Date of publication 08/20/2023
Digital Object Identifier 10.1109/ACCESS.2023.0322000

Crafting Neural Network: From Theory To Practical Implementation, Forged from Scratch

Arahanta Pokhrel[1] Amit Raj Pant[1]

Insitute of Engineering, Thapathali Campus, Final Year

Corresponding authors:
Arahanta Pokhrel(e-mail:[1]`pokhrelarahanta5@gmail.com`),
Amit Raj Pant (e-mail:[2]`amitrajpant7@gmail.com`).

ABSTRACT This lab report presents the implementation and analysis of a neural network for classification tasks. The main focus is on understanding the concepts of forward and backward propagation, parameter updates, and the effect of different network configurations. The network is developed using Python and major libraries such as NumPy and Matplotlib. Through experimentation, the report discusses the impact of hyperparameters, weight initialization methods, and activation functions on the network's performance. The results highlight insights gained from analyzing the training process and evaluating model accuracy. This work contributes to a deeper understanding of neural networks and their practical applications.

INDEX TERMS Activation Functions, Backpropagation, Hyperparameters, Neural Network,Weight Initialization.

I. INTRODUCTION

Neural networks are like super smart tools in the world of computers and learning. They can figure out tricky patterns and stuff from data. This report is all about making and studying a neural network, like peeking inside to see how it works and how it helps in real life. These networks are like the base of modern computer learning or machine learning, and they're really good at understanding complicated stuff. They're used for all sorts of things like recognizing pictures(image processing) and understanding human language(Natural Language Processing).

In this lab project, we want to dig deep into the main ideas of neural networks. We're especially interested in how they use two important steps, one going forward and the other going backward. This helps them process information and make it better over time. Our goal is to make these complex processes easier to understand. We'll do this by doing some tests and looking at how changing things like settings, functions, and starting points affects how well the network works. we will investigate how different configurations, activation functions,and initialization techniques impact the network'performance.

In today's fast-changing tech world, it's super important to really understand how to use neural networks. As businesses and schools use more data to make choices, knowing about neural networks helps us a lot. When we learn how these networks work, it's like getting a powerful set of tools to solve tough problems in real life. Exploring how neural networks function is like going on an adventure to discover all the cool things they can do. This helps us learn important stuff from data and be part of the coolest new things happening in machine learning.

In the next pages, we're going to carefully explore neural networks step by step. We'll break down the theory behind them, explain the math that helps them learn, and plan how we'll test them in experiments. By doing this careful study, we want to really understand how neural networks work. Plus, we hope to learn useful things that will help us create better models and make smart choices in different areas.

As we dive in to this exploration, the aim is to uncover the mechanisms that facilitate the transformation of raw data into intelligent predictions. Combining theoretical insights with empirical findings contributes to the broader discourse surrounding neural networks and their transformable potential.

II. METHODOLOGY

A. THEORY

An artificial neural network**[1]** works like a smart student, where we give it information with some questions (input data with independent variables) and answers (output data with dependent variable). For example, think of the independent variables as X_1, X_2, and X_3, and the answer as Y.

At first, the neural network guesses randomly**[2]**. Then, it checks how wrong it is by comparing its guesses to the real answers. We measure this difference, which we call error. To understand this better, we use a special math function called a cost function. Our main goal is to make this error as small as possible, like trying to fix mistakes.

The whole process happens in two main steps: Feed Forward and Back Propagation.

1) Feed Forward step

In the Feed Forward step:

1) We multiply the input data with some weights.
2) We use an activation function (like the sigmoid function) to squeeze the result into a helpful range (usually between 0 and 1).

Here's how Feed Forward works:

- Calculate the Weighted Sum:
  In the Feed Forward phase, we start by calculating the weighted sum of the inputs and the corresponding weights for each neuron in the network. This is done using the dot product of the input values (X_i) and the weights (W_{ij}) for each neuron.
  Mathematically, for a neuron j in the hidden layer:

$$z_j = \sum_i (X_i \cdot W_{ij}) + b_j \tag{1}$$

  where:
  - X_i is the i-th input feature
  - W_{ij} is the weight connecting the i-th input to the j-th neuron in the hidden layer
  - b_j is the bias term for the j-th neuron

- Apply Activation Function: The calculated weighted sum is then passed through an activation function (σ) to introduce non-linearity. One common activation function is the sigmoid function:

$$a_j = \sigma(z_j) = \frac{1}{1 + \exp(-z_j)} \tag{2}$$

  where a_j is the output of the activation function for neuron j.

2) Back Propagation

In the beginning, the neural network just guesses randomly.

1) It makes random guesses, compares them to the real answers, and finds out how wrong it is.
2) It changes the weights and bias to make its guesses better match the real answers. This is like training the network.

Here's how Back Propagation works:

- Calculate the Error:
  In the Back Propagation phase, we start by calculating the error between the predicted output (a_j) and the actual target output (Y).

$$\text{error} = Y - a_j \tag{3}$$

- Update Weights and Biases:
  We then update the weights and biases using the gradient descent algorithm. The goal is to minimize the error by adjusting the weights and biases to improve the network's performance.
  The weight update for a neuron j is given by:

$$\Delta W_{ij} = \alpha \cdot \text{error} \cdot \sigma'(z_j) \cdot X_i \tag{4}$$

  where:
  - α is the learning rate
  - $\sigma'(z_j)$ is the derivative of the activation function at the weighted sum z_j
  - X_i is the i-th input feature

  The bias update for neuron j is similar:

$$\Delta b_j = \alpha \cdot \text{error} \cdot \sigma'(z_j) \tag{5}$$

We then adjust the weights and biases using the calculated updates.

Our goal is to find the weights that make the cost as tiny as possible, so our guesses become super accurate.

So, we've covered the Feed Forward and Back Propagation steps. It's time to put all this learning into action.

3) Multi-Layer Perceptron and Its Fundamentals

A perceptron, much like atoms forming the basic structure of matter, is the fundamental unit of a neural network. It takes multiple inputs and produces a single output.

Let's consider a perceptron structure with three inputs and one output. The input-output relationships are established through the following methods:

1) Directly combining inputs and computing output based on a threshold value:

$$\text{Output} = \begin{cases} 1 & \text{if } x_1 + x_2 + x_3 > 0 \\ 0 & \text{otherwise} \end{cases}$$

2) Introducing weights to the inputs:

$$\text{Output} = \begin{cases} 1 & \text{if } w_1x_1 + w_2x_2 + w_3x_3 \\ & \quad > \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

3) Incorporating bias and weights:

$$\text{Output} = \begin{cases} 1 & \text{if } w_1x_1 + w_2x_2 + w_3x_3 \\ & \quad +1 \cdot b > \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

This linear approach led to the evolution of the perceptron into the concept of an artificial neuron. Neurons utilize non-linear transformations (activation functions) to process inputs and biases.

4) Activation Functions in Neural Networks

An activation function is a crucial component of a neural network that takes the sum of weighted input ($w_1x_1 + w_2x_2 + w_3x_3 + w_0$) as an argument and produces the output of the neuron. In the equation above, we have represented the bias term b as w_0, and x_0 is introduced to represent the constant input 1.

The activation function serves a critical role in introducing non-linearity into the network. It enables the network to learn complex relationships and make predictions beyond simple linear transformations. Various activation functions are used, each with its characteristics. Some common activation functions include:

- **Sigmoid Function:** The sigmoid function maps input values to a range between 0 and 1, which is useful for predicting probabilities. It smoothens extreme values, making it suitable for the output layer of binary classification tasks.

$$f(z) = \frac{1}{1+e^{-z}} \tag{6}$$

Where:

$$\begin{aligned} f(z) &: \text{Output of the sigmoid function.} \\ e &: \text{The base of the natural logarithm.} \\ z &: \text{Weighted sum of inputs and bias.} \end{aligned}$$

- **Tanh Function:** The hyperbolic tangent function maps input values to a range between -1 and 1. Like the sigmoid, it's useful for introducing non-linearity, often used in hidden layers of neural networks.

$$f(z) = \tanh(z) \tag{7}$$

Where:

$$\begin{aligned} f(z) &: \text{Output of the tanh function.} \\ z &: \text{Weighted sum of inputs and bias.} \end{aligned}$$

- **Rectified Linear Unit (ReLU):** The ReLU function outputs the input if it's positive, and zero otherwise. It introduces sparsity and speeds up training by avoiding the vanishing gradient problem.

$$f(z) = \max(0, z) \tag{8}$$

Where:

$$\begin{aligned} f(z) &: \text{Output of the ReLU function.} \\ z &: \text{Weighted sum of inputs and bias.} \end{aligned}$$

- **Softmax Function:** The softmax function is often used in the output layer of multi-class classification tasks. It converts raw scores into a probability distribution over multiple classes, ensuring that the sum of probabilities is 1.

$$f(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}} \tag{9}$$

Where:

$$\begin{aligned} f(z_i) &: \text{Probability of class } i \text{ in the output.} \\ e &: \text{The base of the natural logarithm.} \\ z_i &: \text{Raw score or logit of class } i. \\ N &: \text{Total number of classes.} \end{aligned}$$

Activation functions allow neural networks to capture intricate patterns and relationships within the data, making them versatile tools for various tasks.

B. DATASET

The handwritten digit dataset, often referred to as the MNIST dataset, is a widely-utilized benchmark in the field of machine learning and computer vision. This dataset comprises a collection of grayscale images, each representing a single handwritten digit ranging from 0 to 9. It is commonly employed for tasks such as digit recognition and classification.

Key characteristics of the MNIST deta set include:

1) **Number of Samples**: The data set consists of pixel value images, which are further divided for training and for testing.
2) **Image Size**: Each image is presented in grayscale and has dimensions of 28x28 pixels. The images are relatively small and depict centered, normalized digits.
3) **Labeling**: Every image is associated with a corresponding label indicating the digit it represents. The labels are integer values ranging from 0 to 9.

C. WORKING METHODOLOGY

The methodology used for training a neural network and evaluating its performance with various weight initialization methods are listed below.

1) Data Preparation

First, we loaded the data into a format suitable for analysis. Next, we shuffled the data to ensure fairness during training. We then scaled the pixel values to a common range between 0 and 1, making them easier for the network to work with. Lastly, we transformed the class labels into one-hot encoded vectors

2) Neural Network Architecture

The neural architecture consists of two layers: a hidden layer with 100 units and an output layer with 10 units (corresponding to the 10 digit classes). The activation functions used in the network are Rectified Linear Unit (ReLU) for the hidden layer and softmax for the output layer.

1) Input and Output:

$$X : \text{Input matrix of size } m \times 784$$
$$y : \text{Output matrix of size } m \times 10$$

2) Initialization:

$$\begin{aligned} wh &: \text{Weight matrix for hidden layer} \\ &\quad \text{of size } 100 \times 784 \\ bh &: \text{Bias matrix for hidden layer} \\ &\quad \text{of size } 100 \times 1 \\ wout &: \text{Weight matrix for output layer} \\ &\quad \text{of size } 10 \times 100 \\ bout &: \text{Bias matrix for output layer} \\ &\quad \text{of size } 100 \times 1 \end{aligned}$$

3) Weight Initialization Methods

Four different weight initialization methods were employed to initialize the neural network's weights and biases:

- **Lecun Initialization:** Weights were initialized using a uniform distribution with bounds determined by the number of input and output units.
- **Random Initialization:** Weights were initialized using a uniform distribution within the range [-0.5, 0.5].
- **Xavier (XE) Initialization:** Weights were initialized using a uniform distribution with bounds calculated based on the number of input and output units.
- **He Initialization:** Weights were initialized using a uniform distribution with bounds determined by the number of input units.

4) Training Procedure

The training procedure was standardized across all weight initialization methods:

- **Initialization of Parameters:** For each weight initialization method, the neural network parameters (weights and biases) were initialized as specified.
- **Training Iterations:** The neural network was trained for 500 epochs using gradient descent as the optimization algorithm.
- **Validation Set:** In some cases, a validation set was utilized during training to monitor the model's performance and prevent overfitting.

5) Performance Evaluation

Performance evaluation was conducted using the following metrics:

- **Accuracy:** Accuracy was calculated as the ratio of correctly predicted instances to the total number of instances.

- **Categorical Cross-Entropy Loss:** Categorical cross-entropy loss was employed as the loss function for training.

6) Visualization

Visualization was used to depict the training progress and compare the performance of different weight initialization methods. Two types of plots were generated:

- **Accuracy vs. Epoch:** This plot illustrates the evolution of training and validation accuracy over the training epochs.
- **Loss vs. Epoch:** This plot demonstrates how the loss changes during the training process.

.

D. SYSTEM BLOCK DIAGRAM

An overview of system block diagram is As illustrated in Figure 1. The diagram visually represents the interconnected components and processes of our neural network implementation, showcasing the flow of data from input to output. It serves as a blueprint for understanding the neural network's architecture and its training pipeline.

E. INSTRUMENTATION

The implementation of our neural network model is facilitated by a combination of essential tools and libraries:

- **Python Programming Language**: Python's versatility provides a solid foundation for our project, enabling efficient implementation of machine learning algorithms.

- **NumPy**: This library streamlines numerical computations and array operations, crucial for handling large datasets and performing essential mathematical functions within the neural network.

- **Matplotlib**: We utilize Matplotlib for data visualization, creating informative plots, graphs, and charts that aid in analyzing training progress, loss curves, and accuracy trends.

- **Jupyter Notebooks**: Jupyter Notebooks offer an interactive coding environment, merging code execution, documentation, and visualization, allowing for seamless experimentation and annotation.

III. RESULTS

A. TRAINING AND VALIDATION ACCURACY

TABLE 1: Model Initialization Results[1]

Model Initialization	Train accuracy (%)	Validation (%)
He	95.1775	93.95
LeCun	95.237	94.05
Xavier (XE)	95.355	93.95
Random	94.355	92.5
Epoch	500.0000	500.00

TABLE 2: Model Initialization Result[2]

Model Initialization	Loss
He	0.172
LeCun	0.169
Xavier (XE)	0.165
Random	0.195
Epoch	500.000000

The presented tables provides valuable insights into how different ways of setting the initial weights of a neural network impact its performance, especially in terms of the accuracy achieved during the final testing phase. This examination highlights the crucial role that weight initialization plays in determining how well the network learns and performs.

The models being studied were initialized using four distinct methods: He, LeCun, Xavier (XE), and Random. A key takeaway is how the choice of weight initialization has a significant impact on how quickly the model learns and how accurate it becomes. The results consistently show that He, LeCun, and Xavier (XE) initialization methods consistently lead to better final test accuracy compared to the Random initialization method. This underscores the importance of selecting an appropriate weight initialization strategy that aligns with the specific architecture of the network. The way weights are initialized influences where the optimization process starts, and methods that provide a better starting point tend to lead to models that not only learn more effectively but also achieve higher accuracy.

Furthermore, the strong similarity between the validation accuracy and final test accuracy suggests that the models are not becoming overly specialized to the training data. Overfitting, which occurs when a model performs well on the training data but poorly on new, unseen data, doesn't seem to be a significant concern here. The consistency in performance metrics across both training and validation data suggests that the models are generalizing well, making them suitable for accurate predictions on new data.

In addition, the examination of loss values reveals a clear connection between lower loss and higher accuracy. Among the different weight initialization methods, the He initialization consistently results in the lowest loss, followed by LeCun and Xavier (XE), while Random initialization leads to the highest loss. This finding suggests that the choice of weight initialization significantly influences how efficiently the optimization process progresses. Models with lower loss values are not just more accurate but also converge more rapidly during training. This highlights the importance of selecting an appropriate weight initialization method to achieve optimal performance from neural networks.

Among all the initialization methods, Xavier (XE) consistently stands out with the highest final test accuracy and the lowest loss. This shows that Xavier initialization is a reliable and effective way to start training this particular model. It's like having a secret weapon that works well every time!

IV. DSCUSSION

We successfully implement neural with good accuracy form scratch, specifically focusing on the impact of weight initialization methods on model performance. The results from our experiments revealed some interesting insights.

Firstly, we observed that the choice of weight initialization method has a profound effect on how quickly and accurately a neural network learns. While randomly initializing the weights without any specified range resulted in very poor performance, the model was unable to converge to the accurate weights, whereas when properly done, random weight initialization resulted in convergence. In our experiment, we initialized weights within in the range [-0.5, 0,5].
The He, LeCun, and Xavier (XE) initialization methods consistently outperformed the Random initialization method in terms of final test accuracy and loss. This emphasizes the importance of thoughtful weight initialization, as it can significantly impact the convergence and overall performance of the model.
Furthermore, our experiments demonstrated that the models trained using the He, LeCun, and Xavier methods not only achieved high final test accuracy but also exhibited strong generalization capabilities.

We found something interesting that the validation accuracy and final test accuracy were consistently similar across all ways of starting the weights. This similarity suggests that the models were good at handling new, unseen data without getting too caught up in just learning the training data. Also, there was a clear connection between the loss values and accuracy. Models that began with the He method always had the smallest loss values, followed by LeCun and Xavier. On the other hand, when the weights were randomly set at the beginning, the models had the highest loss values. This high loss shows that the training process was slower for these models.

V. CONCLUSION

Our journey into understanding neural networks has provided us with valuable insights into their setup and performance. We've discovered that how we begin with the initial weights in a network has a significant impact on how well it works. It's essential to select the right method for setting these starting weights, taking into account the network's design and the functions it uses.

We've also learned that while theoretical knowledge is essential, hands-on experiments are equally vital to fine-tune that knowledge in practical scenarios. As we continue exploring the world of neural networks, we learned how the network weight is initialized, how forward pass and Backward pass occurs. How the update of weight is done to fine tune with data so that correct accuracy can be obtained.

Looking at the bigger picture of machine learning, this experiment emphasizes that constructing an effective neural network goes beyond just choosing its structure. The initial weights, the way we kickstart the network, are equally crucial. In the dynamic field of machine learning, insights like these help us create more accurate and efficient models as we move forward.

VI. REFERENCES

References

[1] Sumijan, Agus Perdana Windarto, Abulwafa Muhammad. *Title of the Article*. *International Journal of Software Engineering and Its Applications*. January 2016, Vol. 10, No. 10, pp. 189-204.

[2] Grossi, Enzo, and Buscema, Massimo. *Introduction to Artificial Neural Networks*. *European Journal of Gastroenterology Hepatology*. 2008, Vol. 19, pp. 1046-1054.

APPENDIX A FIGURES AND PLOTS

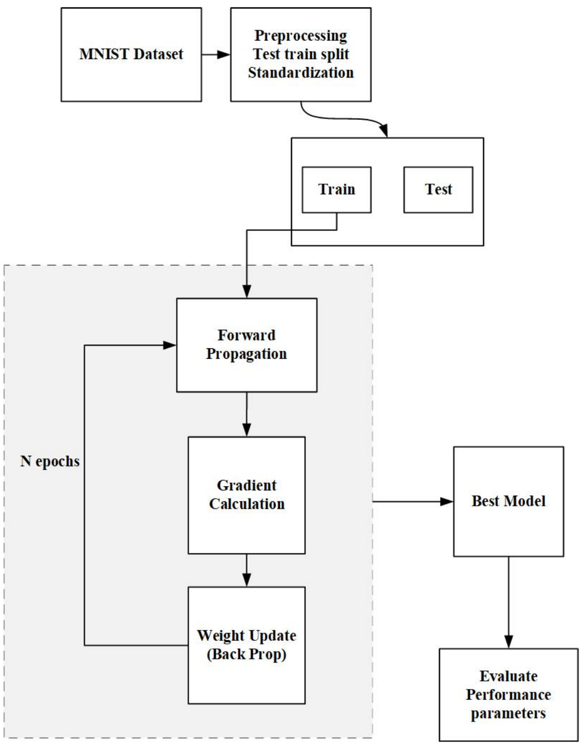


FIGURE 1: System Block Diagram

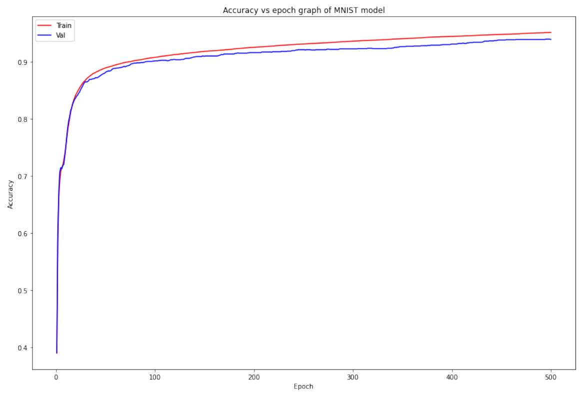


FIGURE 2: Accuracy vs. Epoch (He Initialization)

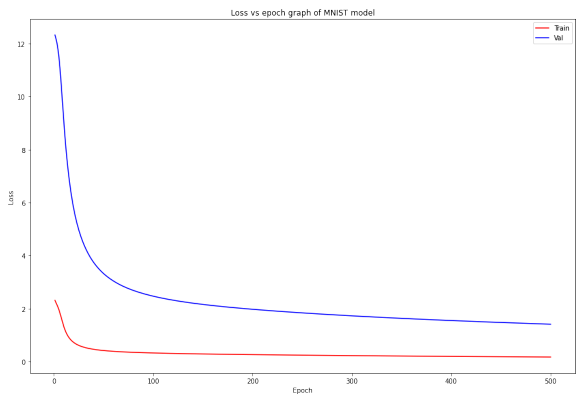


FIGURE 3: Loss vs. Epoch (He Initialization)

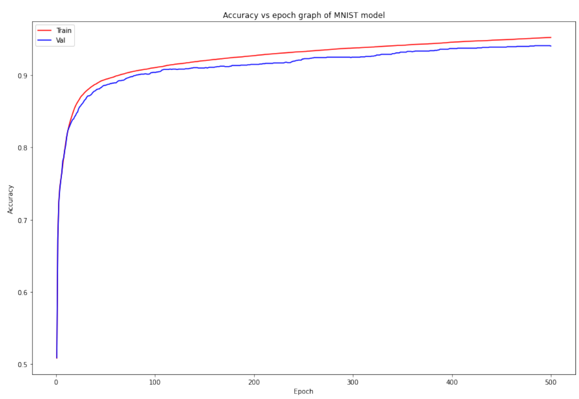


FIGURE 4: Accuracy vs. Epoch (LeCun Initialization)

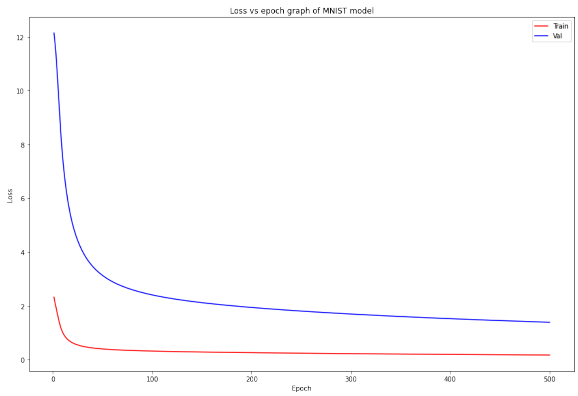


FIGURE 5: Loss vs. Epoch (LeCun Initialization)

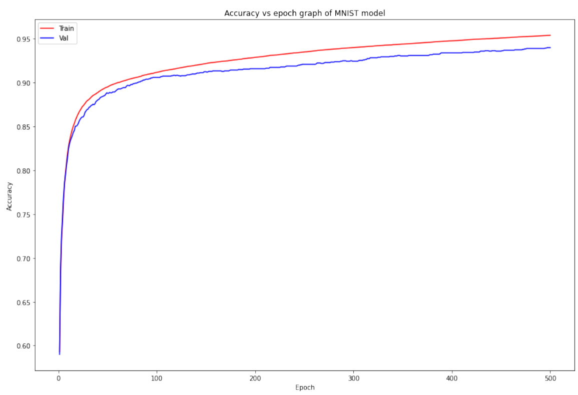


FIGURE 6: Accuracy vs. Epoch (Xavier Initialization)

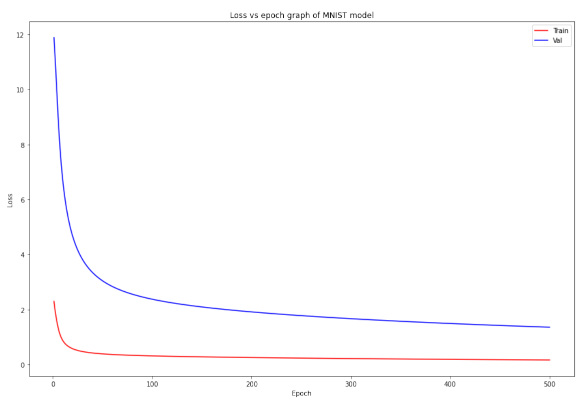


FIGURE 7: Loss vs. Epoch (Xavier Initialization)

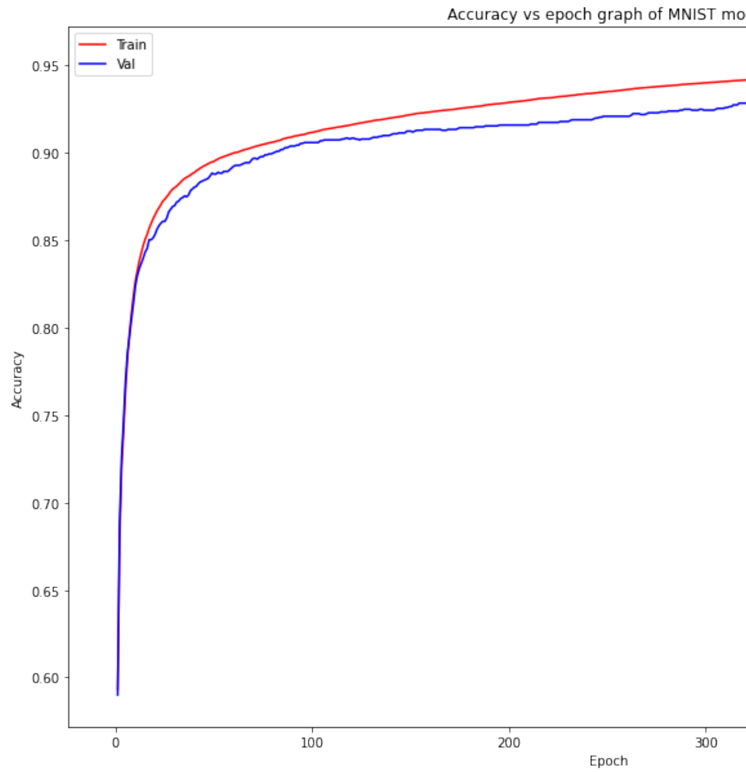


FIGURE 8: Accuracy vs. Epoch (Random Initialization)

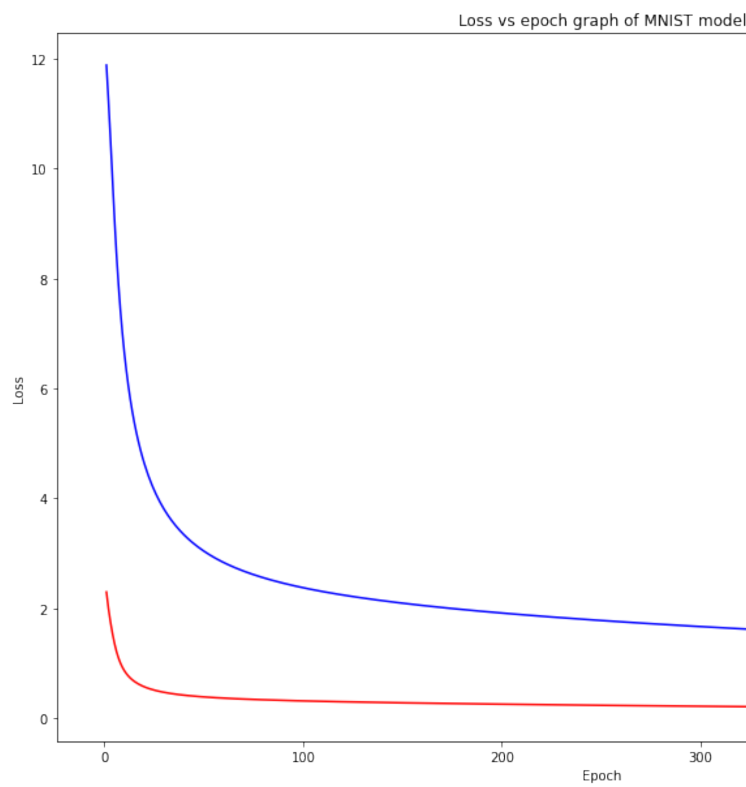


FIGURE 9: Loss vs. Epoch (Random Initialization)

APPENDIX B BACK PROPGATION

Back propagation Algorithm

$$W_{ji} \leftarrow W_{ji} + \Delta w_{ji}$$

where, $\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$

$$E_d(\vec{w}) = \frac{1}{2} \Sigma (t_k - O_k)^2$$

Now, using chain rule

$$\frac{\partial E_d}{\partial w_{ji}} = \frac{\partial E_d}{\partial net_j} \times \frac{\partial net_j}{\partial w_{ji}}$$

where, $net_j = \Sigma w_{ji} x_{ji}$

$$= \frac{\partial E_d}{\partial net_j} \times x_{ji}$$

$$\therefore \partial \Delta W_{ji} = -\eta \frac{\partial E_d}{\partial net_j} \times x_{ji}$$

we consider two case,

Case 1 :- where unit j is an output unit for the network

Case 2 :- where unit j is an internal unit of the network.

Case 1 : Training Rule for output unit weights

: just as w_{ji} can influence the rest of the network only through net_j, net_j can influence the network only through O_j, therefore, we can invoke the chain rule again to write,

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j} \times \frac{\partial o_j}{\partial net_j}$$

Now,

$$\frac{\partial E_d}{\partial O_j} = \frac{\partial}{\partial o_j}\left\{\frac{1}{2} \Sigma (t_k - O_k)^2\right\} =$$

where, t_k is target output, O_k is model output.

$$= \frac{1}{2} \frac{\partial}{\partial o_j} \Sigma (t_j - o_j)^2$$

$$= \frac{1}{2} \times 2 (t_j - o_j) (-1)$$

$$= -(t_j - o_j)$$

we are considering sigmoid function.

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial \sigma(net_j)}{\partial net_j}$$

where $net_j = \Sigma w_{ji} \cdot x_{ji}$

$$\frac{\partial \sigma(x)}{\partial x} = \sigma_x (1 - \sigma(x))$$

$$= \sigma(net_j)(1 - \sigma(net_j))$$

$$= O_j (1 - O_j)$$

$$\frac{\partial E_d}{\partial net_j} = -(t_j - o_j)\, o_j (1 - o_j)$$

Case 1: Training Rule for output unit weights.

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial net_j} x_{ji}$$

$$\Delta w_{ji} = \eta (t_j - o_j)\, o_j (1 - o_j) x_{ji}$$

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

where $\delta_j = (t_j - o_j)\, o_j (1 - o_j)$.

Case 2 : Training Rule for Hidden unit weights Network.

$$\frac{\partial E_d}{\partial net_j} = \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j}$$

Here, k → output units

$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \times \frac{\partial o_j}{\partial net_j}$$

$$= \sum_{k \in Downstream(j)} -\delta_k w_{kj} \frac{\partial o_j}{\partial net_j} \quad \left[\because \frac{\partial net_k}{\partial o_j} = \frac{\partial x_{kj} w_{kj}}{\partial o_j} = \frac{\partial o_j w_{kj}}{\partial o_j}\right.$$

$$= \sum_{k \in Downstream(j)} -\delta_k w_{kj}\, o_j (1 - o_j)$$

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial net_j} x_{ji} = \eta\, o_j (1 - o_j) \sum_{k \in downstream(j)} -\delta_k w_{kj}\, o_j (1 - o_j)$$

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

$$\delta_j = o_j (1 - o_j) \sum_{k \in downstream(j)} \delta_k w_{kj}$$

APPENDIX C CODE

```python

# In[8]:


import numpy as np
import pandas as pd
from tqdm import tqdm, trange
import matplotlib.pyplot as plt


# In[24]:


def initialize_parameters(random_state=None, init_type='random'):
    '''
    init_type: str ('xe', 'he', 'lecun')
    '''
    assert init_type in ['xe', 'he', 'lecun', 'random']

    if random_state is not None:
        np.random.seed(random_state)

    if init_type == 'lecun':
        w_1 = np.random.uniform(low=-1/np.sqrt(784),
                                high=1/np.sqrt(784),
                                size=(100, 28*28))
        b_1 = np.random.uniform(low=-1/np.sqrt(784),
                                high=1/np.sqrt(784),
                                size=(100, 1))
        w_2 = np.random.uniform(low=-1/np.sqrt(100),
                                high=1/np.sqrt(100),
                                size=(10, 100))
        b_2 = np.random.uniform(low=-1/np.sqrt(100),
                                high=1/np.sqrt(100),
                                size=(10, 1))

    elif init_type == 'random':
        w_1 = np.random.uniform(-0.5, 0.5, size=(100, 28*28))
        b_1 = np.random.uniform(-0.5, 0.5, size=(100, 1))
        w_2 = np.random.uniform(-0.5, 0.5, size=(10, 100))
        b_2 = np.random.uniform(-0.5, 0.5, size=(10, 1))

    elif init_type == 'xe':
        w_1 = np.random.uniform(
            low=-np.sqrt(2/(100 + 784)),
                high=np.sqrt(2/(100 + 784)), size=(100, 28*28))
        b_1 = np.random.uniform(
            low=-np.sqrt(2/(100 + 784)),
                high=np.sqrt(2/(100 + 784)), size=(100, 1))
        w_2 = np.random.uniform(
            low=-np.sqrt(2/(10 + 100)),
                high=np.sqrt(2/(10 + 100)), size=(10, 100))
        b_2 = np.random.uniform(
            low=-np.sqrt(2/(10 + 100)),
                high=np.sqrt(2/(10 + 100)), size=(10, 1))

    elif init_type == 'he':
        w_1 = np.random.uniform(low=-np.sqrt(2/784),
                                high=np.sqrt(2/784),
                                size=(100, 28*28))
        b_1 = np.random.uniform(low=-np.sqrt(2/784),
                                high=np.sqrt(2/784),
                                size=(100, 1))
        w_2 = np.random.uniform(low=-np.sqrt(2/784),
                                high=np.sqrt(2/784),
                                size=(10, 100))
        b_2 = np.random.uniform(low=-np.sqrt(2/784),
                                high=np.sqrt(2/784),
                                size=(10, 1))

    params = {
        "w_1": w_1,
        "w_2": w_2,
        "b_1": b_1,
        "b_2": b_2
    }
    return params


def accuracy(y_true, y_pred):
    '''
    y_true: (number of output classes, total number of instances)
    '''

    m = y_true.shape[1]

    y_true = np.argmax(y_true, axis=0)
    y_pred = np.argmax(y_pred, axis=0)

    accuracy = (y_true == y_pred).sum() / m

    return accuracy


def ReLU(X):
    # shape = X.shape
    # temp = X.reshape(-1,1)
    # for i in range(len(temp)):
    #     temp[i] = max(0, temp[i])

    # temp = temp.reshape(shape)
    # return temp
    return np.maximum(X, np.zeros(X.shape))


def d_ReLU(X):
    return X >= 0


def categorical_cross_entropy(y_true, y_pred):
    '''
    y_true: (number of output classes, total number of instances)
    '''

    epsilon = 1e-6

    assert y_true.shape == y_pred.shape

    loss = (- (y_true) * np.log(y_pred
```

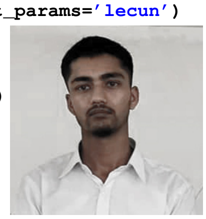
```
            + epsilon)).sum(axis=0)

    return loss.mean()


def tanh(X, back_prop=False):

    if back_prop:
        th = tanh(X)
        return 1 - np.power(th, 2)

    # np.clip(X, -250, 250)

    return np.tanh(X)


def softmax(X):
    '''
    X: (number of units, number of instances)

    return (units, number of instances)
    '''

    X = np.clip(X, -500, 500)

    divisor = np.exp(X).sum(axis=0)

    return np.exp(X) / divisor


# In[11]:


def accu_plot(epoch_data):
    # Accuracy plot
    plt.figure(figsize=(15, 10))
    plt.plot(epoch_data[:, 0], epoch_data[:, 2], color='red')
    plt.plot(epoch_data[:, 0], epoch_data[:, 4], color='blue')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.legend(['Train', 'Val'], loc='upper left')
    plt.title('Accuracy vs epoch graph of MNIST model')
    plt.show()


def loss_plot(epoch_data):
    plt.figure(figsize=(15, 10))
    plt.plot(epoch_data[:, 0], epoch_data[:, 1], color='red')
    plt.plot(epoch_data[:, 0], epoch_data[:, 3], color='blue')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend(['Train', 'Val'], loc='upper right')
    plt.title('Loss vs epoch graph of MNIST model')
    plt.show()


# In[4]:


def forward_vec(X, params, train=False):
    '''
    X: (number of features, number of training instances)
    '''

    m = X.shape[1]  # total number of instances

    w_1 = params["w_1"]
    w_2 = params["w_2"]
    b_1 = params["b_1"]
    b_2 = params["b_2"]

    z_1 = np.matmul(w_1, X) + b_1  # (number of units, number of instances)
    a_1 = tanh(z_1)  # (number of units, number of instances)

    z_2 = np.matmul(w_2, a_1) + b_2  # (number of units, input to the layer)
    out = softmax(z_2)

    cache = {
        "z_1": z_1,
        "z_2": z_2,
        "a_1": a_1,
        "a_2": out,
    }

    '''
    cache format:
    {
        "z_1": [number of units, number of instances],
        "z_2": [number of units, number of instances],
        "a_1": [number of units, number of instances],
        "a_2": [number of units, number of instances]
    }
    '''
    if not train:
        return out

    return out, cache


def backward_vec(X, y, params, cache, lr):
    '''
    X: array : (number of features, number of instances)
    y: array : (number of output classes, number of instances)
    params: dict : {
        "w_1":array(number of units in the layer, number of input feaures),
        "w_2":array(number of units in the layer, number of units in the previous layer),
        "b_1":array(number of units in the layer, 1),
        "b_2":array(number of units in the layer, 1)
    }

    cache : dict : {
        "z_1": (number of units , number of instances),
        "z_2": (number of output classes , number of instances),
        "a_1": (number of units , number of instances),
        "a_2": (number of output classes , number of instances),
    }
    '''

    m = X.shape[1]  # total number of
```

```
        instances

    z_1 = cache["z_1"]
    z_2 = cache["z_2"]
    a_1 = cache["a_1"]
    a_2 = cache["a_2"]

    w_1 = params["w_1"]
    w_2 = params["w_2"]
    b_1 = params["b_1"]
    b_2 = params["b_2"]

    dz_2 = a_2 - y  # (number of output classes, number of instances)
    # (number of output classes, number of units in the prev layer)
    dw_2 = np.matmul(dz_2, a_1.T) / m
    db_2 = np.sum(dz_2, axis=1, keepdims=True) / \
        m  # (number of output classes, 1)

    # (number of units in the layer, number of instances)
    dz_1 = np.matmul(w_2.T, dz_2) * tanh(z_1, back_prop=True)
    # (number of units in the layer, number of input features)
    dw_1 = np.matmul(dz_1, X.T) / m
    db_1 = np.sum(dz_1, axis=1, keepdims=True) / \
        m  # (number of units in the layer, 1)

    w_1 = w_1 - lr * dw_1
    w_2 = w_2 - lr * dw_2
    b_1 = b_1 - lr * db_1
    b_2 = b_2 - lr * db_2

    loss = categorical_cross_entropy(y, a_2)

    params = {
        "w_1": w_1,
        "w_2": w_2,
        "b_1": b_1,
        "b_2": b_2
    }

    return params, loss


def fit(X, y, iter=30, lr=0.01, val_set=None, init_params='random'):
    '''
    X: (number of features, number of instances)\n
    y: (number of output classes, number of instances)\n
    val_set : list : [
        validation instances: (number of features, validation set size),\n
        validation target: (number of output classes, validation set size)
    ]
    '''
    epoch_data = []

    params = initialize_parameters(init_type=init_params)

    print("training started...")

    for i in range(iter):
        out, cache = forward_vec(X, params, train=True)

        # return forward_vec(X, params, train=True)
        params, loss = backward_vec(X, y, params, cache, lr=lr)

        if val_set is not None:

            y_val = forward_vec(val_set[0], params)
            out = forward_vec(X, params)
            print(
                f"Epoch: {i+1}/{iter}\tloss: {loss} \t\t\ttrain accuracy: {accuracy(out, y)}\t\t\tvalidation accuracy: {accuracy(val_set[1], y_val)}")
            epoch_data.append([i+1, loss, accuracy(out, y), categorical_cross_entropy(
                y_val, val_set[1]), accuracy(val_set[1], y_val)])
        else:
            out = forward_vec(X, params)
            print(
                f"Epoch: {i+1}/{iter}\tloss: {loss} \t\ttrain accuracy: {accuracy(out, y)}")
    epoch_data = np.array(epoch_data)
    return params, epoch_data


# # Loading data and training
# In[5]:
data = pd.read_csv("digit_data.csv")
data = data.to_numpy()
np.random.shuffle(data)

X = data[:, 1:] / data[:, 1:].max()
y = data[:, 0]
target = np.zeros((y.shape[0], y.max()+1))

for i, row in enumerate(target):
    row[y[i]] = 1

y = target
print(X.shape, y.shape)


# In[6]:


x_train, y_train, x_test, y_test = X[:40000].T, y[:40000].T, X[40000:].T, y[40000:].T



# In[26]:


```

```
# Random
params, epoch_data = fit(x_train,
     y_train, lr=0.5, iter=500,
     val_set=[
                         x_test,
                              y_tes
                              init_
accu_plot(epoch_data)
loss_plot(epoch_data)

pred = forward_vec(x_test, params)
print(accuracy(y_test, pred))


# In[13]:


# he, the weights are initialized
     using a uniform distribution with
     bounds determined by the fan-in
     of each layer
params, epoch_data = fit(x_train,
     y_train, lr=0.5, iter=500,
     val_set=[
                         x_test,
                              y_test],
                              init_params='he')
accu_plot(epoch_data)
loss_plot(epoch_data)

pred = forward_vec(x_test, params)
print(accuracy(y_test, pred))


# In[14]:


# Lecun
params, epoch_data = fit(x_train,
     y_train, lr=0.5, iter=500,
     val_set=[
                         x_test,
                              y_test],
                              init_params='lecun')
accu_plot(epoch_data)
loss_plot(epoch_data)

pred = forward_vec(x_test, params)
print(accuracy(y_test, pred))


# In[15]:


# xe
params, epoch_data = fit(x_train,
     y_train, lr=0.5, iter=500,
     val_set=[
                         x_test,
                              y_test],
                              init_params='xe')
accu_plot(epoch_data)
loss_plot(epoch_data)

pred = forward_vec(x_test, params)
print(accuracy(y_test, pred))
```

APPENDIX D AUTHORS

ARAHANTA POKHAREL was born in 1999 in Biratnagar, Nepal. He is a dedicated individual with a strong passion for learning and research. Currently pursuing a Bachelor's degree in Computer Technology at the Institute of Engineering, Thapathali Campus, he is in the final year of his studies. Throughout his academic journey, he has developed a keen interest in machine learning and data science. Additionally, he has a curious mind and actively engages in quizzes and current affairs to stay updated with the latest information and is committed to acquiring new skills.



AMIT RAJ PANT Amit Raj Pant is a dedicated student pursuing his studies in the Department of Electronics and Computers at Thapathali Engineering Campus, Tribhuvan University, located in Kathmandu, Nepal. With a strong passion for technology, his interests include computer vision and machine learning on resource-constrained edge devices, which involves performing computational tasks on local devices rather than relying solely on remote servers.