


**FIGURE 41.** **Incorrect Prediction using sigmoid activation, 1 hidden layer with 128 neuron**

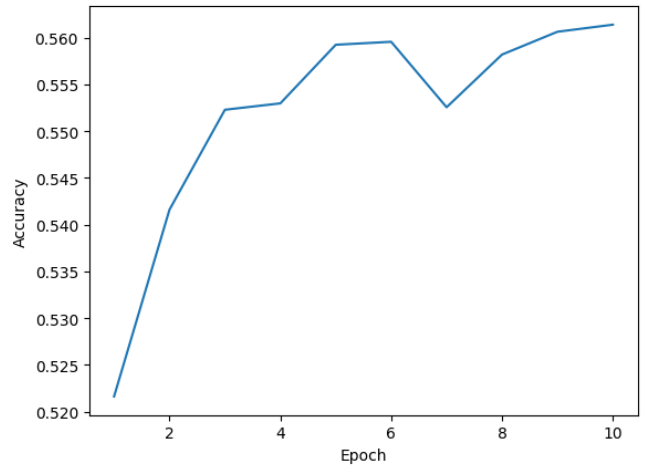


**FIGURE 42.** **Accuracy vs Epoch curve using sigmoid activation, 1 hidden layer with 128 neuron with 0.5 dropout rate**

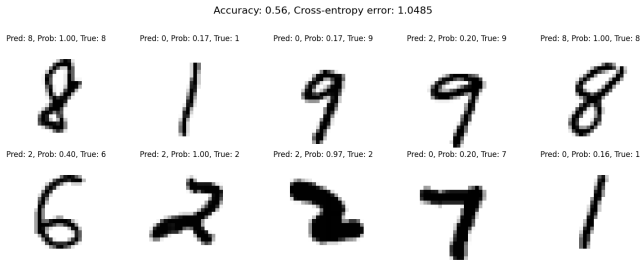


**FIGURE 43. Inference Results using sigmoid activation, 1 hidden layer with 128 neuron with 0.5 dropout rate**

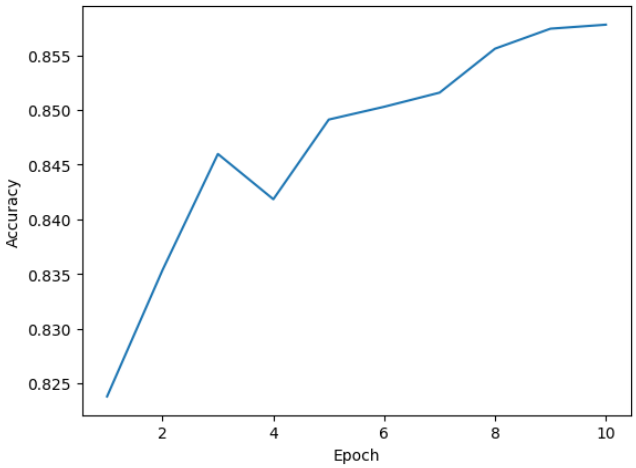


**FIGURE 44. Accuracy vs Epoch curve using sigmoid activation, 1 hidden layer with 128 neuron with 0.2 dropout rate**

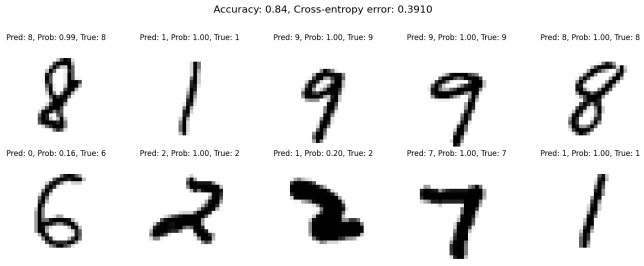


**FIGURE 45. Inference Results using sigmoid activation, 1 hidden layer with 128 neuron with 0.2 dropout rate**

## REFERENCES

[1] A. Katal and N. Singh, "Artificial Neural Network: Models, Applications, and Challenges," in *Innovative Trends in Computational Intelligence*, R. Tomar, M. D. Hina, R. Zitouni, and A. Ramdane-Cherif, Eds. Cham, Switzerland: Springer, 2022, pp. 245-267.

[2] I. N. da Silva, D. H. Spatti, R. A. Flauzino, L. H. B. Liboni, and S. F. dos Reis Alves, *Artificial Neural Networks: A Practical Course*, 1st ed. Cham, Switzerland: Springer International Publishing, 2018.

[3] R. Dastres and M. Soori, "Artificial Neural Network Systems," *Int. J. Imaging Robot.*, vol. 21, pp. 13–25, Mar. 2021.

[4] Y.-C. Wu and J.-W. Feng, "Development and Application of Artificial Neural Network," *Wireless Personal Communications: An International Journal*, vol. 102, no. 2, pp. 1645-1656, Sep. 2018

[5] A. Ali, "Artificial Neural Network (ANN) with Practical Implementation," *The Art of Data Scicne, Medium*, May 20, 2019. [Online]. Available: https://medium.com/machine-learning-researcher/artificial-neural-network-ann-4481fa33d85a. [Accessed: Aug. 8, 2024].

[6] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550-1560, Oct. 1990.

[7] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *Journal of Machine Learning Research - Proceedings Track*, vol. 9, pp. 249-256, Jan. 2010.

[8] G. Jha, "MNIST Handwritten Digit Recognition using multi-layer neural network," *Analytics Vidhya, Medium*, Jun. 13, 2020. [Online]. Available: https://medium.com/analytics-vidhya/mnist-handwritten-digit-recognition-using-neural-network-2b729bacb0 [Accessed: Aug. 8, 2024].

**NIMESH G. PRADHAN** is currently pursuing a Bachelor's degree in Electronics, Communication, and Information Engineering at Thapathali Campus. He is currently in the final year of his degree. His interests lie in the fields of Data Mining, Machine Learning, and Deep Learning.

**RAMAN BHATTARAI** is currently pursuing a Bachelor's degree in Electronics, Communication, and Information Engineering at Thapathali Campus. He is currently in the final year of his degree. His interests lie in the fields of Data Mining, Computer Vision, and Deep Learning.