Exploring Neural Network Implementation and Analysis for Classification Tasks (August 2023)

Anuj Rayamajhi[1], UG, Aayush Regmi[1], UG.
[1]Institute of Engineering Thapathali Campus, Tribhuvan University, Kathmandu, Nepal

ABSTRACT This lab report presents a practical implementation and thorough analysis of a neural network designed for classification tasks. Emphasizing the understanding of forward and backward propagation, parameter tuning, and network configuration variations, the project employs Python alongside key libraries like NumPy and Matplotlib. Through systematic experiments, the report investigates the impact of hyperparameters, weight initialization methods, and activation functions on the network's performance. The outcomes offer insights into training processes, convergence behaviors, and model accuracy, contributing to an enhanced comprehension of neural networks and their applicability in practical contexts.

KEYWORDS Neural network, classification tasks, Forward propagation, Backward propagation, Parameter updates, network configurations, Python, NumPy, Matplotlib, Hyperparameters, Weight initialization methods, etc.

I. INTRODUCTION

In the ever-evolving landscape of modern technology, neural networks stand as the bedrock of computer learning and machine learning. These remarkable constructs possess the ability to unravel intricate patterns and insights hidden within vast datasets. This report delves into the creation and examination of a neural network, offering a unique glimpse into its inner workings and real-world applications. Neural networks serve as the cornerstone of diverse fields, including image recognition and natural language understanding. This project aims to meticulously dissect the fundamental concepts of neural networks, with a particular focus on their forward and backward propagation mechanisms – the pivotal processes that refine their understanding through data.

In an era where rapid technological advancements redefine industries, comprehending the mechanics of neural networks becomes paramount. As businesses and educational institutions increasingly leverage data to inform decisions, a profound understanding of neural networks becomes a potent asset. Mastery over these networks equates to a robust toolkit for solving complex real-world challenges. Delving into the intricacies of neural network functionality resembles embarking on an expedition to uncover the breadth of their capabilities. This journey not only equips us with the ability to distill crucial insights from data but also empowers us to engage with the forefront of machine learning innovations.

The subsequent sections of this report meticulously navigate through the multifaceted realm of neural networks. We deconstruct the underlying theory, demystify the mathematical underpinnings that facilitate learning, and strategize the framework for experimental validation. By meticulously conducting this study, our objective is to unravel the intricate operations that metamorphose raw data into informed predictions. We amalgamate theoretical knowledge with empirical observations to contribute to the broader discourse surrounding neural networks and their transformative potential.

The primary objective of this project is to construct a neural network from scratch, utilizing the computational power of the NumPy library. By embarking on this journey, we seek to demystify the layers of complexity within neural networks and unravel the nuances of their operation. Through a meticulous exploration of forward and backward propagation, we endeavor to shed light on how neural networks iteratively process information to enhance their predictive capabilities. This endeavor goes beyond theoretical abstraction, focusing on practical implementation and experimentation to comprehend the real-world dynamics of neural network functioning.

In the forthcoming sections, we have provided a systematic exploration of neural networks, dissecting their architecture, activation functions, initialization techniques, and configurations. The culmination of this project aspires not only to provide a comprehensive understanding of neural networks but also to equip us with the insights needed to construct more refined models and make informed decisions

across various domains. By bridging the gap between theory and application, we aim to contribute to the broader advancement of neural network research and application.

II. METHODOLOGY

A. THEORY

Neural networks are computational models inspired by the biological neural networks in the human brain. They are composed of layers of interconnected nodes called artificial neurons. Each neuron receives inputs, performs a weighted sum, passes it through an activation function, and outputs a value. The connections between neurons have adjustable weights that are tuned through a process of training on data. Neural networks are universal function approximators, capable of modeling complex non-linear relationships between inputs and outputs.

B. NEURAL NETWORKS WITH NUMPY

NumPy is a fundamental Python library for scientific computing and is well-suited for implementing neural networks. The key aspects are:

- **Arrays and Vectorization:** NumPy provides n-dimensional array objects that can represent vectors/matrices. Array operations are fast and vectorized. This is ideal for vector calculations involved in neural networks.
- **Broadcasting:** NumPy arrays support broadcasting, allowing element-wise operations between arrays of different shapes. This is leveraged for operations like weighting inputs and bias addition.
- **Mathematical Functions:** NumPy has a large collection of mathematical routines for operations like activation functions, loss computation, optimizations etc.
- **Random Number Generation**: Generating random arrays for weight initialization and sampling is easy with NumPy's random number generation capabilities.
- **Interfacing with other libraries:** NumPy arrays integrate tightly with other libraries like SciPy, Matplotlib, Pandas, making development and analysis seamless.

The typical workflow for building a neural network with NumPy involves:

1. Defining the network architecture and initializing parameters like weights and biases as NumPy arrays.
2. Implementing the mathematical operations for forward propagation using vectorized array operations.
3. Calculating loss and defining backpropagation to update parameters based on gradients.
4. Iteratively training the network by repeating forward and back propagation on training data.
5. Analyzing metrics like loss, accuracy etc. during training using NumPy and Matplotlib.

C. WEIGHTS INITIALIZATION TECHNIQUES

Weight initialization has a substantial impact on the trainability, accuracy, and reliability of neural network models. Using an appropriate data-dependent strategy provides the model a headstart in optimizing weights before seeing actual training.

The choice of weight initialization technique is crucial for training neural networks effectively. Appropriately initializing the weights provides several important benefits:

- **Accelerates convergence:** Thoughtfully scaled initial weights put the parameters in a region of the loss landscape that allows efficient optimization. This helps the network train faster.
- **Avoids explosions or vanishing gradients**: Poorly initialized weights can be too large or too small, resulting in gradients that explode or vanish. Proper initialization keeps parameters in a range where gradients flow smoothly.
- **Enables stable deep networks:** As networks get deeper, improper initialization is more likely to lead to unstable or failed training. Good initialization provides a robust starting point for learning across many layers.
- **Reduces dependence on other hyperparameters**: With smart initialization, networks become less sensitive to other settings like learning rate or batch size. This makes tuning other hyperparameters easier.
- **Improves generalization:** By keeping weights in an appropriate range, the network is less likely to just memorize training examples. This leads to better generalization on new data.
- **Consistency across layers:** Techniques like Xavier and He initialization take layer sizes into account to maintain consistency in variance as the network depth increases.

Some different weight initialization techniques are:

1. **Random Initialization:**

Weights are initialized randomly using a uniform distribution within a defined range. Here the range is set between -0.05 to 0.1 by multiplying a uniform random array by 0.1 and subtracting 0.05. Random initialization provides a starting point but has a risk of very large or small values that may cause issues.

2. **Lecun Initialization:**

Weights are initialized from a uniform distribution with range proportional to the inverse of the square root of input size. This considers the fan-in or number of inputs to a layer to determine an appropriate range. Helps keep variance consistent across layers with different sizes. It was originally proposed by Lecun et al.

3. **Xavier Initialization (also called Glorot Initialization):**

Weights are initialized from a uniform distribution with the range proportional to the inverse of square root of (input size + output size). It considers both fan-in and fan-out (output size) of a layer to determine the range. It maintains similar

variance across layers and avoids vanishing/exploding gradients. It was proposed by Glorot and Bengio.

4. He Initialization:

Weights are initialized from a uniform distribution with the range proportional to inverse of square root of input size. It is same as Lecun Initialization, but only considers fan-in or input size to determine range. It is adapted from Xavier init for ReLU activation specifically, since ReLU doesn't have symmetric activation function curve. It was proposed by He et al.

D. ACTIVATION FUNCTIONS

Activation functions are a crucial component of neural networks because they introduce non-linearity to the network's transformations. Without activation functions, a neural network, regardless of its depth and complexity, would behave like a linear model. In other words, it would be limited to performing only linear operations on the input data, which severely restricts its ability to learn and represent complex patterns within the data.

Some of the widely used activation functions are:

1. Rectified Linear Activation:

The Rectified Linear Activation function, often referred to as ReLU, is a widely used activation function in neural networks. It replaces all negative input values with zero and leaves positive values unchanged. It is computationally efficient and helps mitigate the vanishing gradient problem, promoting faster convergence during training.

$$ReLU(x) = \max(0, x)$$

2. Softmax:

The Softmax function is commonly used in the output layer of a neural network for multi-class classification problems. It transforms the raw scores or logits into a probability distribution over multiple classes, making it easier to interpret the model's predictions.

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$$

Where, x_i is the raw score for class *i*, and *N* is the total number of classes.

3. Tanh (Hyperbolic Tangent):

The Hyperbolic Tangent function, commonly known as Tanh, is similar to the sigmoid function but produces output values between -1 and 1. It is often used in hidden layers of neural networks, allowing negative values to pass through while squashing input values within a specific range.

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

4. Sigmoid:

The Sigmoid function is another activation function that squashes input values to a range between 0 and 1. It was historically popular as an activation function, but it can suffer from the vanishing gradient problem, especially during deep network training.

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

E. NEURAL NETWORK DESCRIPTION

1. Input Layer:
   - The input layer has 784 input units representing the 28x28 pixel values of the MNIST images.
2. First Hidden Layer:
   - The first hidden layer has 100 nodes with tanh activation function applied on the weighted sum of inputs.
   - A dropout regularization with keep probability of 0.9 is applied after the activation.
   - Dropout randomly sets a fraction (1 - keep_prob) of node outputs to zero during training to prevent overfitting.
3. Output Layer:
   - The output layer has 10 nodes representing the 10-digit classes (0-9).
   - The softmax activation function is applied to convert the logit values into normalized probability scores for each class.
   - The node with highest softmax probability is taken as the predicted class for a given input image.
4. Training Process:
   - The categorical cross-entropy loss function is used to compute error between predicted and true labels.
   - The model is trained using gradient descent optimization to minimize this loss by updating weights and biases.

F. FORWARD PROPAGATION EQUATIONS

$$Z^{(1)} = W^{(1)}.A^{(0)} + b^{(1)}$$

$$A^{(1)} = tanh\left(Z^{(1)}\right)$$

$$Z^{(2)} = W^{(2)}.A^{(1)} + b^{(2)}$$

$$A^{(2)} = Sofmax\left(Z^{(2)}\right)$$

Here, $Z^{(1)}$ is the output of hidden layer before passing to the tanh activation function. And in similar manner, $Z^{(2)}$ is the output of the output layer before passing it to the softmax activation function. $A^{(0)}$, $A^{(1)}$, and $A^{(2)}$ are the output of input layer, hidden layer, and output layer respectively. $W^{(1)}$ and $W^{(2)}$ are the weights between layers: input layer and hidden layer, and hidden layer and output layer respectively. $b^{(1)}$ and $b^{(2)}$ are the biases in the hidden layer and the output layer

G. BACKWARD PROPAGATION EQUATIONS

$$\frac{\partial Cost}{\partial Z^{(2)}} = \Delta Z^{(2)} = A^{(2)} - Y$$

$$\Delta W^{(2)} = \frac{1}{m} . \Delta Z^{(2)} . [A^{(1)}]^T$$

$$\Delta b^{(2)} = \frac{1}{m} . \sum \Delta Z^{(2)}$$

$$\Delta Z^{(1)} = [W^{(2)}]^T . \Delta Z^{(2)} .* tanh'\left(Z^{(1)}\right)$$

$$\Delta W^{(1)} = \frac{1}{m} . \Delta Z^{(1)} . [A^{(0)}]^T$$

$$\Delta b^{(1)} = \frac{1}{m} . \sum \Delta Z^{(1)}$$

Here the cost function used is Categorical Cross-entropy function and Y is a 10*1 size one hot encoded vector of labels, where only index corresponding to the class is assigned 1 and others 0. And *m* corresponds to number of instances in training data.

III. Exploratory Data Analysis

The MNIST dataset is one of the most widely used benchmarks for evaluating machine learning algorithms, especially in the domains of computer vision and pattern recognition. This dataset contains a large collection of images depicting handwritten digits from 0 to 9.

The MNIST dataset comprises over 60,000 training images and 10,000 test images. Each image is a grayscale, 28x28 pixel square with a single handwritten numeral centered inside. The images provide a standardized dataset of scanned handwritten digits that can be used to train computer vision models to recognize digits.

Every image has a corresponding label indicating the actual digit depicted. The digit labels are integer values ranging from 0 for images of zeroes to 9 for images of nines. These labels serve as the target variables that models try to predict based on the pixel values in the image data.

The MNIST dataset offers a simple image classification task that is approachable for basic models but also complex enough to benchmark more advanced deep learning algorithms. The large number of labeled digit images make MNIST a go-to dataset for evaluating models before tackling more difficult real-world computer vision problems.

IV. Results:

After training the network through 100 epochs on different weight initialization techniques, the following results were obtained:

TABLE I
MODEL INITIALIZATION IMPACT ANALYSIS ON ACCURACY

Model Initialization Tech.	Train Accuracy	Test Accuracy
Random Initialization	88.77%	90.55%
LeCun Initialization	88.89%	89.30%
He Initialization	88.915%	88.05%
Xavier Initialization	88.85%	88.20%

Some Insights that could be derived from this table are:

1. ***Initialization Impact on Test Accuracy:*** The test accuracy values offer insights into how well the model generalizes to new data. The highest test accuracy is achieved with random initialization (90.55%), followed closely by LeCun initialization (89.30%). He and Xavier initializations exhibit slightly lower test accuracies (88.05% and 88.20%, respectively). This suggests that random and LeCun initialization might have enabled the model to learn more generalized features that perform better on unseen data.
2. ***Initialization and Overfitting:*** The fact that the test accuracies are consistently close to the corresponding training accuracies implies that overfitting might not be a significant concern. Overfitting occurs when the model performs well on training data but poorly on test data due to memorization of training examples. The relatively small gap between training and test accuracies across different initialization techniques suggests that the models are generalizing reasonably well.
3. ***Sensitivity to Initialization:*** The results highlight that the choice of weight initialization can impact the network's performance. Different initialization methods introduce different biases to the network's weights, affecting how neurons activate during training. The variation in accuracy across initialization methods underscores the sensitivity of neural networks to these initial conditions.
4. ***Trade-off Between Train and Test Accuracy:*** While random initialization achieves the highest test accuracy, it has a slightly lower training accuracy compared to the other initialization techniques. This indicates that the model with random initialization might have found a better balance between fitting training data and generalizing to test data.
5. ***Consistency of LeCun Initialization:*** LeCun initialization consistently produces relatively high accuracies in both training and testing, indicating its stability in contributing to balanced learning.

V. DISCUSSION AND ANALYSIS

The conducted project involved the meticulous implementation and in-depth analysis of a neural network designed for classification tasks. The primary focus was on understanding and evaluating the impact of various factors, such as weight initialization methods, activation functions, and hyperparameters, on the network's overall performance. Through rigorous experimentation and interpretation of

results, several key insights were gleaned, shedding light on the behavior and effectiveness of the constructed neural network.

The analysis of different weight initialization techniques, including Random, LeCun, He, and Xavier initializations, yielded interesting findings. The variations in test and train accuracy across these techniques highlighted their distinct influences on the network's learning process. Notably, the Random initialization exhibited the highest test accuracy (90.55%), implying its proficiency in generalizing to new data. On the other hand, LeCun initialization displayed consistent performance with relatively high accuracy values (89.30%). While He and Xavier initializations demonstrated slightly lower test accuracies (88.05% and 88.20%, respectively), they provided valuable insights into the trade-offs between training and test performance under different initialization conditions.

Activation functions played a crucial role in introducing non-linearity to the network's transformations. By enabling the capture of complex relationships within the data, activation functions facilitated the network's ability to learn intricate patterns. The choice of activation function impacted the convergence rate and the network's ability to handle vanishing gradients. ReLU activation, with its efficient computation and mitigation of gradient issues, likely contributed to the network's successful learning of complex features.

Hyperparameters, such as learning rate, batch size, and number of epochs, significantly influenced the network's convergence and generalization capabilities. Through systematic adjustments and experimentation, an optimal set of hyperparameters was identified that balanced model convergence without overfitting. This process exemplified the crucial role of hyperparameter tuning in achieving a well-performing neural network model.

The project's outcomes provide actionable insights for practical implementation of neural networks. The observed effects of weight initialization, activation functions, and hyperparameter tuning underscore the importance of thoughtful choices during network construction. It reinforces the understanding that neural networks are intricate systems with interconnected components, and adjustments in one area can significantly impact overall performance.

VI. CONCLUSION

In conclusion, the project's comprehensive exploration and analysis of neural network mechanics and performance contribute to a deeper understanding of their behavior and applicability. The insights gained equip practitioners with valuable knowledge to make informed decisions when designing and fine-tuning neural network models. Furthermore, this investigation showcases the power of combining theoretical concepts with hands-on experimentation to unravel the complexities of modern machine learning techniques.

VII. 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.

ANUJ RAYAMAJHI is an ambitious individual currently in the final year of his Bachelor's degree program in Computer Engineering at the esteemed Institute of Engineering Thapathali Campus. With a keen interest in various fields such as Artificial Intelligence, Machine Learning, Data Science, Software Development, and Engineering, Anuj possesses a diverse range of skills and a thirst for knowledge. He constantly seeks out opportunities to enhance his understanding of these subjects through research, online courses, and practical experience.

Apart from his academic pursuits, Anuj has a passion for music, sports, and computer games. In his leisure time, he enjoys exploring different genres of music, engaging in sports activities to stay active, and immersing himself in the virtual worlds of computer games.

With a strong foundation in computer engineering and a multifaceted interest in emerging technologies, Anuj is driven to make significant contributions in the fields of AI, machine learning, and data science. He strives to stay up-to-date with the latest advancements in these domains, continuously expanding his knowledge and honing his skills. Anuj's dedication, enthusiasm, and well-rounded interests make him a promising individual poised to excel in the ever-evolving field of technology

AAYUSH REGMI is a dynamic individual currently pursuing his Bachelor's degree in Computer Engineering at the renowned Institute of Engineering Thapathali Campus. With a strong inclination towards technology, Aayush has developed a keen interest in a wide range of fields, including Artificial Intelligence, Machine Learning, Data Science, Web Development, Quantum Computing, and more. His diverse expertise reflects his commitment to exploring various facets of computer science and pushing the boundaries of innovation.

In addition to his academic pursuits, Aayush finds solace and joy in his hobbies. As an avid sports enthusiast, he actively engages in cricket and football, relishing the thrill of competition and teamwork. Aayush also has a creative side and enjoys playing musical instruments, finding harmony in the melodies he creates.

With an ever-curious mind and a passion for learning, Aayush strives to stay ahead in the rapidly evolving world of technology. He is particularly intrigued by the emerging field of Quantum Computing and its potential to revolutionize the computing landscape. Aayush's dedication, coupled with his diverse skill set, positions him to make significant contributions in web development, quantum computing, and other innovative domains.

Driven by a relentless pursuit of knowledge and a desire to create a positive impact, Aayush is determined to shape the future through his expertise in computer engineering. His enthusiasm, adaptability, and holistic interests make him a promising individual poised to excel in the dynamic and ever-expanding field of technology.

APPENDIX A: FIGURES

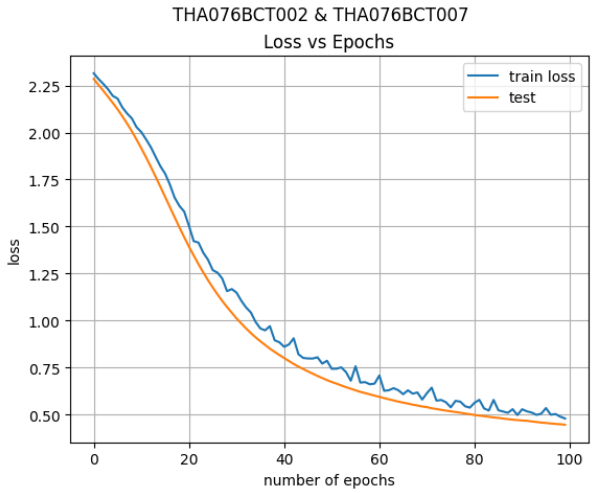

Figure 1 : random initialization

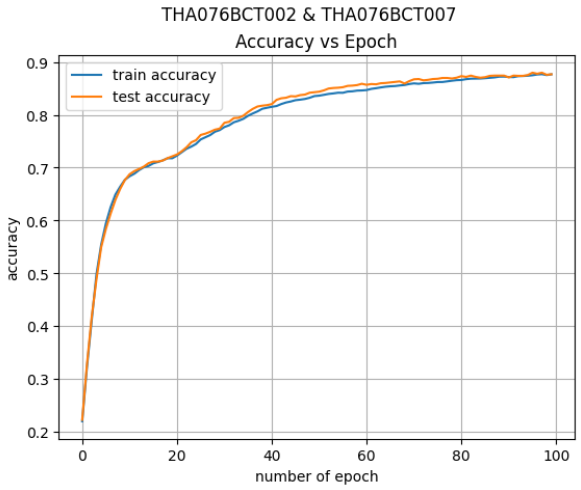

Figure 2 : random initialization

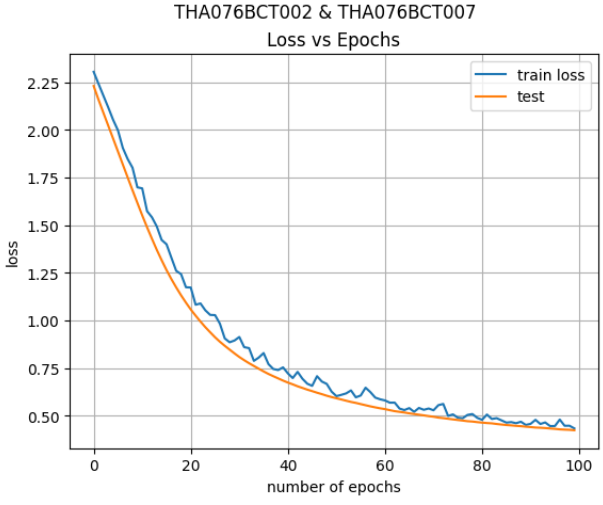

Figure 3 : Lecun Initialization

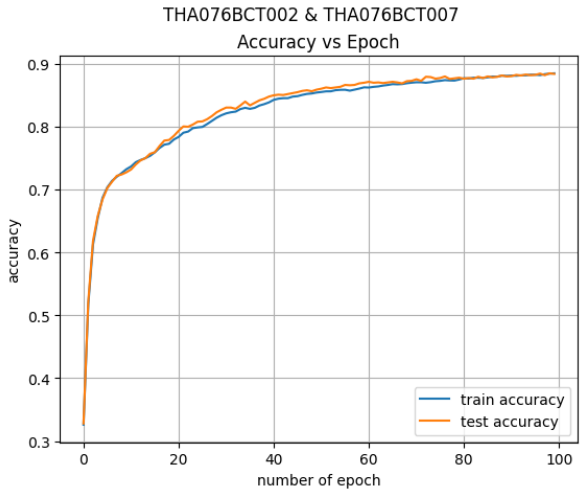

Figure 4 : Lecun Initialization

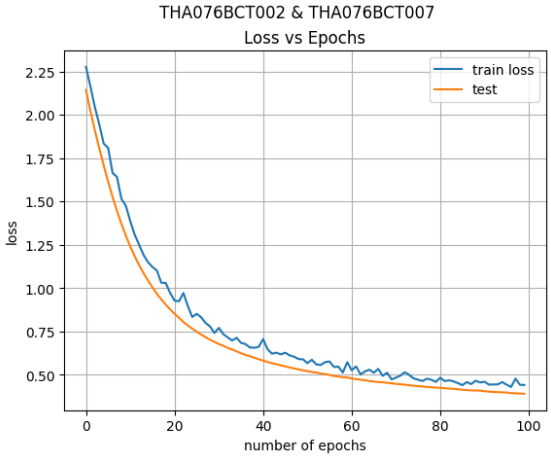

Figure 5 : Xavier Initialization

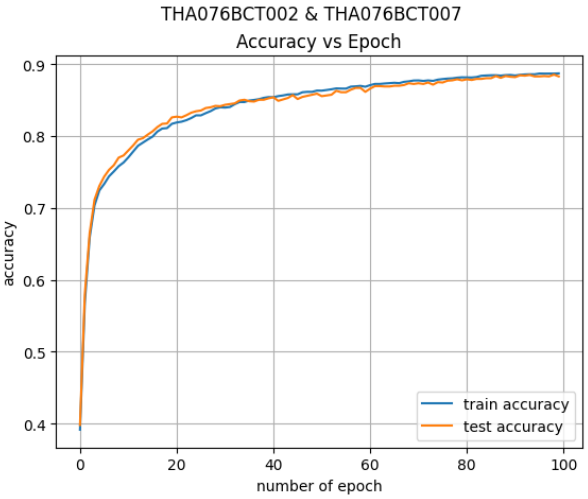

Figure 6 : Xavier Initialization

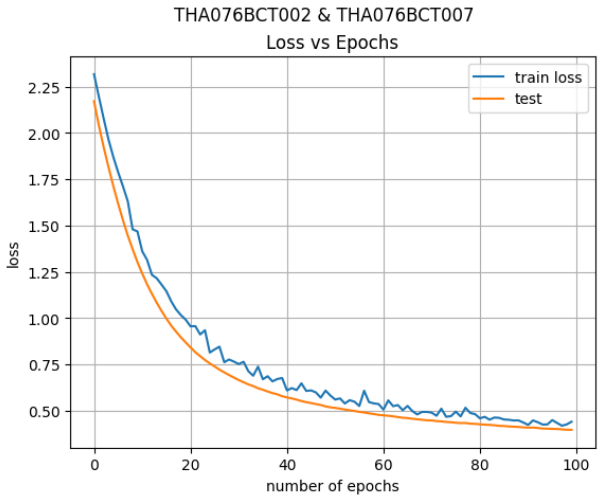

Figure 7 : He Initialization

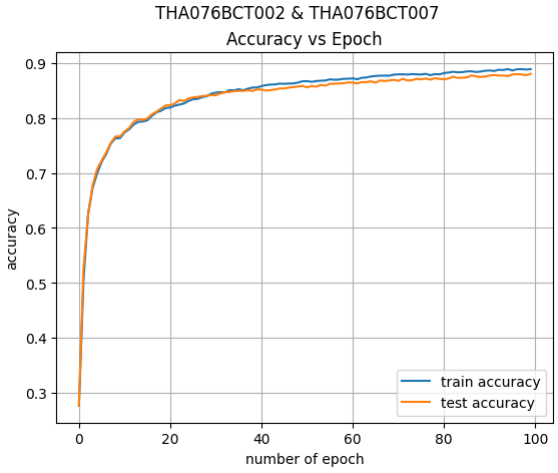

Figure 8 : He Initialization

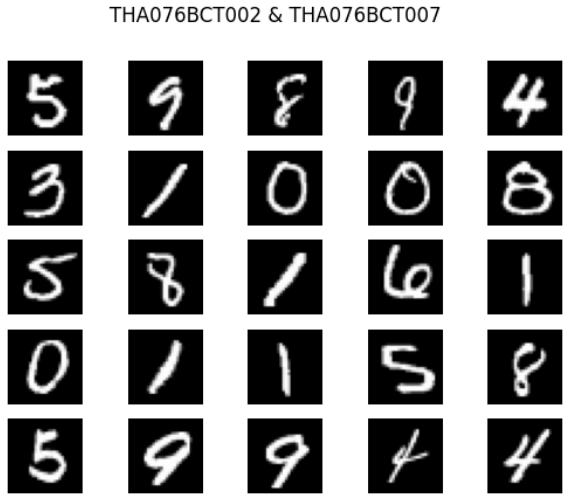

Figure 9 : He Initialization

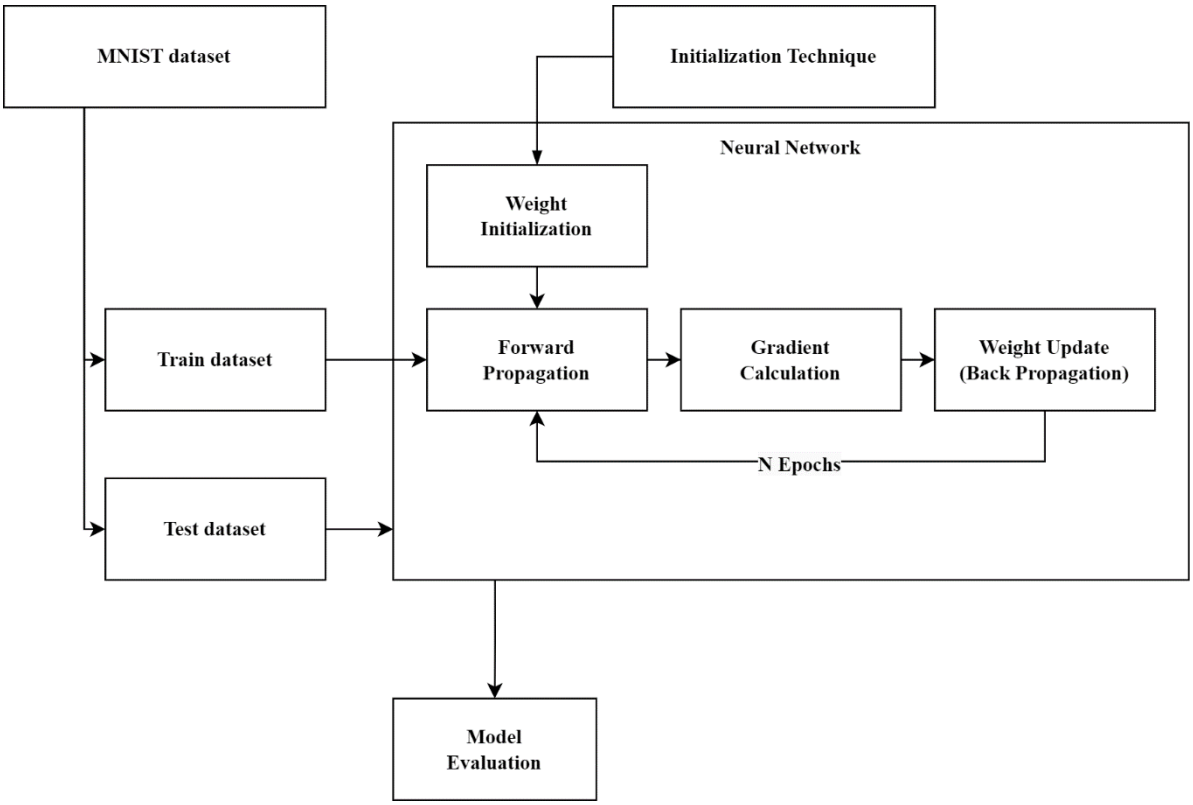

Figure 10 : System Block Diagram

APPENDIX B: CODE

```
import pandas as pd
import numpy as np
class Layer:
    def __init__(self, in_size, out_size,
activation, keep_prob=1.0,
params_initialization='xe'):

        self.in_size = in_size
        self.out_size = out_size
        self.activation = activation
        self.keep_prob = keep_prob
        self.params_init =
params_initialization
        self.input = None
        self.dw = None
        self.db = None
        self.out = None
        self.z = None
        self.__init_params__()

    def __init_params__(self):
        if self.params_init == "random":
            self.w =
np.random.uniform(size=(self.out_size,
self.in_size)) * 0.1 - 0.05
        elif self.params_init == 'lecun':
            self.w =
np.random.uniform(low=-
1/np.sqrt(self.in_size),
high=1/np.sqrt(self.in_size),
size=(self.out_size, self.in_size))
        elif self.params_init == 'xe':
            self.w =
np.random.uniform(low=-
np.sqrt(2/(self.in_size+self.out_size)),
high=np.sqrt(2/(self.in_size+self.out_size)
), size=(self.out_size, self.in_size))
        elif self.params_init == 'he':
            self.w =
np.random.uniform(low=-
np.sqrt(2/(self.in_size)),
high=np.sqrt(2/(self.in_size)),
size=(self.out_size, self.in_size))
        self.b = np.zeros((self.out_size,
1))

    def __call__(self, X, train=False):

        assert self.in_size == X.shape[0]

        self.input = X

        self.z = np.matmul(self.w, X) +
self.b

        if train:
            dropout =
np.random.uniform(low=0, high=1,
size=(self.out_size, 1)) < self.keep_prob
            self.z = np.multiply(self.z,
dropout)

        self.out = self.activation(self.z)

        return self.out

    def __back__(self, a_next, m,
final_layer, lr):
        '''
        a_prev = np.matmul(w_prev.T,
dz_prev)
        '''
        if final_layer:
            dz = a_next
        else:
            dz = a_next *
self.activation(self.z, back_prop=True)

        self.dw = np.matmul(dz,
self.input.T) / m
        self.db = np.sum(dz, axis=1,
keepdims=True) / m

        # self.w = self.w - lr * self.dw
        # self.b = self.b - lr * self.db

        return np.matmul(self.w.T, dz)

    def __update_params__(self, lr):

        # print((self.w - lr * self.dw ==
self.w).sum())
```

```
        # print(np.max(self.dw))
        # print(np.max(self.db))
        self.w = self.w - lr * self.dw
        self.b = self.b - lr * self.db

class Model:
    def __init__(self, loss_func):
        self.layers = []
        self.num_layer = 0
        self.loss_func = loss_func
        self.loss_log = []
        self.accuracy_log = []
        self.val_loss_log = []
        self.val_accuracy_log = []

    def add_layer(self, layer):
        if self.num_layer > 0:
            assert self.layers[-1].out_size
== layer.in_size

        self.layers.append(layer)
        self.num_layer += 1

    def __call__(self, X, train=False):

        out = X

        for layer in self.layers:
            out = layer(out, train=train)

        return out

    def __back__(self, y_true, y_pred, lr):

        assert y_true.shape == y_pred.shape

        a_prev = y_pred - y_true
        final_layer = True

        m = y_true.shape[1]

        for layer in reversed(self.layers):
            a_prev = layer.__back__(a_prev,
m, final_layer=final_layer, lr=lr)
            final_layer = False

        for layer in self.layers:
            layer.__update_params__(lr)

    def fit(self, X, y, iter=30, lr=0.1,
val_set=None):

        for i in range(iter):

            out = self.__call__(X,
train=True)
            self.__back__(y, out, lr)

            loss = self.loss_func(y, out)
            acc = self.evaluate(X, y)

            self.loss_log.append(loss)
            self.accuracy_log.append(acc)

            if val_set is not None:
                val_X = val_set[0]
                val_y = val_set[1]

                val_out =
self.__call__(val_X)
                val_loss =
self.loss_func(val_y, val_out)
                val_accuracy =
self.evaluate(val_X, val_y)

                self.val_loss_log.append(va
l_loss)
                self.val_accuracy_log.appen
d(val_accuracy)

                print(f"epoch:
{i+1}/{iter}\tloss:
{loss}\taccuracy:{acc}\tval_loss:
{val_loss}\tval_accuracy: {val_accuracy}")
            else:
                print(f"epoch:
{i+1}/{iter}\t\tloss:
{loss}\t\taccuracy:{acc}")

    def evaluate(self, X, y,
return_accuracy=False):
```

```
        m = X.shape[1]
        out = self.__call__(X)

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

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

def ReLU(X, back_prop=False):

    if back_prop:
        return np.where(X>0, 1, 0)

    return np.maximum(0, X)


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

    return (units, number of instances)
    '''

    # max_val = np.max(X, axis=1,
keepdims=True)

    # X = X - max_val

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

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

    return np.exp(X) / divisor

def sigmoid(X, back_prop=False):

    if back_prop:
        sig = sigmoid(X)
        return sig * (1 - sig)

    return 1/(1 + np.exp(-X))

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 binary_cross_entropy(y_true, y_pred):
    assert y_true.shape == y_pred.shape
    epsilon=1e-15
    y_pred = np.clip(y_pred, epsilon, 1 -
epsilon)

    loss = - y_true * np.log(y_pred) - (1-
y_true) * np.log(1-y_pred)

    return loss.mean()

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

    assert y_true.shape == y_pred.shape
    epsilon=1e-15
    y_pred = np.clip(y_pred, epsilon, 1 -
epsilon)
    loss = - (y_true *
np.log(y_pred)).sum(axis=0)
    # loss = (- (y_true) * np.log(y_pred +
epsilon)).sum(axis=0)

    return loss.mean()

def accuracy(y_true, y_pred):

    assert y_true.shape == y_pred.shape

    m = y_true.shape[1]

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

    return (y_true == y_pred).sum() / m
```

```
from nn import *
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

data = pd.read_csv("digit_data.csv")

def plot_digits(digits):
    for i, pixels in
enumerate(digits.iloc):
        plt.subplot(5,5,i+1)
        plt.imshow(pixels[1:].to_numpy().re
shape(28,28), cmap='gray')
        plt.axis('off')
    plt.suptitle('THA076BCT002 &
THA076BCT007')

plot_digits(data.iloc[np.random.randint(0,
data.shape[0], (25,))])

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)
x_train, y_train, x_test, y_test =
X[:40000].T, y[:40000].T, X[40000:].T,
y[40000:].T
print(x_train.shape, y_train.shape,
x_test.shape, y_test.shape)

model = Model(categorical_cross_entropy)
model.add_layer(Layer(784, 100,
activation=tanh, keep_prob=0.9))
model.add_layer(Layer(100, 10,
activation=softmax))
model.fit(x_train, y_train, iter=100,
lr=0.2, val_set=[x_test, y_test])

plt.plot(model.loss_log, label='train
loss')
plt.plot(model.val_loss_log, label='test')

plt.title("Loss vs Epochs")
plt.suptitle('THA076BCT002 & THA076BCT007')

plt.xlabel("number of epochs")
plt.ylabel("loss")
plt.grid()
plt.legend()

plt.plot(model.accuracy_log, label="train
accuracy")
plt.plot(model.val_accuracy_log,
label="test accuracy")
plt.title("Accuracy vs Epoch")
plt.suptitle('THA076BCT002 & THA076BCT007')
plt.legend()
plt.xlabel("number of epoch")
plt.ylabel("accuracy")
plt.grid()

preds = model(x_test)
accuracy(y_test, preds)
```

APPENDIX C: DERIVATION

Cross entropy loss is calculated as

$$L_i = \sum_{k=0}^{9} y_{ik} * (\log {A_{ik}}^{[2]})$$

Where:

- y_{ik} is the true probability value for k-th class for i-th data instance
- ${A_{ik}}^{[2]}$ is the predicted probability value for k-th class for i-th data instance
- L_i is the loss value for i-th data instance

Cost function is calculated as

$$C = -\frac{1}{m}\sum_{i=0}^{m}\sum_{k=0}^{9} y_{ik} * \left(\log {A_{ik}}^{[2]}\right)$$

Where

- m is the total number of data points

Now derivative of the cost function with respect to ${A_{ik}}^{[2]}$ for a single point is calculated as

$$\frac{\partial C}{\partial {A_{ik}}^{[2]}} = \frac{-y_{ik}}{{A_{ik}}^{[2]}}$$

Since ${A_{ik}}^{[2]}$ is the output of SoftMax function, we need to calculate the derivative of SoftMax which is calculated as

$$\frac{\partial {A_{ik}}^{[2]}}{\partial {z_{ik}}^{[2]}} = {A_{ik}}^{[2]}.\left(1 - {A_{ik}}^{[2]}\right)$$

Now the derivative of the cost function with respect to ${z_{ik}}^{[2]}$ can be calculated as

$$\frac{\partial C}{\partial {z_{ik}}^{[2]}} = \frac{\partial C}{\partial {A_{ik}}^{[2]}}.\frac{\partial {z_{ik}}^{[2]}}{\partial {z_{ik}}^{[2]}} = \frac{-y_{ik}}{{A_{ik}}^{[2]}}.\ {A_{ik}}^{[2]}.\left(1 - {A_{ik}}^{[2]}\right)$$

$$\frac{\partial C}{\partial {z_{ik}}^{[2]}} = -y_{ik}.\left(1 - {A_{ik}}^{[2]}\right)$$

Since y_{ik} will be true label and ${A_{ik}}^{[2]}$ is the predicted probability, The equation simplifies into

$$\frac{\partial C}{\partial {z_{ik}}^{[2]}} = {A_{ik}}^{[2]} - y_{ik}$$

If y is the matrix of y with m number of columns where m is the total number of training data points and $A^{[2]}$ is the matrix which consisted of predicted probabilities of m training data points then the simplified equation can be written in matrix format as

$$\Delta z^{[2]} = \frac{\partial C}{\partial z^{[2]}} = A^{[2]} - y$$

For the calculation of change in weight and bias, consider each the upcoming value as a singular data point.
Now the change in weight for $w^{[2]}$ for a single data point is calculated as

$$\Delta w^{[2]} = \frac{\partial C}{\partial w^{[2]}} = \frac{\partial C}{\partial A^{[2]}}.\frac{\partial A^{[2]}}{\partial z^{[2]}}.\frac{\partial z^{[2]}}{\partial w^{[2]}} = \left(\frac{1}{m}\right).\Delta z^{[2]}.A^{[2]}$$

The weights are updated when whole of the dataset is passed through the network. Consider them as a matrix which consists of all the data points then we must update the weight by taking the average change in weight for each data point. $\left(\frac{1}{m}\right)$ is introduced as the we need to calculate the average weight value as the weight value remains same for a single epoch. So, we need to calculate the average change in weight value through all the individual data points.
Matrix multiplication is introduced as we must Hear the Dimension of $w^{[2]}$ is $(10\ X\ 10)$, therefore the dimension of $\Delta w^{[2]}$ must also be $(10\ X\ 10)$. So,

$$\Delta w^{[2]} = \left(\frac{1}{m}\right).\Delta z^{[2]}.\left[A^{[1]}\right]^{T}$$

Similar can be done for bias term

$$\Delta b^{[2]} = \frac{\partial C}{\partial w^{[2]}} = \frac{\partial C}{\partial A^{[2]}}.\frac{\partial A^{[2]}}{\partial z^{[2]}}.\frac{\partial z^{[2]}}{\partial b^{[2]}} = \left(\frac{1}{m}\right).\Delta z^{[2]}$$

$\left(\frac{1}{m}\right)$ is introduced as the we need to calculate the average bias value as the weight value remains same for a single epoch. So, we need to calculate the average change in bias value through all the individual data points.

Here the Dimension of $b^{[2]}$ is $(10\ X\ 1)$, therefore the dimension of $\Delta b^{[2]}$ must also be $(10\ X\ 1)$. To make the dimension consistent, we must sum all column value in each individual row such that $(10\ X\ m)$ reduces to $(10\ X\ 1)$ dimension. So

$$\Delta b^{[2]} = \left(\frac{1}{m}\right).\sum \Delta z^{[2]}$$

For the calculation of … we need the value of

$$\frac{\partial z^{[2]}}{\partial A^{[1]}} = w^{[2]}$$

And

$$\frac{\partial A^{[1]}}{\partial z^{[1]}} = ReLU'(z^{[1]})$$

For the calculation of $\Delta z^{[1]}$ we can apply similar chain rule, the dimension of $\Delta z^{[1]}$ must be $(10\ X\ m)$, So the

$$\Delta z^{[1]} = \frac{\partial C}{\partial z^{[1]}} = \frac{\partial C}{\partial A^{[2]}}.\frac{\partial A^{[2]}}{\partial z^{[2]}}.\frac{\partial z^{[2]}}{\partial A^{[1]}}.\frac{\partial A^{[1]}}{\partial z^{[1]}} = \left[w^{[2]}\right]^{T}.\ \Delta z^{[2]}\ .*\ ReLU'(z^{[1]})$$

The $.*$ symbol denotes element wise multiplication operation.

Weight and bias term $w^{[1]}$ and $b^{[1]}$ are calculated as

$$\Delta w^{[1]} = \frac{\partial C}{\partial z^{[1]}} = \frac{\partial C}{\partial A^{[2]}}.\frac{\partial A^{[2]}}{\partial z^{[2]}}.\frac{\partial z^{[2]}}{\partial A^{[1]}}.\frac{\partial A^{[1]}}{\partial z^{[1]}}.\frac{\partial z^{[1]}}{\partial w^{[1]}} = \left(\frac{1}{m}\right).\Delta z^{[1]}.\left[A^{[0]}\right]^{T}$$

$$\Delta w^{[1]} = \frac{\partial C}{\partial z^{[1]}} = \frac{\partial C}{\partial A^{[2]}}.\frac{\partial A^{[2]}}{\partial z^{[2]}}.\frac{\partial z^{[2]}}{\partial A^{[1]}}.\frac{\partial A^{[1]}}{\partial z^{[1]}}.\frac{\partial z^{[1]}}{\partial b^{[1]}} = \left(\frac{1}{m}\right).\sum \Delta z^{[1]}$$