Building a Robust Handwritten Digit Recognition Neural Network from Scratch using the MNIST Dataset

Adhip Bhattarai[1*], Bishal Rijal[2*]

[1]Department of Electronics and Computer Engineering, Thapathali Campus, IOE, Tribhuvan University (e-mail: adhipbh200@gmail.com)

[2]Department of Electronics and Computer Engineering, Thapathali Campus, IOE, Tribhuvan University (e-mail: bishalrijal5467@gmail.com)

Corresponding author: First A. Author (e-mail: author@ boulder.nist.gov).

*Authors contributed equally

ABSTRACT During our lab work, we built an Artificial Neural Network (ANN) with two hidden layers right from the basics. We trained this network using the MNIST dataset, which is commonly used for recognizing numbers in images. We did a bunch of tests to see how well the network worked with different setups. First, we changed the number of neurons in those hidden layers to see if it made the network better at understanding the data. We also tried training the network for different numbers of rounds, which we call epochs, to figure out the best timing for learning. Another thing we looked at was how to start the network's "thinking" process. We tried three different ways: one where the starting points were random, another where they followed a LeCun method, and the last one using a He method. This basically means we tried three different setups for the initial weights inside the network, which can affect how well it learns. When we began with random weights and used ReLU as the activation function for both hidden layers, the network got about 79.47% of the answers right. When we used the He method, the accuracy dropped a bit to 27.83%, and with the LeCun method, it was even lower at 24.01%. For the LeCun and He methods, we used the ReLU function as the activation for the first hidden layer and the tanh function as the activation function for the second hidden layer.

INDEX TERMS ANN, hidden, layers, output, weight

I. INTRODUCTION

The phrase "artificial neural network" refers to a biologically inspired branch of artificial intelligence that is fashioned after the brain. An artificial neural network is a computer network that is based on biological neural networks that build the structure of the human brain. Similar to how neurons in the human brain are interconnected, neurons in artificial neural networks are linked to each other at various levels of the networks. These neurons are referred to as nodes.

Neural networks are a class of machine learning algorithms inspired by the way biological nervous systems, such as the human brain, process information. They have gained immense popularity in recent years due to their ability to learn complex patterns and solve a wide range of tasks, from image and speech recognition to language translation and game playing. At their core, neural networks consist of interconnected nodes, known as neurons or units, organized into layers. Each neuron processes information and passes it along to the next layer, ultimately producing an output. The connections between neurons, often referred to as "weights," determine the strength of the signal being transmitted.

Neural networks have a rich history that spans decades. Initially inspired by the human brain, early conceptualizations of artificial neurons emerged in the 1940s. The 1960s and 1970s saw initial attempts at neural network application in pattern recognition, but limited computational resources and training complexities impeded significant progress. In the 1980s and 1990s, the rediscovery of the backpropagation algorithm reignited interest in neural networks, yielding advancements in handwriting recognition and other fields. The 2000s, however, saw a shift toward other machine-learning techniques. The pivotal resurgence occurred in the 2010s with the advent of deep learning. Improved computing power, novel activation functions, and abundant data facilitated the training of deep neural networks. Convolutional and recurrent neural networks achieved breakthroughs in image analysis and sequence tasks. Presently, neural networks are integral to AI, excelling in diverse areas like image recognition, language processing, and medical diagnosis, while ongoing research strives to refine architectures and techniques, underscoring their profound and enduring impact on modern technology and research endeavors.

II. METHODOLOGY

A. ARTIFICIAL NEURAL NETWORKS

The artificial neural network (ANN), which uses a mix of AI and brain-inspired architecture, has transformed modern computing.

An ANN has three major types of layers: the input layer, one or more hidden layers, and the output layer. The initial data is received by the input layer, while the final outputs, such as classifications or predictions, are produced by the output layer. As the name implies, hidden layers are intermediary layers between the input and output layers that play an important role in collecting complicated patterns in data. Each neuron in one layer is linked to every neuron in the next layer. Weights are assigned to these connections, which affect the intensity of the signal carried from one neuron to the next. These weights are applied to the input data, and the resultant values are sent to the neurons of the following layer [1].

In the hidden and output layers, neurons generally apply an activation function to the weighted sum of inputs. This function adds non-linearity to the network, allowing it to mimic more complicated data interactions. The sigmoid, hyperbolic tangent (tanh), and rectified linear unit (ReLU) are examples of common activation functions. It is proven that if the activation function is not used in the ANN, then no matter how many hidden layers, we add the ANN fails to capture the non-linearity between the input and output variables.

An Artificial Neural Network is shown in Figure 1. In an ANN, there is a single input and output layer but there can be multiple hidden layers.

Neural networks learn and depict patterns in data using weights and activation functions to collect information. During training, these weights are adjusted to best suit the input-output correlations in the data. Neural networks capture information in the following ways:

1. Representation of Input Data: Neural networks begin by expressing input data as numerical values. These inputs might be raw picture pixel values, textual characteristics, sensor readings, or any other type of data that the network is built to analyze.

2. Weighted Sum of Inputs: In a neural network, each neuron receives these input values, multiplies them by the corresponding weights, and computes the weighted sum of inputs. The weights function as changeable parameters that govern the relevance of each input to the neuron's output.

3. Activation Function: The weighted total of the inputs is then processed by an activation function. This stage brings non-linearity to the network, allowing it to record complicated data correlations. To alter the neuron's output, several activation functions such as sigmoid, tanh, or ReLU are utilized.

4. Pattern Recognition through Training: Initially, the network's weights are assigned random values. A set of labeled training examples is supplied to the network during training. The network is fed the inputs, and the outputs are compared to the real labels.

5. Loss/Cost Calculation: A loss or cost function calculates the difference between projected and actual output labels. The purpose of training is to reduce this mistake. The initial random weights of the network often result in a significant loss, but the weights are continuously changed to improve predictions.

6. Backpropagation and Weight Adjustment: Backpropagation is the process of propagating an error backward across a network to determine how much each weight contributed to the error. Gradient descent or other optimization methods are then used to decrease the error by adjusting the weights. Weights that contribute to bigger mistakes are modified more aggressively, whereas weights that contribute to minor errors are altered less aggressively.

7. Iterative Refinement: This is the process of sending data through the network, assessing mistakes, and modifying weights over and over until the network's performance improves. The network changes the weights as it learns to catch relevant patterns in the data.

8. Feature Hierarchies and Abstractions: Each hidden layer in deep neural networks collects increasingly more abstract and higher-level characteristics. Lower layers record simple picture elements like as edges and corners, while higher layers learn more complicated patterns, eventually leading to the network recognizing entire objects or scenes.

9. Generalization: The purpose of training is to construct a neural network that is good at generalizing to new inputs. This means that, in addition to the training data, the network should be able to generate correct predictions on fresh, previously unknown samples.

B. MODEL ARCHITECTURE

The System Block Diagram is represented in the Appendix and Explained in the Working Principle section.

C. INSTRUMENTATION

- Matplotlib: It is a popular data visualization library for Python. It is used to visualize graphs between training and validation loss as well as training and validation accuracy.

- Numpy: Numpy provides tools for handling large,multi-dimensional arrays, along with a wide range of mathematical functions. It was used for calculating the activation of different functions as well as their derivatives.

- Pandas: It is a library for data manipulation and analysis. It was used for data loading of the MNIST dataset. Furthermore, data selection was used to filter the target column.

III. WORKING PRINCIPLE

A. DATASET PREPARATION

Consider a dataset

$$X = (x_{ij})_{m\times n} \qquad 1$$

having m number of instances with n number of features. For x_{ij}, i represents the data and j represents the feature. The MNIST dataset is a collection of pictures of handwritten digits. In this case, "i" would represent the total number of pictures we have, and "j" would represent the different aspects we want to know about each picture. In total, each picture has 785 different aspects, where 784 aspects describe the intensity of individual pixel values in a 28x28 grid (that's 784 pixels in total), and the last aspect tells us which number the handwritten digit represents, ranging from 0 to 9.

B. DATASET PREPROCESSING

To make sure our analysis of the MNIST dataset [2] is accurate and useful, we need to prepare the data properly. This involves a series of steps that help us get the data ready for processing by machine learning algorithms.

- Label Encoding: Unlike typical datasets with categorical variables, the MNIST dataset contains pixel values that represent the intensity of each pixel in grayscale images. These values are already numerical and don't require label encoding. However, since our output is multiclass, we performed one hot encoding to the output variable.

- Normalization: Normalization is crucial for ensuring that all the pixel values have a consistent scale. In the case of MNIST, pixel values are in the range of 0 to 255 (representing pixel intensity). So, we divided all the input columns by 255 to normalize the pixel values in the range between 0 and 1.

- Handling Missing Values: Fortunately, the MNIST dataset doesn't have missing values, as each image is well-defined with its pixel values.

C. Neural Network Classification

Neural networks perform better on classification tasks than traditional machine learning algorithms. However, unlike traditional machine learning algorithms which can give better results on a few amounts of data neural networks require a lot of data for training.

At first, we defined our model architecture. We used different architectures for different weight initialization methods. They can be seen in Figure 2, Figure 3 and Figure 4. All the architectures consist of one input layer, two hidden layers, and one output layer. The number of neurons in the input and layer is 784 and 10 respectively for all architectures. Only the number of neurons in the hidden layer varies in different architectures.

Before we start training our neural networks, we first need to initialize their weights. We used three methods for weight initialization:

- Random Initialization: Here we randomly initialized the weights and biases randomly. The random values were between -0.5 to 0.5.

- He Initialization: This method is named after its creator, Kaiming He. ‘He’ initialization is commonly used for networks that use the ReLU (Rectified Linear Unit) activation function. ReLU is popular because it doesn't suffer from the vanishing gradient problem, but it can cause issues with weights exploding during training. ‘He’ initialization addresses this issue. The idea behind He initialization is to set the initial weights of each neuron to be drawn from a Gaussian distribution with mean 0 and variance of double of reciprocal of the number of input neurons. This variance scaling is based on the properties of the ReLU activation function, and it helps prevent the gradients from becoming too large during the forward and backward passes, thus aiding in stable and efficient training.

- LeCun Initialization: It was proposed by Yann LeCun. This initialization method is designed to work well with activation functions like tanh and sigmoid, which can suffer from the vanishing gradient problem when weights are initialized improperly. The LeCun initialization sets the initial weights of each neuron to be drawn from a Gaussian distribution with mean 0 and variance of reciprocal of number of input neurons. This helps in distributing the initial weights in a way that the activations don't get too small as they pass through the layers, thus addressing the vanishing gradient problem.

Then we performed forward propagation. During forward propagation, we performed the following steps:

1. Initialization: We began by loading the learned parameters of the neural network, which included biases and weights for each layer.

2. Layer 1 Calculation: We calculated the weighted sum of the input data by applying the weights of the first layer and adding the bias term. After that, we applied an activation function (ReLU) to the weighted sum to obtain the activation values of the first layer. If A be the output of ReLU,

$$\mathrm{A} = \max(0,\mathrm{z}) \qquad 2$$

The Rectified Linear Unit (ReLU) function retains a positive neuron output unchanged while setting negative outputs to zero. The derivative of ReLU is 1 if the neuron output is positive and 0 if the neuron output is negative. If the neuron output is 0, then the derivative doesnot exist. Here, ∂A represents the derivative of ReLU function.

$$\partial A = \begin{cases} 0 \; if \; z < 0 \\ 1 \; if \; z > 0 \end{cases} \qquad 3$$

3. Layer 2 Calculation: Moving on, we computed the weighted sum of the activation values from the previous layer using the weights of the second layer and adding the corresponding bias term. We then applied an activation function (Tanh) to the weighted sum to get the activation values for the second layer. For random initialization of weights, we used ReLU as an activation function instead of tanh. For the tanh function, the equation is:

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \qquad 4$$

The derivative of tanh(x) is given as:

$$tanh'(x) = 1 - tanh^{2}(x) \qquad 5$$

4. Output Layer Calculation: Continuing the process, we computed the weighted sum of the activation values from the previous layer using the weights of the output layer and adding the bias specific to the output layer. Subsequently, we applied an activation function (Sigmoid) to the weighted sum to obtain the final output of our neural network.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \qquad 6$$

The derivative of the sigmoid function is given as:

$$\sigma'(x) = \sigma(x)[1 - \sigma(x)] \qquad 7$$

After forward propagation, we obtained a list of probabilities for each class for each training instance. After this, we calculated the loss for one epoch summing up the loss from all the training instances. Here, we used mean squared loss as our loss function. The losses can be used for backpropagation for updating the parameters. Here is the algorithm for the backpropagation:

1. Retrieved Intermediate Values: The algorithm began by retrieving the activation values (a1, a2, a3) and weighted sum values (z1, z2, z3) obtained from the previous forward propagation step.

2. Extracted Parameters: The learned parameters of the neural network, including weights (w1, w2, w3) and biases (b1, b2, b3) for each layer, were extracted from the parameters dictionary.

3. One-Hot Encoding: The true labels (y_true) were converted into one-hot encoded format (y_one_hot) to match the network's output structure.

4. Calculated Output Layer Gradients and Updates: Gradients were calculated for the output layer using the difference between the computed activation values (a3) and the one-hot encoded labels (y_one_hot). These gradients were then multiplied by the derivative of the sigmoid activation function (sigmoid_derivative) applied to the weighted sum (z3). The weight updates (dw3) and bias updates (db3) for the output layer were computed by taking the dot product of the gradient (dz3) with the activation values from the previous layer (a2), normalized by the number of training examples.

5. Calculated Second Hidden Layer Gradients and Updates: Gradients for the second hidden layer (dz2) were calculated by multiplying the transpose of the weights of the output layer (w3) with the gradients of the output layer (dz3). These gradients were then multiplied by the derivative of the hyperbolic tangent (tanh_derivative) applied to the weighted sum (z2). Weight updates (dw2) and bias updates (db2) for the second hidden layer were computed in a similar manner as for the output layer, using the gradient (dz2) and the activation values from the first hidden layer (a1).

6. Calculated First Hidden Layer Gradients and Updates: Gradients for the first hidden layer (dz1) were calculated by multiplying the transpose of the weights of the second hidden layer (w2) with the gradients of the second hidden layer (dz2). These gradients were then multiplied by the derivative

of the ReLU activation function (relu derivative) and applied to the weighted sum (z1). Weight updates (dw1) and bias updates (db1) for the first hidden layer were computed in a similar manner as for the other layers, using the gradient (dz1) and the input data (x_train).

8. Updated Parameters: The learned parameters (weights and biases) for all layers were updated by subtracting the respective weight or bias updates scaled by the learning rate (lr).

9. Updated Intermediate Values: The updated weights and biases were stored within the intermediate values dictionary to keep track of the changes for subsequent iterations.

D. Forward Propagation Equations

Here are the forward propagation equations:

1. At first, we multiplied the transpose of the input matrix with the transpose of initialized weights between the input layer and the first hidden layer. The number of initialized weights is dependent on the number of neurons in the first hidden layer.

$$Z_1 = {W_1}^T . X + b_1 \qquad 8$$

Here, Z_1 is the matrix consisting output of all neurons of 1st hidden layer before passing through the activation
W_1 is the matrix containing weights between the input and the first hidden layer
X is a transposed training matrix.
b_1 is the matrix consisting of the bias of all neurons in the first hidden layer.

2. Then we passed Z_1 to ReLU function. Here, A_1 is the matrix consisting of all the output of neurons of 1st hidden layer after passing through activation.

$$A_1 = \max(0, Z_1) \qquad 9$$

3. After this we passed A1 to 2nd hidden layer.

$$Z_2 = {W_2}^T \ . A_1 + b_2 \qquad 10$$

Here, Z_2 is the matrix consisting output of all neurons of 2nd hidden layer before passing through the activation
W_2 is the matrix containing weight between the first and the second hidden layer
b_2 is the matrix consisting of the bias of all neurons in the second hidden layer.

4. Then we passed Z_2 to the tanh and ReLU function. For random weight initialization we passed Z_2 through ReLU and for other initialization techniques we used tanh function.

$$A_2 = ReLU(Z_2) \qquad 11$$

Or,

$$A_2 = \tanh(Z_2) \qquad 12$$

5. After this, we calculated Z_3 which is the output of the output neuron before passing through the activation.

$$Z_3 = {W_3}^T . A_2 + b_3 \qquad 13$$

Here, W_3 is the matrix containing weight between the second hidden layer and the output layer.
b_3 is the matrix consisting of the bias of all neurons in the output layer.

6. Finally, we calculated A3 by passing Z3 through sigmoid activation.

$$A_3 = sigmoid(Z_3) \qquad 14$$

E. BACKWARD PROPAGATION EQUATIONS

After the forward propagation, we performed backward propagation to adjust the weights of the neural networks. Here, we used mean squared loss as the loss function for calculating the loss.

The mean squared error (MSE) or mean squared deviation (MSD) of an estimator (a process for estimating an unobserved variable) in statistics measures the average of the squares of the errors, i.e., the average squared difference between the estimated and actual values. MSE is a risk function that represents the expected value of squared error loss. Because of randomness or because the estimator does not account for information that may give a more accurate estimate, MSE is almost always strictly positive (rather than zero). MSE may refer to the empirical risk (the average loss on an observed data set) as an estimate of the real MSE (the true risk: the average loss on the actual population distribution) in machine learning, especially empirical risk minimization.

$$MSE = \frac{1}{2n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2 \qquad 15$$

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

In forward propagation $\hat{Y}_i = A_3$.

So, for backward propagation, we need derivative of the loss function MSE with respect to $\boldsymbol{A_3}$.

$$\frac{\partial L}{\partial A_3} = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right) \qquad 16$$

Here, L represents the Loss function i.e., MSE.

In matrix form if M is the matrix containing the difference between all true and predicted values then,

$$\frac{\partial L}{\partial A_3} = \frac{1}{n} M \cdot M^T \quad 17$$

For Z_3 the equation is,

$$\frac{\partial L}{\partial Z_3} = \frac{\partial L}{\partial A_3} \cdot A_3'(Z_3) \quad 18$$

For updating weights between 2nd hidden layer and output layer the slope is given as,

$$\frac{\partial L}{\partial W_3} = \frac{\partial L}{\partial Z_3} \cdot \frac{\partial Z_3}{\partial W_3} \quad 19$$

It can also be written as,

$$\partial W_3 = \frac{1}{n} \partial Z_3 \cdot {A_2}^T \quad 20$$

Here, ∂W_3 represents the matrix containing the gradients of all weights between output layer and 2nd hidden layer.

Also, slope for weights between 1st and 2nd hidden layer

$$\partial Z_2 = W_3 \,.\, \partial Z_3 \,.\, A'_2(Z_2) \quad 21$$

$$\partial W_2 = \frac{1}{n} \partial Z_2 \,.\, {A_1}^T \quad 22$$

For weights between input layer and first hidden layer the slopes are:

$$\partial Z_1 = W_2 . \partial Z_2 \,.\, A'_2(Z_2) \quad 23$$

$$W_1 = \frac{1}{n} \partial Z_1 . X^T \quad 24$$

Here, X represents training matrix.

For biases,

$$\partial b_j = \frac{1}{n} \sum_{i=1}^{3} \sum_{k=1}^{3} \partial z_{ik} \quad 25$$

Here, 'j' represents the layer number. It varies from 1 to 3. 'i' represents the ith row of the ∂Z_j matrix and 'k' represents kth column of ∂Z_j matrix.

∂z_{ik} represents the element at ith row and kth column of the ∂Z_j matrix.

Then the weights and biases are updated using gradient descent:

$$W_{j\_n} = W_{j\_o} - \alpha . \partial W_j \quad 26$$

Here, W_{j_n} represents the updated weight, W_{j_o} represents the old weight.

α represents the learning rate.

$$b_{j\_n} = b_{j\_o} - \alpha . \partial b_j \quad 27$$

Here, b_{j_n} represents the updated bias, b_{j_o} represents the old bias.

IV. RESULTS

We embarked on the task of creating an Artificial Neural Network (ANN) tailored to the MNIST dataset, a widely recognized collection of handwritten digit images. The objective was to develop a model capable of classifying these digits accurately.

The MNIST dataset comprises a multitude of grayscale images, each depicting a single digit from 0 to 9. Our approach began with preprocessing the dataset. We normalized the pixel values, which ranged from 0 to 255, to a standardized scale, typically between 0 and 1, to facilitate convergence during training.

Subsequently, we designed our ANN architecture from the ground up. The architecture consisted of an input layer, two hidden layers, and an output layer. Each layer comprised a set of neurons, and the connections between these neurons were governed by weights. To introduce non-linearity into the model, we incorporated activation functions such as ReLU, tanh for the hidden layers and sigmoid for the output layer.

For the purpose of training the model, we implemented a backpropagation algorithm. This involved forward-passing an input through the network to obtain predictions, comparing these predictions to the actual labels, and then propagating the error backward to adjust the weights iteratively. We employed a loss function, typically mean-squared loss, to quantify the disparity between predictions and actual labels.

Upon employing random weight initialization, our ANN demonstrated remarkable Train Accuracy of 79.47% and Validation Accuracy of 78.71%. Upon implementing He initialization—a technique designed to align with networks employing ReLU activations—the Train Accuracy diminished to 27.83%, accompanied by a Validation Accuracy of 27.61%. For lecun initialization the train Accuracy dwindled further to 24.01%, and the Validation Accuracy echoed this decline at 22.88%. We also strategically plotted graphs that tracked the trajectory of training loss and validation loss across epochs.

V. DISCUSSION AND ANALYSIS

In our exploration of constructing an Artificial Neural Network (ANN) for the MNIST dataset, we engaged in a comprehensive analysis to discern the model's performance and glean insights from our experimental configuration. By systematically fine-tuning various parameters, our objective was to optimize the network's capabilities and uncover the nuances of its operation.

Throughout our experimentation, we subjected the ANN to a rigorous evaluation using the MNIST dataset. We evaluated the model's performance using the provided dataset, which includes handwritten digit images. Employing meticulous tuning and training, we witnessed intriguing results that shed light on the effectiveness of our approach.

Firstly, upon employing random weight initialization, our ANN demonstrated a remarkable Train Accuracy of 79.47% and a Validation Accuracy of 78.71%. These compelling results underscored the network's ability to grasp intricate patterns within the training data and effectively generalize its understanding to previously unseen validation data. By embracing this approach, we permitted the network to dynamically evolve its knowledge through iterative learning cycles, leading to commendable performance.

However, as we ventured into the territory of weight initialization strategies, deviations in outcomes surfaced. Upon implementing He initialization—a technique designed to align with networks employing ReLU activations—the Train Accuracy diminished to 27.83%, accompanied by a Validation Accuracy of 27.61%. This unexpected drop in accuracy signaled that He initialization might not harmonize optimally with our specific architecture or dataset.

A parallel trend emerged with LeCun initialization, originally tailored for networks featuring tanh activations. In this scenario, the Train Accuracy dwindled further to 24.01%, and the Validation Accuracy echoed this decline at 22.88%. The observations here indicated that the choice of weight initialization should be meticulously aligned with the network's architecture and activation functions.

As part of our analysis, we also plotted graphs depicting the training loss and validation loss in relation to the number of training epochs. These visualizations provided a dynamic view of the learning process and revealed how the model's performance evolved over time. Such insights were invaluable for optimizing hyperparameters and addressing potential overfitting or underfitting issues. The graphs can be seen in Figure 5, Figure 6, Figure 7, Figure 8, Figure 9, and Figure 10.

VI. CONCLUSION:

Applying the principles of Artificial Neural Networks (ANNs) to the MNIST dataset, we embarked on a journey of discovery, aiming to achieve accurate digit classification through careful design and iterative optimization. Our approach was grounded in the thoughtful preprocessing of the dataset and the construction of a customized neural network architecture.

In preparation for training, we meticulously preprocessed the MNIST dataset, normalizing pixel values to a consistent scale, and partitioning it into training, validation, and testing sets. This step laid a solid foundation for subsequent analysis.

With the dataset primed, we delved into constructing the ANN architecture. Our model consisted of an input layer, two hidden layers, and an output layer. The network's neurons were interconnected with weighted connections, and activation functions like ReLU, tanh, etc. were integrated to introduce non-linearity. The number of neurons in each layer, the depth of the network, and hyperparameters were all critical considerations that influenced the architecture's performance.

Looking ahead, enhancing the ANN's accuracy could involve diving deeper into advanced architectural variations like convolutional neural networks (CNNs) tailored for image data. Additionally, adopting techniques such as transfer learning and ensemble methods could unlock even more robust and precise classification outcomes.

VII. REFERENCES:

[1] S. Lek and Y. Park, "Artificial Neural Networks," in *Encyclopedia of Ecology*, Oxford, Academic Press, 2008, pp. 237-245.

[2] Y. LeCun, C. Cortes and C. Burges, "MNIST handwritten digit database," *ATT Labs*, vol. 2, 2010.

Adhip Bhattarai is a dedicated individual pursuing a Bachelor's degree in Computer Engineering at Tribhuvan University. With a strong passion for machine learning and data science, he is constantly exploring the latest advancements in these fields. Although he may not have notable accomplishments just yet, Adhip's enthusiasm and drive for learning and applying cutting-edge technologies make him a promising and ambitious individual in the world of computer engineering.

Bishal Rijal is a dedicated individual currently studying Bachelor’s in Computer and Technology at Tribhuvan University. Bishal’s enthusiasm for research and innovation has led him to undertake various projects and engage in practical applications of his knowledge. He continually seeks to deepen in understanding of the subject matter, staying up-to-date with the latest advancements and trends. With his relentless determination, inquisitive mindset, and expertise in machine learning and data science, Bishal Rijal is poised to make significant contributions to the ever-evolving field of technology.

VIII. APPENDICES

A. FIGURES

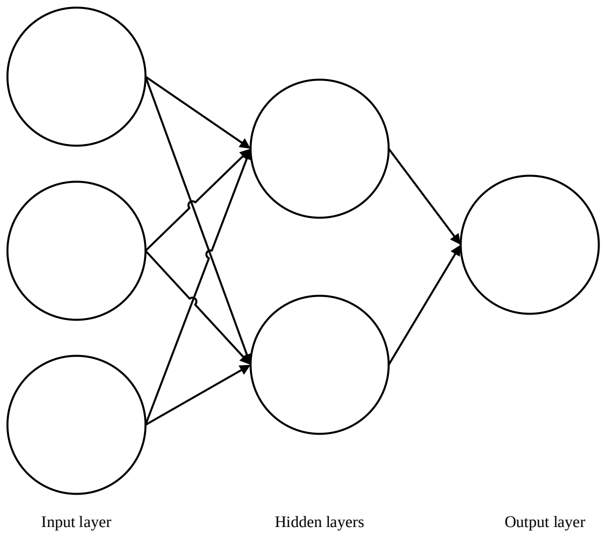
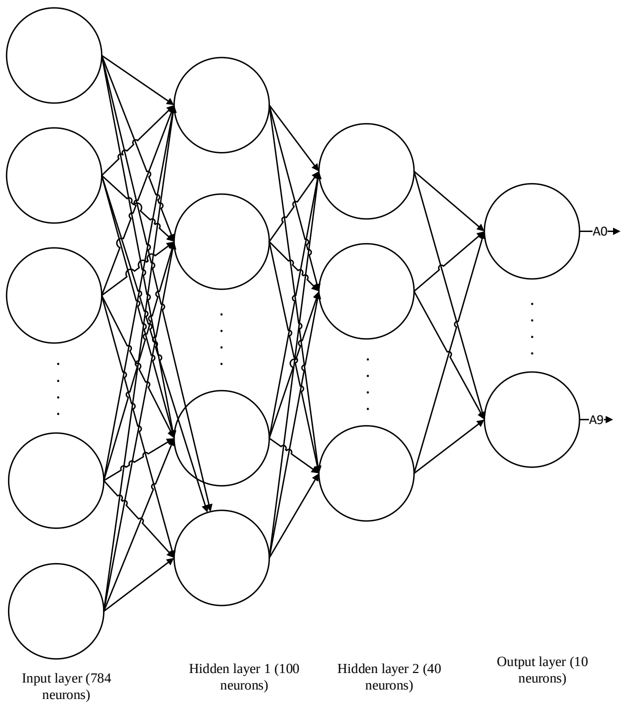
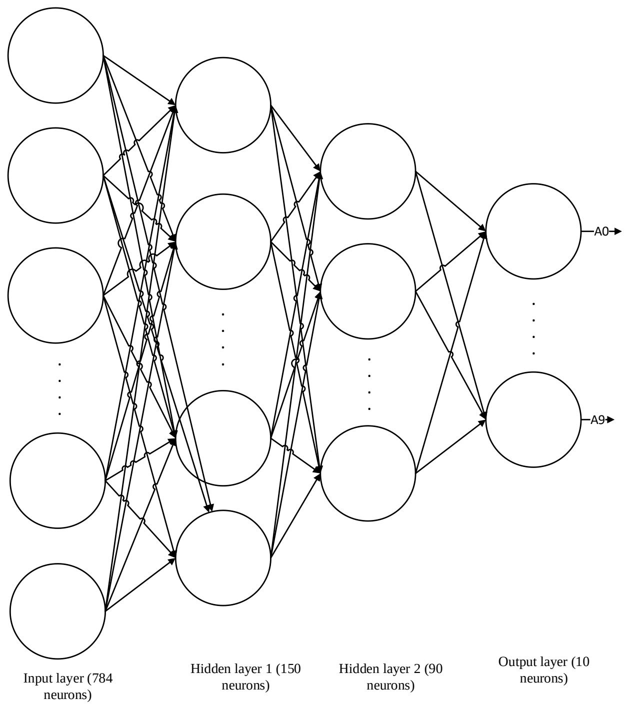


Figure 1. Artificial Neural Network



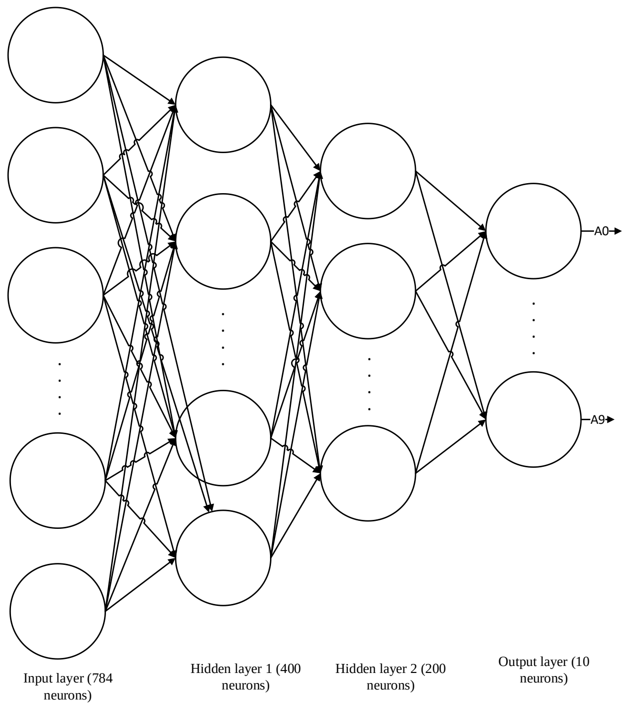
A0
A9
Input layer (784 neurons)
Hidden layer 1 (100 neurons)
Hidden layer 2 (40 neurons)
Output layer (10 neurons)

Figure 2. ANN for random initialization of weights



A0
A9
Input layer (784 neurons)
Hidden layer 1 (150 neurons)
Hidden layer 2 (90 neurons)
Output layer (10 neurons)

Figure 3. ANN for He initialization of weights



A0
A9
Input layer (784 neurons)
Hidden layer 1 (400 neurons)
Hidden layer 2 (200 neurons)
Output layer (10 neurons)

Figure 4. ANN for Lecun initialization of weights

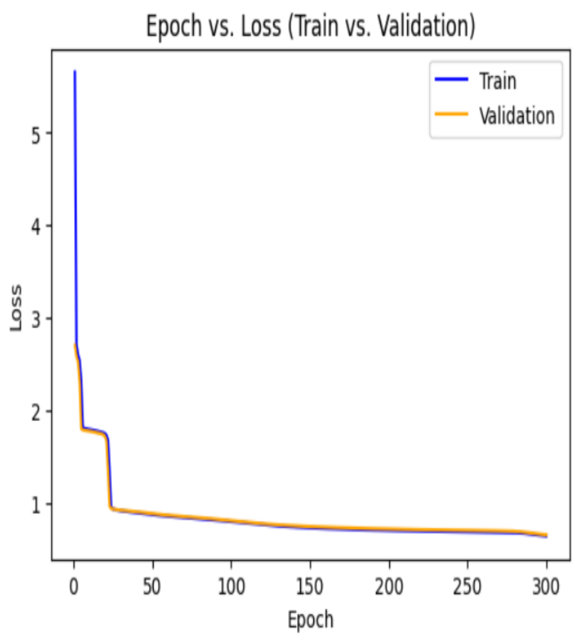


Figure 5. Epoch vs Loss graph for random weight initialization

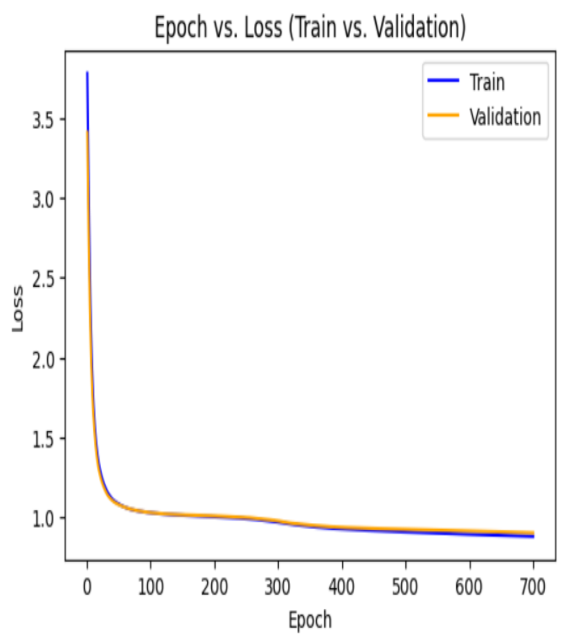


Figure 7. Epoch vs Loss Graph for He weight initialization

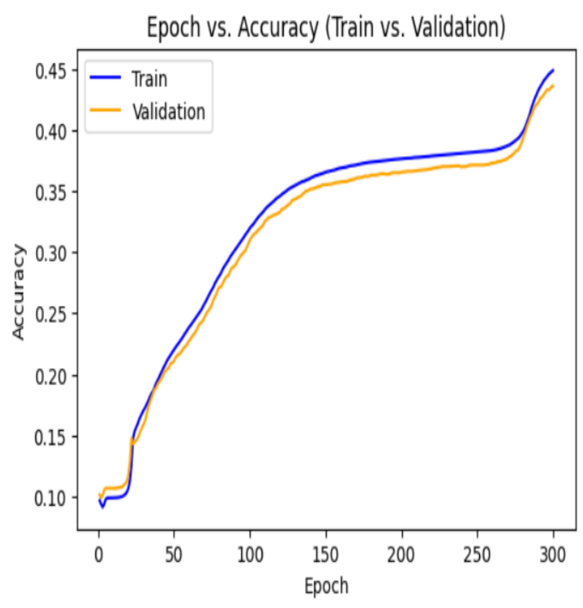


Figure 6. Epoch vs Accuracy graph for random weight initialization

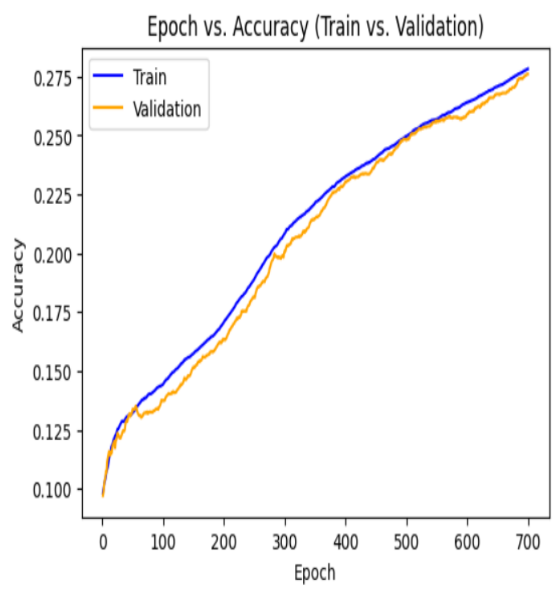


Figure 8. Epoch vs Accuracy Graph for He weight initialization

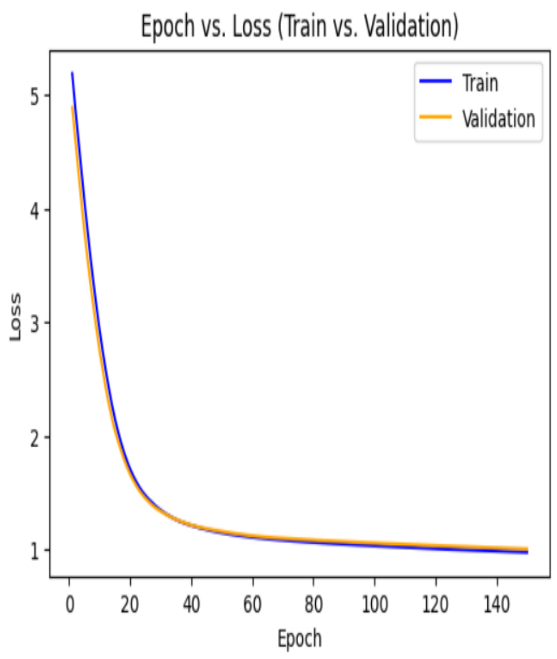


Figure 9. Epoch vs Loss Graph for Lecun Weight initialization

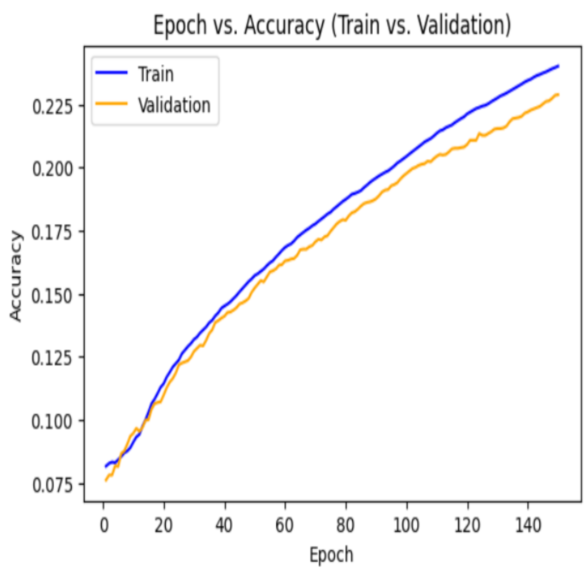


Figure 10. Epoch vs Accuracy Graph for Lecun Weight initilization

B. CODE:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import random

file_path = 'digit_data.csv'
df = pd.read_csv(file_path)
df

df.head()

df_array = np.array(df)

df_array

np.random.shuffle(df_array)

df_array

test = df_array[:1000,:]
train = df_array[1000:,:]

test.shape

train.shape

train = train.T

y = train[0,:]
y

X = train[1:,:]
X

X = X/255
X

X.shape

index = random.randint(0, 40999)
index

img = X[:, index].reshape(28,28)
plt.imshow(img)
```

```
def
initialize_weights_and_biases(input_size,
hidden_size1, hidden_size2, output_size,
init_type='random', random_state=None):
    if init_type == 'random':
        if random_state is not None:
            np.random.seed(random_state)
        w1 = np.random.uniform(-0.5, 0.5,
size=(input_size, hidden_size1))
        w2 = np.random.uniform(-0.5, 0.5,
size=(hidden_size1, hidden_size2))
        w3 = np.random.uniform(-0.5, 0.5,
size=(hidden_size2, output_size))
        b1 = np.random.uniform(-0.5, 0.5,
size=(hidden_size1, 1))
        b2 = np.random.uniform(-0.5, 0.5,
size=(hidden_size2, 1))
        b3 = np.random.uniform(-0.5, 0.5,
size=(output_size, 1))


    elif init_type == 'he':
        w1 = np.random.randn(input_size,
hidden_size1)
        w2 =
np.random.randn(hidden_size1,
hidden_size2)
        w3 =
np.random.randn(hidden_size2,
output_size)
        b1 = np.zeros((hidden_size1, 1))
        b2 = np.zeros((hidden_size2, 1))
        b3 = np.zeros((output_size, 1))

    elif init_type == 'lecun':
        w1 = np.random.randn(input_size,
hidden_size1)
        w2 =
np.random.randn(hidden_size1,
hidden_size2)
        w3 =
np.random.randn(hidden_size2,
output_size)
        b1 = np.zeros((hidden_size1, 1))
        b2 = np.zeros((hidden_size2, 1))
        b3 = np.zeros((output_size, 1))
```

```
    else:
        raise ValueError("Invalid
init_type. Supported values are 'random',
'xe', 'he', and 'lecun'.")

    initial_weights = {
        'w1': w1,
        'w2': w2,
        'w3': w3,
        'b1': b1,
        'b2': b2,
        'b3': b3
    }

    return initial_weights

def relu_function(z):
    return np.maximum(z,
np.zeros(z.shape))

def relu_derivative(x):
  return np.where(x > 0, 1, 0)

def softmax(x):
  e_x = np.exp(x - np.max(x))  #
Subtracting max(x) for numerical
stability
  return e_x / e_x.sum(axis=0)

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

def sigmoid_derivative(x):
    s = sigmoid(x)
    return s * (1 - s)

def tanh(x):
    return np.tanh(x)

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

def forward_prop(parameters, x_train):
    b1 = parameters['b1']
    w1 = parameters['w1']
    b2 = parameters['b2']
    w2 = parameters['w2']
    b3 = parameters['b3']
    w3 = parameters['w3']

    # Layer 1
    z1 = np.matmul(w1.T, x_train) + b1
    a1 = relu_function(z1)

    # Layer 2
    z2 = np.matmul(w2.T, a1) + b2
    # a2 = relu_function(z2)    #For
random initilization of weights
    a2 = tanh(z2) # for lecun, he

    # Output Layer
    z3 = np.matmul(w3.T, a2) + b3
    a3 = sigmoid(z3)

    intermediate_values = {
        'z1': z1,
        'a1': a1,
        'z2': z2,
        'a2': a2,
        'z3': z3,
        'a3': a3
    }

    return intermediate_values

def
categorical_cross_entropy(predicted_label
s, true_labels):
    # Ensure predicted_labels and
true_labels have the same shape
    if predicted_labels.shape !=
true_labels.shape:
        raise ValueError("Shapes of
predicted_labels and true_labels must
match.")

    m = predicted_labels.shape[0]
    # Clip predicted probabilities to
avoid log(0)
    epsilon = 1e-15
    predicted_labels =
np.clip(predicted_labels, epsilon, 1 -
epsilon)

    # Calculate cross-entropy loss
    loss = (- (true_labels) *
np.log(predicted_labels +
epsilon)).sum(axis=1)
    total_loss = loss.sum(axis = 0)
    average_loss = total_loss/m
    return average_loss

def mean_squared_loss(predictions,
targets):
    # Ensure predictions and targets have
the same shape
    if predictions.shape !=
targets.shape:
        raise ValueError("Shapes of
predictions and targets must match.")

    m = predictions.shape[0]

    # Calculate mean squared loss
    loss = ((predictions - targets) **
2).sum(axis=1)
    total_loss = loss.sum(axis=0)
    average_loss = total_loss / m

    return average_loss

def calculate_train_accuracy(parameters,
x_train, y_train):
    k = parameters
    s = x_train.shape[1]
    predicted_labels =
np.argmax(parameters, axis=0)
    predicted_labels =
np.argmax(parameters, axis=0).reshape(-1,
1)
    acc = np.sum(predicted_labels ==
y_train)
    accuracy = acc/s
    return accuracy
```

```
def backpropagation(x_train, y_true,
parameters, intermediate_values,
lr=0.01):
    a1 = intermediate_values['a1']
    a2 = intermediate_values['a2']
    a3 = intermediate_values['a3']
    z1 = intermediate_values['z1']
    z2 = intermediate_values['z2']
    z3 = intermediate_values['z3']

    w1 = parameters['w1']
    w2 = parameters['w2']
    w3 = parameters['w3']
    b1 = parameters['b1']
    b2 = parameters['b2']
    b3 = parameters['b3']

    y_one = np.eye(10)[y_true.reshape(-
1)]
    y_one_hot = y_one.astype(int)

    dz3 = (a3 - y_one_hot.T) *
sigmoid_derivative(z3)
    dw3 = np.matmul(dz3, a2.T) /
x_train.shape[1]
    db3 = np.sum(dz3, axis=1,
keepdims=True) / x_train.shape[1]

    # dz2 = np.matmul(w3, dz3) *
relu_derivative(z2) #For random
initilization of weights
    dz2 = np.matmul(w3, dz3) *
tanh_derivative(z2)
    dw2 = np.matmul(dz2, a1.T) /
x_train.shape[1]
    db2 = np.sum(dz2, axis=1,
keepdims=True) / x_train.shape[1]

    dz1 = np.matmul(w2, dz2) *
relu_derivative(z1)
    dw1 = np.matmul(dz1, x_train) /
x_train.shape[1]
    db1 = np.sum(dz1, axis=1,
keepdims=True) / x_train.shape[1]

    dw1 = dw1.T
    dw2 = dw2.T
    dw3 = dw3.T

    # Update weights and biases
    w1 = w1 - lr * dw1
    w2 = w2 - lr * dw2
    w3 = w3 - lr * dw3
    b1 = b1 - lr * db1
    b2 = b2 - lr * db2
    b3 = b3 - lr * db3

    intermediate_values = {
        "w1": w1,
        "w2": w2,
        "w3": w3,
        "b1": b1,
        "b2": b2,
        "b3": b3
    }

    return intermediate_values

def train_neural_network(x_train,
y_train, input_size, hidden_size1,
hidden_size2, output_size,
learning_rate=0.01, num_epochs=1000,
validation_data=None,
print_every=100,init_type = "random"):
    parameters =
initialize_weights_and_biases(input_size,
hidden_size1, hidden_size2, output_size,
init_type=init_type)

    train_data = []
    val_data = []

    for epoch in range(num_epochs):
        if validation_data is not None:
            intermediate_values_train =
forward_prop(parameters, x_train)
            y_one =
np.eye(10)[y_train.reshape(-1)]
            y_one_hot = y_one.astype(int)
            p =
intermediate_values_train['a3'].T
```

```
            loss_train =
mean_squared_loss(y_one_hot, p)
            train_acc =
calculate_train_accuracy(intermediate_val
ues_train['a3'], x_train, y_train)

            parameters =
backpropagation(x_train.T, y_train,
parameters, intermediate_values_train,
learning_rate)

            x_val, y_val =
validation_data
            y_for_val =
np.eye(10)[y_val.reshape(-1)]
            y_hot_val =
y_for_val.astype(int)

            intermediate_values_val =
forward_prop(parameters, x_val.T)
            p =
intermediate_values_val['a3'].T
            loss_val =
mean_squared_loss(y_hot_val, p)
            val_acc =
calculate_train_accuracy(intermediate_val
ues_val['a3'], x_val.T, y_val)

            train_data.append({
                'epoch': epoch + 1,
                'loss': loss_train,
                'accuracy': train_acc,
            })

            val_data.append({
                'epoch': epoch + 1,
                'loss': loss_val,
                'accuracy': val_acc,
            })

            if ((epoch+1) % print_every)
== 0 or (epoch == 0):
                print(f"Epoch
{epoch+1},\tTrain Loss:
{loss_train.item():.4f},\tTrain Accuracy:
{train_acc.item():.2%},\tValidation Loss:
{loss_val.item():.4f}\tValidation
Accuracy: {val_acc.item():.2%}")

        else:
            intermediate_values_train =
forward_prop(parameters, x_train.T)
            y_one =
np.eye(10)[y_train.reshape(-1)]
            y_one_hot = y_one.astype(int)
            p =
intermediate_values_train['a3'].T
            loss_train =
categorical_cross_entropy(y_one_hot, p)

            parameters =
backpropagation(x_train.T, y_train,
parameters, intermediate_values_train,
learning_rate)

            if ((epoch+1) % print_every)
== 0 or (epoch == 0):
                print(f"Epoch
{epoch+1},\tTrain Loss:
{loss_train.item():.4f}\tTrain Accuracy:
{train_acc.item():.2%}")

            train_data.append({
                'epoch': epoch + 1,
                'loss': loss_train,
                'accuracy': train_acc,
            })

    if validation_data is not None:
        return parameters, train_data,
val_data
    else:
        return parameters, train_data

X = X.T
X.shape

validation_size = int(0.1 * len(X))
validation_data = X[:validation_size]
y = y.T
y = y.reshape(-1,1)
```

```
validation_label =
y[:validation_size].reshape(-1,1)
training_data = X[validation_size:]
training_label =
y[validation_size:].reshape(-1,1)

print(validation_data.shape)
print(validation_label.shape)
print(training_data.shape)
print(training_label.shape)
print(y.shape)
print(y)

num_classes = 10
y_one = np.eye(num_classes)[y.reshape(-
1)]
y_one_hot = y_one.astype(int)
print("One-Hot Encoded Labels:")
print(y_one_hot)
print("Shape:", y_one_hot.shape)

training_data = training_data.T

parameters, train_data, val_data =
train_neural_network(
    x_train=training_data,
    y_train=training_label,
    input_size=784,
    hidden_size1=100,  # First hidden
layer with 100 neurons
    hidden_size2=40,   # Second hidden
layer with 40 neurons
    output_size=10,    # Output size
    learning_rate=0.01,
    num_epochs=400,
    validation_data=[validation_data,
validation_label],
    print_every=5
)

def plot_train_val_metrics(train_data,
val_data, metric_name, ylabel):
    epochs_train = [entry['epoch'] for
entry in train_data]
    metric_values_train =
[entry[metric_name] for entry in
train_data]

    epochs_val = [entry['epoch'] for
entry in val_data]
    metric_values_val =
[entry[metric_name] for entry in
val_data]

    plt.figure(figsize=(6, 4))
    plt.plot(epochs_train,
metric_values_train, label='Train',
color='blue')
    plt.plot(epochs_val,
metric_values_val, label='Validation',
color='orange')
    plt.xlabel('Epoch')
    plt.ylabel(ylabel)
    plt.title(f'Epoch vs. {ylabel} (Train
vs. Validation)')
    plt.legend()
    plt.show()

plot_train_val_metrics(train_data,
val_data, 'loss', 'Loss')

plot_train_val_metrics(train_data,
val_data, 'accuracy', 'Accuracy')



plot_train_val_metrics(train_data,
val_data, 'loss', 'Loss')

plot_train_val_metrics(train_data,
val_data, 'accuracy', 'Accuracy')

parameters, train_data, val_data =
train_neural_network(
    x_train=training_data,
    y_train=training_label,
    input_size=784,
    hidden_size1=150,  # First hidden
layer with 100 neurons
```

```
    hidden_size2=90,    # Second hidden
layer with 40 neurons
    output_size=10,     # Output size
    learning_rate=0.01,
    num_epochs=700,
    validation_data=[validation_data,
validation_label],
    print_every=5,
    init_type = 'he'
)

plot_train_val_metrics(train_data,
val_data, 'loss', 'Loss')

plot_train_val_metrics(train_data,
val_data, 'accuracy', 'Accuracy')

parameters, train_data, val_data =
train_neural_network(
    x_train=training_data,
    y_train=training_label,
    input_size=784,
    hidden_size1=400,  # First hidden
layer with 100 neurons
    hidden_size2=200,   # Second hidden
layer with 40 neurons
    output_size=10,     # Output size
    learning_rate=0.01,
    num_epochs=150,
    validation_data=[validation_data,
validation_label],
    print_every=5,
    init_type = 'lecun'
)

plot_train_val_metrics(train_data,
val_data, 'loss', 'Loss')

plot_train_val_metrics(train_data,
val_data, 'accuracy', 'Accuracy')
```