Date of publication August 25, 2023
Digital Object Identifier TBD

Identification of Digits Using Different Configuration ANNs

PRATIGYA PAUDEL[1], SUSHANK GHIMIRE[1]
[1]Institute of Engineering, Thapathali Campus, Bagmati 44600 Nepal (e-mail: pratigyapaudel0@gmail.com)
Corresponding author: Pratigya Paudel (e-mail: pratigyapaudel0@gmail.com).

"This work was completed as a part of a college practical for Data Mining (CT725)."

ABSTRACT Artificial Neural Networks (ANNs) have garnered significant attention due to their effectiveness in various machine learning tasks, particularly image classification. ANNs are renowned for their capability to learn complex patterns and relationships from data, and they have revolutionized the field of computer vision. In this study, we delve into the application of ANNs to classify images from the MNIST dataset. The MNIST dataset comprises a collection of handwritten digits, each represented as a grayscale image. Our objective is to harness the power of ANNs to accurately classify these images into their respective numerical classes (0 to 9). This research aims to explore the potential of ANNs in discerning intricate features and characteristics that distinguish different digits. By utilizing layers of interconnected neurons, ANNs can capture hierarchical features from raw pixel values. This involves learning and fine-tuning weights that enable the network to recognize distinctive patterns in the images. The training process involves feeding the network a large set of labeled images and iteratively adjusting the weights to minimize the classification error. The paper also focuses on comparing the results on the models by using different weight initialization approaches. The Kaiming and Xavier initialization approaches bring different results to the vanilla approach of training a model. Furthermore, the fundamentals of a neural network with dropout layers are designed from the scratch to test their performance.

INDEX TERMS Neurons, Layers, Supervised Machine Learning

I. INTRODUCTION

ANN ARTIFICIAL NEURAL NETWORKS are a class of machine learning models inspired by the structure and functioning of the human brain's neural networks. Comprising interconnected nodes, or "neurons," arranged in layers, ANNs have the capacity to learn complex patterns and relationships from data. They excel in tasks ranging from image and speech recognition to natural language processing. Through a process called training, ANNs adjust their internal parameters, or weights, to minimize prediction errors. This adaptability enables ANNs to generalize well to new data, making them effective tools for classification, regression, and even more advanced tasks like generative modeling.

Artificial Neural Networks (ANNs) comprise distinct layers, each serving a unique role in the network's learning process. The input layer accepts raw data and passes it to subsequent layers. Hidden layers, situated between the input and output layers, are responsible for extracting and learning relevant features from the input. Neurons within these layers process information through weighted connections, applying activation functions that introduce non-linearity to the model. Hidden layers enable ANNs to capture intricate relationships within data. Finally, the output layer produces the network's predictions or classifications. The architecture's depth and arrangement of layers influence the model's complexity and its ability to learn complex patterns. This layered structure empowers ANNs to tackle diverse machine learning tasks, from image recognition to language processing, by progressively transforming input data into meaningful predictions or decisions.

Forward propagation is a crucial step in training neural networks, where input data is passed through the network's layers, progressively transformed by weighted connections and activation functions to produce predictions or output values. Backward propagation, also known as backpropagation, follows forward propagation and involves calculating gradients of the loss function with respect to network weights. This process enables the network to understand how each weight contributes to the overall error, facilitating weight updates to minimize the loss. Gradient descent, a fundamental optimization technique, leverages these gradients to iteratively adjust weights in the opposite direction of the gradient, gradually approaching the optimal values that minimize the loss function. These interconnected processes play a central role in training neural networks, enabling them to learn and improve their predictions over successive iterations.

Activation functions are fundamental components in Artificial Neural Networks (ANNs) that introduce non-linearity to the model, enabling it to capture complex patterns and relationships within data. These functions determine the output of a neuron based on the weighted sum of its inputs, influencing whether the neuron should "fire" or remain inactive. Common activation functions include the sigmoid, hyperbolic tangent (tanh), and rectified linear unit (ReLU), each with distinct properties. Sigmoid and tanh functions squash values into a specific range, while ReLU provides a simple thresholding mechanism that accelerates training by mitigating the vanishing gradient problem. The choice of activation function significantly impacts network behavior, affecting training speed, convergence, and overall model performance.

Kaiming and Xavier initializations are essential techniques in training Artificial Neural Networks (ANNs) to ensure efficient and stable learning. Xavier initialization focuses on maintaining consistent variance of activations across layers, particularly benefiting networks with linear-like activation functions. On the other hand, Kaiming initialization is tailored for networks using rectified linear units (ReLUs) and their variants, aiming to prevent the "dying ReLU" problem by adapting weight initialization to suit these non-linear activations. Both methods play a critical role in mitigating issues like vanishing or exploding gradients, ultimately enhancing the convergence and performance of ANNs during training.

The MNIST dataset is a widely recognized benchmark in the field of machine learning and computer vision. It consists of a collection of grayscale images depicting handwritten digits ranging from 0 to 9. Comprising a training set of 60,000 images and a test set of 10,000 images, MNIST serves as a foundational resource for developing and evaluating image classification algorithms. Each image is 28x28 pixels in size, representing a digit drawn by various individuals. The dataset's simplicity and accessibility make it a popular choice for introducing newcomers to image recognition tasks and for benchmarking the performance of various algorithms. Despite its basic nature, MNIST remains a valuable resource for testing and refining machine learning techniques, contributing to the advancement of the field.

II. METHODOLOGY

A. THEORY

Artificial Neural Networks (ANNs) are machine learning models inspired by the human brain's neural networks. They consist of interconnected layers of nodes that learn complex patterns from data. Neurons are fundamental units in artificial neural networks (ANNs) inspired by their biological counterparts in the human brain. Each neuron receives input signals, processes them using weighted connections, applies an activation function, and generates an output signal. These connections, known as synapses, have associated weights that determine the importance of each input. The weighted sum of inputs, along with a bias term, is then passed through an activation function to introduce non-linearity, enabling the neuron to capture complex patterns in data. Neurons collectively form layers within neural networks, facilitating information propagation and transformations.

B. INSTRUMENTATION TOOLS

The entirety of the process is done using Python. Google Colab, short for Google Colaboratory, is an online platform provided by Google for running and sharing Jupyter notebook environments and it was used for all of the coding. Google colab provides a number of built-in functions for data analysis. The process of building and training has been carried out using a number of available functions within the numpy library. The dataset is visualized using pandas. The results are then visualized using different visualization tools like Seaborn and matplotlib.

C. WORKING PRINCIPLE

1) Neural Network

The output of a neuron in an artificial neural network is calculated by summing the weighted inputs, adding a bias term, and then passing the result through an activation function:

$$\text{Output} = \text{Activation}\left(\sum_{i=1}^{n}(w_i \cdot x_i) + b\right) \tag{1}$$

Where w_i are the weights, x_i are the input values, b is the bias, and Activation is the chosen activation function.

2) Forward Propagation

Forward propagation is the process in neural networks where inputs are transformed layer by layer, producing an output through weighted connections and activation functions.

$$\begin{aligned}
Z^{[1]} &= W^{[1]}X + b^{[1]}\\
A^{[1]} &= g_{\text{ReLU}}(Z^{[1]})\\
Z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]}\\
A^{[2]} &= g_{\text{softmax}}(Z^{[2]})
\end{aligned}$$

3) Backpropagation

Backpropagation is the iterative process in neural networks where the gradients of the loss function with respect to the network's parameters are calculated. These gradients guide the adjustment of weights and biases during optimization.

$$\begin{aligned}
dZ^{[2]} &= A^{[2]} - Y \\
dW^{[2]} &= \frac{1}{m} dZ^{[2]} A^{[1]T} \\
db^{[2]} &= \frac{1}{m} \text{np.sum}(dZ^{[2]}, \text{axis} = 1, \text{keepdims=True}) \\
dZ^{[1]} &= W^{[2]T} dZ^{[2]} \cdot g'_{\text{ReLU}}(Z^{[1]}) \\
dW^{[1]} &= \frac{1}{m} dZ^{[1]} X^T \\
db^{[1]} &= \frac{1}{m} \text{np.sum}(dZ^{[1]}, \text{axis} = 1, \text{keepdims=True})
\end{aligned}$$

4) Parameter Update

After computing the gradients during backpropagation, the network's parameters are updated using an optimization algorithm like stochastic gradient descent (SGD).

$$\begin{aligned}
W^{[2]} &:= W^{[2]} - \alpha dW^{[2]} \\
b^{[2]} &:= b^{[2]} - \alpha db^{[2]} \\
W^{[1]} &:= W^{[1]} - \alpha dW^{[1]} \\
b^{[1]} &:= b^{[1]} - \alpha db^{[1]}
\end{aligned}$$

Where α is the learning rate, $W^{[i]}$ are the weight matrices, and $b^{[i]}$ are the bias vectors for each layer i.

5) Activation Functions

Activation functions introduce non-linearity in neural networks, enabling them to capture complex patterns and relationships in data.

Tanh Activation: The Hyperbolic Tangent (tanh) activation function maps the input values to the range of -1 to 1, providing a symmetric output that can model both positive and negative values effectively.

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

Sigmoid Activation: The Sigmoid activation function produces outputs between 0 and 1, effectively squashing the input values and is commonly used in binary classification problems.

$$g_{\text{sigmoid}}(z) = \frac{1}{1 + e^{-z}}$$

ReLU Activation: The Rectified Linear Unit (ReLU) is a widely used activation function that outputs the input value if it's positive, and zero otherwise.

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

Softmax Function: The softmax function is commonly used in the output layer for multi-class classification. It converts a vector of raw scores into a probability distribution.

$$g_{\text{softmax}}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

Where z_i is the raw score of class i, N is the total number of classes, and e is the base of the natural logarithm.

6) Loss Functions

Loss functions quantify the difference between the predicted values and the actual targets, guiding the optimization process during training.

Mean Squared Error (MSE): MSE computes the average of the squared differences between predicted and actual values. It's commonly used for regression problems.

$$\text{MSE} = \frac{1}{m} \sum_{i=1}^{m} (y_{\text{pred}}^{(i)} - y^{(i)})^2$$

Cross-Entropy Loss (Binary Classification): Cross-entropy loss measures the dissimilarity between the predicted probabilities and the true binary labels. It's often used in binary classification problems.

$$\text{Cross-Entropy} = -\frac{1}{m} \sum_{i=1}^{m} [y^{(i)} \log(y_{\text{pred}}^{(i)}) + (1 - y^{(i)}) \log(1 - y_{\text{pred}}^{(i)})]$$

Categorical Cross-Entropy Loss (Multiclass Classification): For multiclass classification, categorical cross-entropy loss computes the sum of the cross-entropy losses for each class.

$$\text{Categorical Cross-Entropy} = -\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{C} y_j^{(i)} \log(y_{\text{pred},j}^{(i)})$$

7) Gradient Descent

Gradient Descent is a fundamental optimization algorithm used to update the parameters of neural networks by minimizing the loss function.

Batch Gradient Descent: In batch gradient descent, the gradients of the loss function with respect to all training examples are computed, and the parameters are updated in the opposite direction of the gradient.

$$\theta := \theta - \alpha \cdot \nabla J(\theta)$$

Where θ represents the parameters (weights and biases), α is the learning rate, and $\nabla J(\theta)$ is the gradient of the loss function.

Stochastic Gradient Descent (SGD): Stochastic Gradient Descent updates the parameters using the gradient of the loss computed for a single training example at each iteration.

$$\theta := \theta - \alpha \cdot \nabla J_{\text{sample}}(\theta)$$

Where $\nabla J_{\text{sample}}(\theta)$ is the gradient of the loss with respect to the current training example.

Mini-Batch Gradient Descent: Mini-batch gradient descent strikes a balance between batch and stochastic methods by updating the parameters using a small subset (mini-batch) of training examples.

8) Weight Initialization

Proper weight initialization is crucial for effective neural network training. Kaiming and Xavier initialization are techniques used to set initial weights, promoting faster convergence and preventing vanishing or exploding gradients.

Xavier Initialization (Glorot Initialization): Xavier initialization sets the initial weights of a layer to values drawn from a uniform or normal distribution with specific variances. It's designed to balance the scale of activations and gradients, preventing the network from becoming too slow or too fast during training.

For a layer with n_{in} input units and n_{out} output units, the weights are initialized as:

$$W \sim U\left(-\frac{\sqrt{6}}{\sqrt{n_{\text{in}}+n_{\text{out}}}}, \frac{\sqrt{6}}{\sqrt{n_{\text{in}}+n_{\text{out}}}}\right)$$

Kaiming Initialization (He Initialization): Kaiming initialization, also known as He initialization, is tailored for networks that use rectified linear units (ReLUs) as activation functions. It adapts the initialization to the properties of the ReLU activation, leading to improved training performance.

For a ReLU-activated layer with n_{in} input units, the weights are initialized as:

$$W \sim N(0, \sqrt{\frac{2}{n_{\text{in}}}})$$

9) Regularization Techniques

Regularization techniques are used in neural networks to prevent overfitting and improve the generalization performance of the model.

L1 Regularization (Lasso): L1 regularization adds the absolute values of the weights to the loss function, encouraging some of the weights to become exactly zero. This leads to sparsity in the model's parameters.

$$J(\theta) = \text{Loss} + \lambda \sum_{i=1}^{n} |w_i|$$

L2 Regularization (Ridge): L2 regularization adds the squared values of the weights to the loss function, which forces the weights to be small but not exactly zero. It helps in reducing the impact of large weights on the model.

$$J(\theta) = \text{Loss} + \lambda \sum_{i=1}^{n} w_i^2$$

Dropout: Dropout is a regularization technique that randomly sets a fraction of the neurons' activations to zero during each forward and backward pass. This prevents the network from relying heavily on any specific neuron and encourages robust learning of features.

D. MODEL TRAINING ALGORITHM

The process of training a neural network from scratch involves iteratively adjusting its parameters to learn from the provided training data. At each epoch, the algorithm performs forward propagation to predict outputs, computes the loss between predictions and actual targets, and then conducts backward propagation to calculate gradients for parameter updates. During backward propagation, the gradients are propagated layer by layer, helping to adjust weights and biases. These parameter updates are achieved through optimization methods like gradient descent. The process repeats for a specified number of epochs, gradually refining the network's parameters to improve its predictive accuracy.

Algorithm 1 Train Neural Network from Scratch

```
Require: Training data ($X_{\text{train}}, y_{\text{train}}$), Number of epochs num_epochs, Learning rate $\alpha$, Number of hidden layers num_hidden_layers, Number of neurons per hidden layer num_neurons, Activation function activation_function, Loss function loss_function
Ensure: Trained neural network parameters
  Initialize neural network parameters:
    Randomly initialize weights and biases for input, hidden, and output layers.
  for epoch = 1 to num_epochs do
    for each training example $(X, y)$ in $(X_{\text{train}}, y_{\text{train}})$ do
      Perform forward propagation:
        Set input layer values: $A^{[0]} = X$
      for $l = 1$ to num_hidden_layers + 1 do
          Compute $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$
          Compute $A^{[l]} = \text{activation\_function}(Z^{[l]})$
      end for
      Compute loss using loss_function($y, A^{[\text{num\_hidden\_layers}+1]}$)
      Perform backward propagation (Backpropagation):
        Compute $dZ^{[\text{num\_hidden\_layers}+1]} = A^{[\text{num\_hidden\_layers}+1]} - y$
      for $l$ = num_hidden_layers + 1 to 1 do
          Compute $dW^{[l]} = \frac{1}{m} dZ^{[l]} A^{[l-1]T}$
          Compute $db^{[l]} = \frac{1}{m}\text{np.sum}(dZ^{[l]}, \text{axis} = 1, \text{keepdims=True})$
          Compute $dZ^{[l-1]} = W^{[l]T} dZ^{[l]} \cdot \text{activation\_function\_derivative}(Z^{[l-1]})$
      end for
      Update parameters using Gradient Descent:
      for $l = 1$ to num_hidden_layers + 1 do
          Update $W^{[l]} = W^{[l]} - \alpha dW^{[l]}$
          Update $b^{[l]} = b^{[l]} - \alpha db^{[l]}$
      end for
    end for
  end for
  return Trained neural network parameters (weights and biases)
```

III. RESULTS

A. TRAINING ANN WITH RANDOM WEIGHT INITIALIZATION

The neural network was firstly trained by initializing the weights and biases randomly with the use of numpy functions. Random numbers can lead to model underperformance and high training time if the assigned weights and biases are very far off from the accurate weight and biases. The training dataset was trained for over 500 iterations, closing out at about 84.6% accuracy. The initial slope for the graph of accuracy plotted against the number of iterations shows a steep leap in the early training which eventually slowed down. The trained model was used to predict the numbers from the MNIST dataset. The confusion matrix for the thus mentioned inference on the dataset yielded a decent performance with high scores consistently. This, doesn't however come without flaws as there are still a number of data points, being incorrectly labeled but those data instances are very low in number. The testing accuracy was barely able to edge out 85%. The vanilla approach to training the model used ReLU activation function only.

B. TRAINING ANN WITH XAVIER INITIALIZATION

The same model was trained over the same dataset for equal iterations (500) but with some different configurations. Firstly, as opposed to the ReLU activation function only as in the vanilla training approach, the introduction of other non-linear functions, namely tanh and sigmoid was done. The weights were no more randomly initialized but were taken form a Xavier Distribution. Training the model with the different configurations brought about different results. The accuracy graph plotted against the number of iterations had almost no resemblance to the plot obtained from random initialization. The maximum achieved accuracy from the curve was only about 22% with very bad pattern. The curve seems to be rising and dropping at times and thereby doesn't represent the case of gradient descent. With that, the confusion matrix for the predicitons with the model was atrocious, leading to a familiar 22% accuracy. The data instances were labeled into either of the first three class groups with absolute nil afterwards. There were a lot of true positives with the number 1 but there were also a lot more numbers being classified as 1 that are not actually 1.

C. TRAINING ANN WITH KAIMING INITIALIZATION

The model underwent training using varying configurations across 500 iterations, diverging from the conventional ReLU activation function. Instead, it explored other nonlinear functions such as tanh and sigmoid. Notably, the weight initialization method shifted from random values to Xavier Distribution. This alternative approach yielded distinct outcomes. Plotting the accuracy graph against the iteration count revealed a dissimilarity to the random initialization plot. Although the curve displayed fluctuations, it displayed an upward trend followed by fluctuations, which deviates from the characteristic behavior of gradient descent. Despite these modifications, the maximum attained accuracy plateaued at around 24%, showcasing an unfavorable pattern. The confusion matrix for the predictions illustrated concerning results. The majority of data instances were classified into the initial three class groups, and the subsequent classifications dwindled significantly. While the number 1 class exhibited numerous true positives, the model also classified a substantial number of non-"1" instances as "1". Even with these changes in configuration, the predictions saw an improvement in accuracy, reaching 76%, yet still falling short of the random initialization results. The testing accuracy however came to around 78%.

D. TRAINING ANN WITH REGULARIZATION

For the forth configuration, regularization was performed on the model to evaluate the performance. The dropout regularization used on the model randomly sets some of the weights on the model to be 0, basically disabling the neurons. The other configurations were similar with only ReLU activation function being used and a learning rate of 0.1. The training accuracy from the model was obtained to be 78.5% with a very normal looking curve that starts to flatten at around 500 iterations. The predictions from the model are pretty decent with an accuracy of 83.9% over the test dataset. The confusion matrix has a regular appearance with most of the values lying in the principal diagonal and very few off-diagonal elements. Out of the many samples, there are less than 5 instances of double digit off-diagonals in the confusion matrix.

E. TRAINING ANN WITH MULTIPLE HIDDEN LAYERS

The final training setup for the ANN model consisted of multiple hidden layers that the data points must pass through before a prediction is made. Precisely, 3 hidden layers with RelU, tanh and sigmoid activation functions were used in succession. The training of the model for 500 iterations with 0.1 learning rate yielded a sub-par 67.3% accuracy over the training dataset. There was a large number of off-diagonal elements in the confusion matrix with the accuracy barely hitting the 67% mark. While there are a lot of forgivable instances of misclassification, like a 2 being classified as 7, there are still a number of images that have no resemblance to the class they are being classified as. Overall, the use of the hidden layers degrades the model performance more than when there is only a single one.

IV. DISCUSSION AND ANALYSIS

The initial training of the neural network involved random initialization of weights and biases using numpy functions. This approach can lead to underperformance and longer training times when the assigned weights and biases deviate significantly from accurate values. After 500 iterations, the training dataset achieved an accuracy of approximately 84.6%. The accuracy graph exhibited an initial steep increase followed by a slowdown, indicative of learning convergence. This trained model was then employed for number prediction from the MNIST dataset. The resulting confusion matrix showed a commendable performance, albeit with some misclassifications, particularly in a limited number of data points. The testing accuracy narrowly surpassed 85%, validating the vanilla training approach with ReLU activation.

In contrast, the model's second configuration introduced tanh and sigmoid non-linear activation functions alongside Xavier Distribution weight initialization. However, this approach resulted in unexpected outcomes. The accuracy graph demonstrated an erratic pattern, not resembling the typical gradient descent behavior. The maximum accuracy achieved was around 22%, much lower than anticipated. The confusion matrix corroborated this poor performance, with most instances being wrongly labeled as the first three class groups.

Subsequently, a configuration change incorporated the non-ReLU activation functions with Xavier Initialization. The accuracy graph again diverged from the anticipated gradient descent pattern, peaking at approximately 24%. The confusion matrix further underscored the inadequacies, displaying a significant number of misclassifications. While the accuracy improved to 76%, it remained lower than the initial random initialization.

In the fourth configuration, regularization was introduced using dropout. However, even with ReLU activation and a learning rate of 0.1, the training accuracy plateaued at 78.5% with a flattening curve. Despite the regularization, the testing accuracy reached 83.9%, showcasing improved results. The confusion matrix exhibited a more favorable pattern, with most values on the principal diagonal and minimal off-diagonal elements.

In summary, the model's performance deviations can be attributed to the introduction of different activation functions and weight initialization methods. While some configurations yielded better results, they still fell short of expectations. The abnormalities suggest that careful tuning and parameter adjustments are necessary for achieving desirable accuracy levels.

V. CONCLUSION

In conclusion, our exploration of various configurations and techniques for training a neural network using the MNIST dataset has provided valuable insights into the complex interplay of activation functions, weight initialization, and regularization. Through our experiments, we witnessed the significant impact that these factors can have on the model's performance. The initial training, employing ReLU activation and random weight initialization, demonstrated a commendable accuracy of around 84.6%. This result underscored the effectiveness of the vanilla approach in capturing meaningful patterns within the dataset.

However, as we ventured into alternative configurations, we observed that deviations from the norm could lead to unexpected outcomes. Introduction of tanh and sigmoid activations along with Xavier Initialization showed notably lower accuracies, highlighting the sensitivity of the model's behavior to these changes. The same trend persisted with the inclusion of dropout regularization, showcasing the need for careful selection and combination of techniques.

While our models achieved varying degrees of success, the achieved results emphasized the intricate nature of neural network training. Despite the challenges encountered, the models were still able to provide predictive capabilities, especially in the regularization-integrated approach where testing accuracy reached 83.9%. This demonstrates the potential of neural networks in handling complex classification tasks, even if the results did not match initial expectations.

In essence, our study underscores the necessity of a systematic and iterative approach when building and fine-tuning neural network models. The journey through these configurations has offered a deeper appreciation for the delicate balance between architectural choices and their impact on model performance. These findings provide a strong foundation for further exploration and refinement, contributing to our broader understanding of how neural networks can be effectively leveraged for various applications.

VI. REFERENCES

- David Bowser-Chao and Debra L. Dzialo. "Comparison of the use of nearest neighbours and neural networks in top-quark detection." *Physical Review D*, vol. 47, no. 5, pp. 1900–1905, Mar. 1993. doi: 10.1103/physrevd.47.1900.

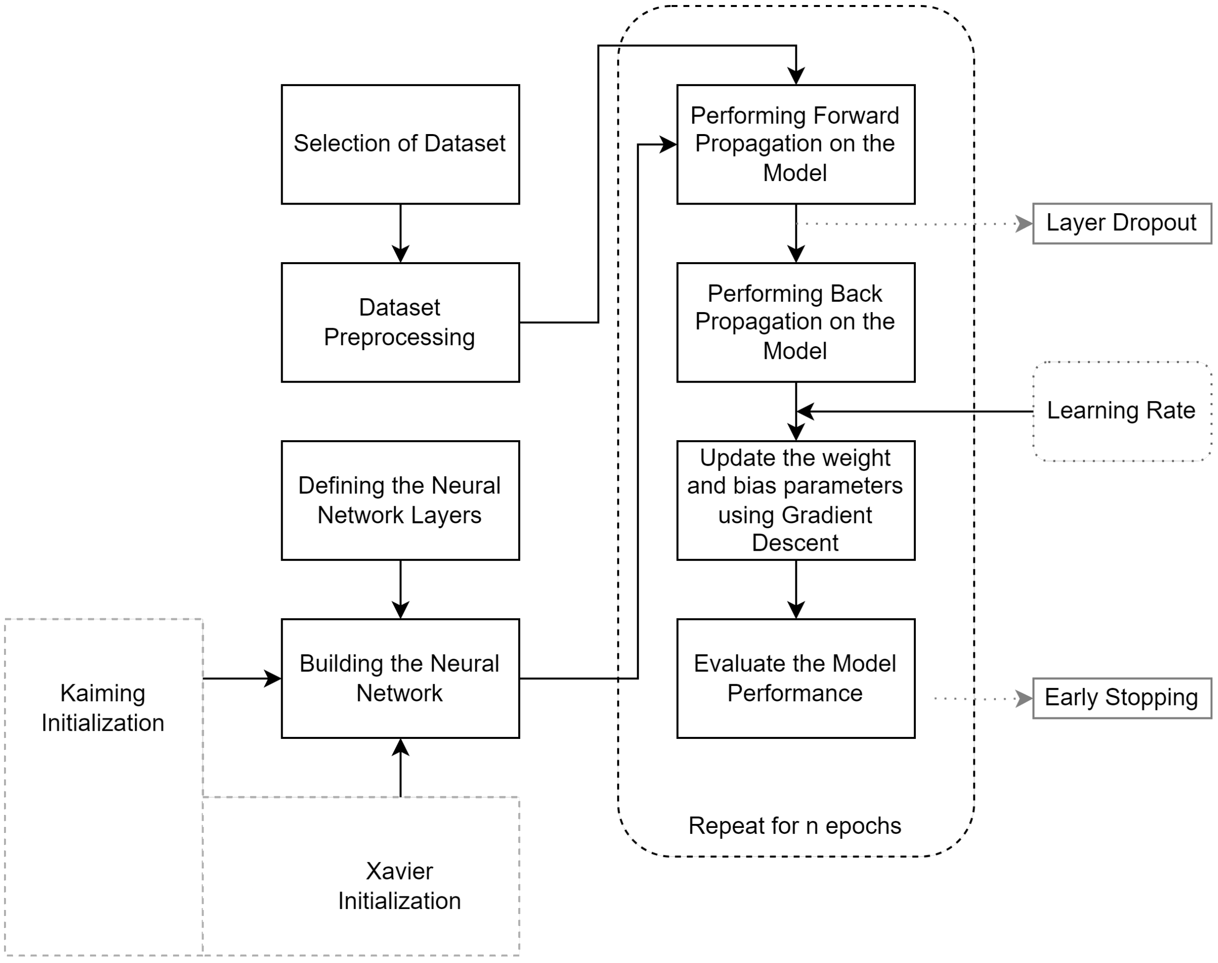
PRATIGYA PAUDEL is a fourth year student, studying computer engineering under IOE, Thapathali Campus. She has been involved in a lot of machine learning projects and has a keen eye for data analysis and AI related stuff. With the enthusiasm for Artificial Intelligence (AI), she is driven by the potential of AI to transform industries and tackle complex challenges. Her academic journey has equipped her with a strong foundation in AI concepts, including machine learning and data analysis. She possesses a relentless curiosity and is always eager to explore the latest advancements in AI. Her goal is to apply her knowledge and make a meaningful contribution in the field.

SUSHANK GHIMIRE is a fourth year student, studying computer engineering under IOE, Thapathali Campus. He possesses a lot of interest, working with data. His educational path has provided him with a solid understanding of AI concepts, encompassing machine learning and data analysis. He possesses an unwavering curiosity and is constantly eager to delve into the latest advancements in AI. His objective is to leverage his knowledge and expertise to create a significant impact in the field.

APPENDIX

A. FIGURES AND PLOTS

FIGURE 1. System Block Diagram


Selection of Dataset
Dataset Preprocessing
Defining the Neural Network Layers
Building the Neural Network
Kaiming Initialization
Xavier Initialization
Performing Forward Propagation on the Model
Performing Back Propagation on the Model
Update the weight and bias parameters using Gradient Descent
Evaluate the Model Performance
Repeat for n epochs
Layer Dropout
Learning Rate
Early Stopping

1) Activation Function Curves

FIGURE 2. ReLU Graph

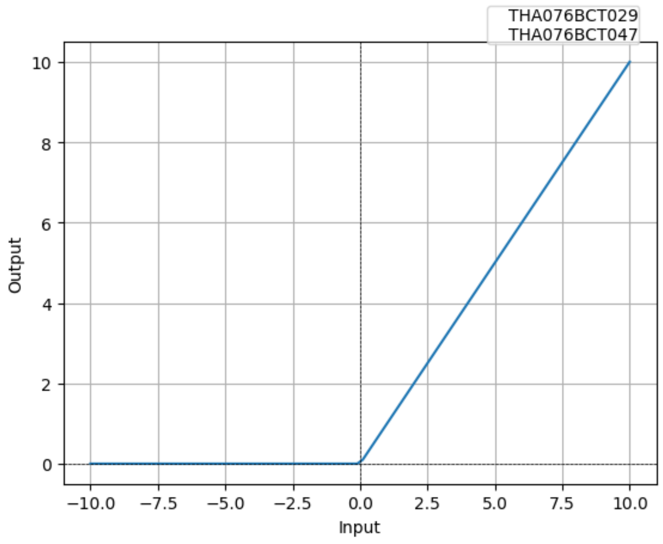


FIGURE 3. Tanh Graph

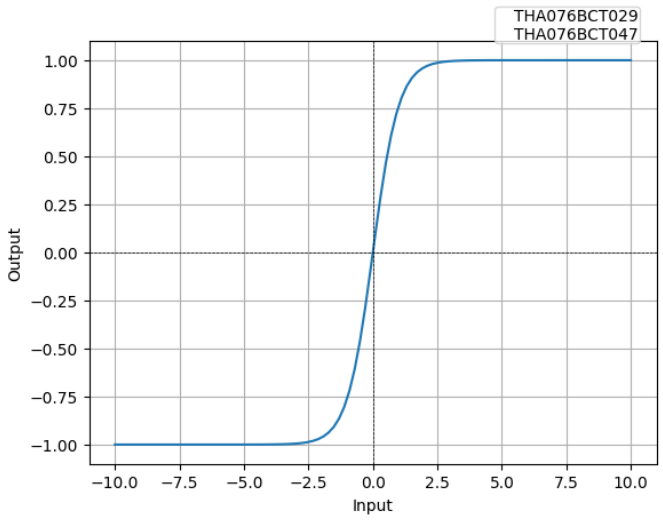


FIGURE 4. Sigmoid Graph

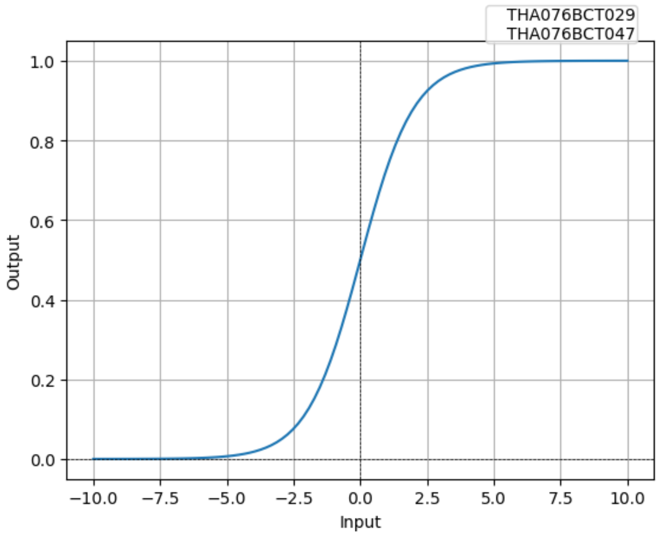
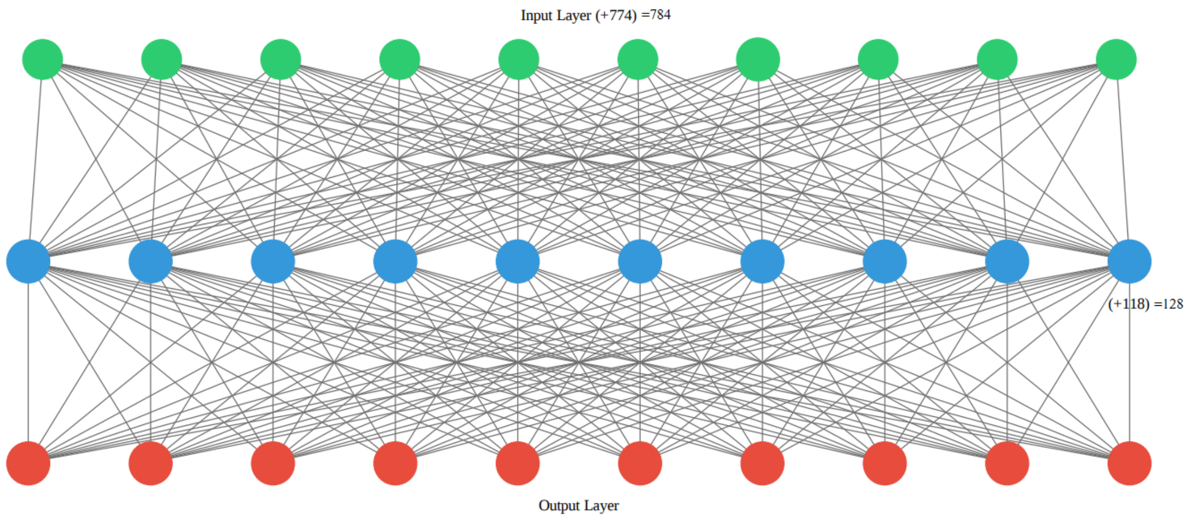


FIGURE 5. Model Structure



2) Vanilla Training

FIGURE 6. Accuracy Plot for Vanilla ANN Training

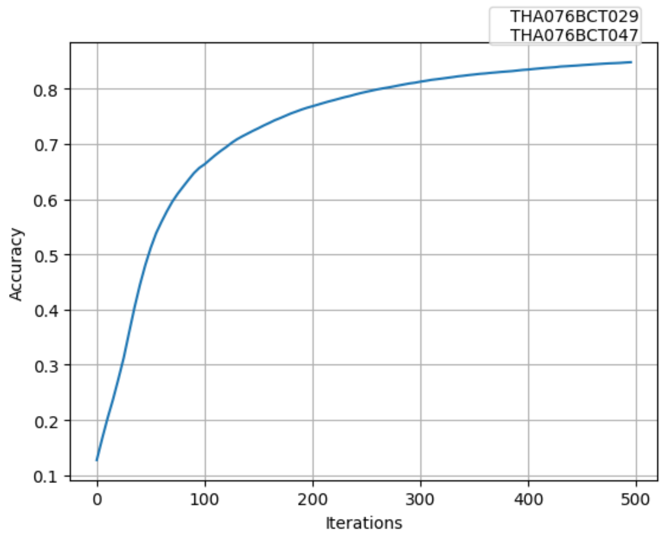
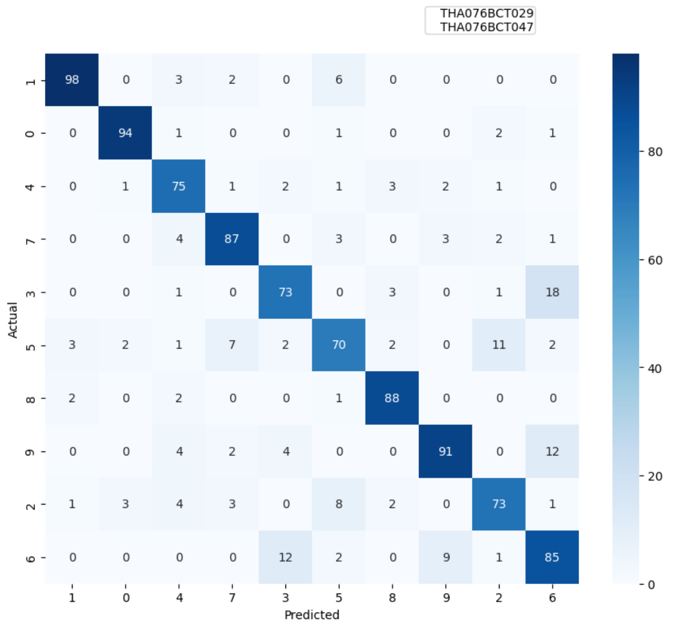


FIGURE 7. Confusion Matrix for Vanilla ANN Training



Actual \ Predicted	1	0	4	7	3	5	8	9	2	6
1	98	0	3	2	0	6	0	0	0	0
0	0	94	1	0	0	1	0	0	2	1
4	0	1	75	1	2	1	3	2	1	0
7	0	0	4	87	0	3	0	3	2	1
3	0	0	1	0	73	0	3	0	1	18
5	3	2	1	7	2	70	2	0	11	2
8	2	0	2	0	0	1	88	0	0	0
9	0	0	4	2	4	0	0	91	0	12
2	1	3	4	3	0	8	2	0	73	1
6	0	0	0	0	12	2	0	9	1	85

3) Xavier Initialization

FIGURE 8. Accuracy Plot for ANN Training with Xavier Initialization

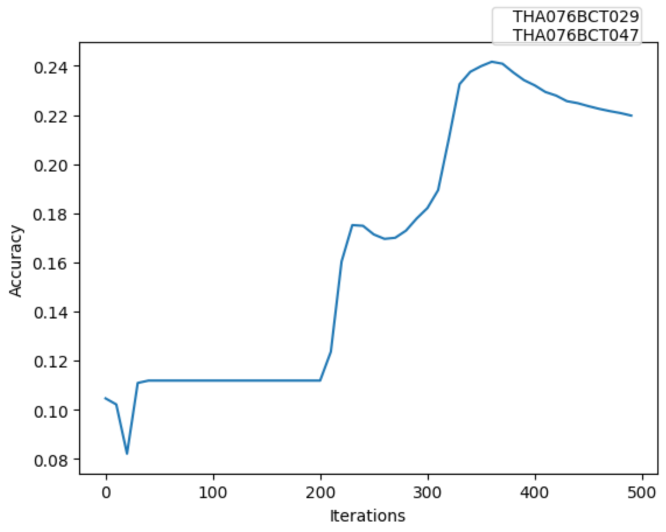
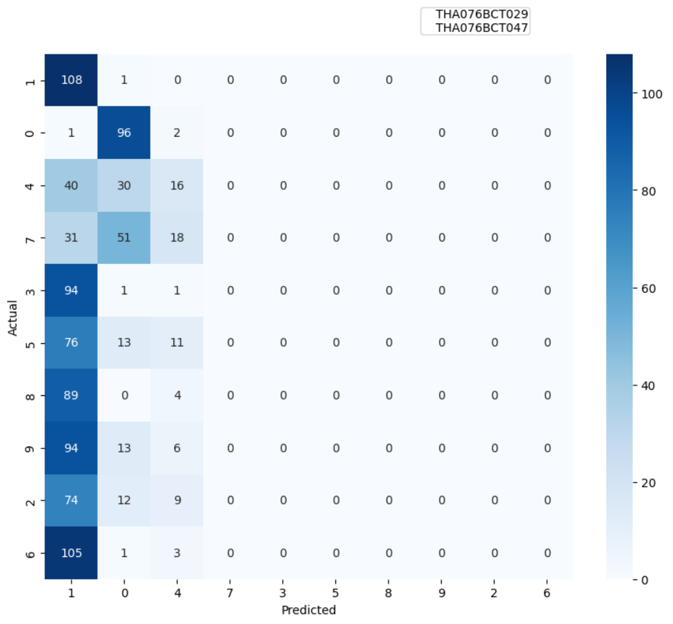


FIGURE 9. Confusion Matrix for ANN Training with Xavier Initialization



Actual \ Predicted	1	0	4	7	3	5	8	9	2	6
1	108	1	0	0	0	0	0	0	0	0
0	1	96	2	0	0	0	0	0	0	0
4	40	30	16	0	0	0	0	0	0	0
7	31	51	18	0	0	0	0	0	0	0
3	94	1	1	0	0	0	0	0	0	0
5	76	13	11	0	0	0	0	0	0	0
8	89	0	4	0	0	0	0	0	0	0
9	94	13	6	0	0	0	0	0	0	0
2	74	12	9	0	0	0	0	0	0	0
6	105	1	3	0	0	0	0	0	0	0

4) Kaiming Initialization

FIGURE 10. Accuracy Plot for ANN Training with Kaiming Initialization

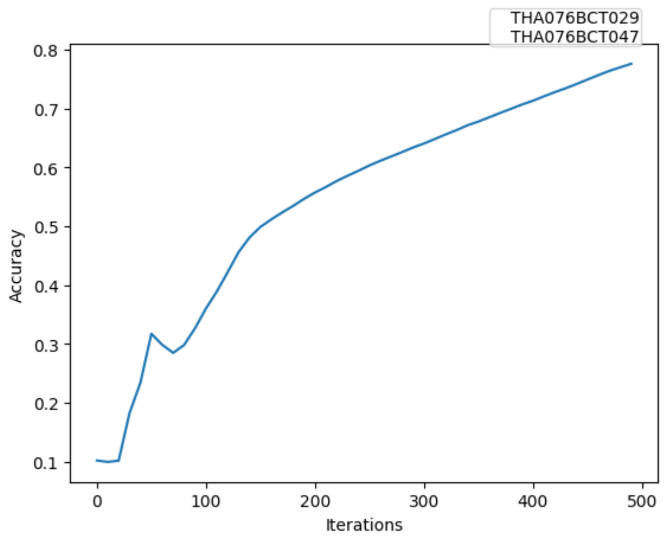
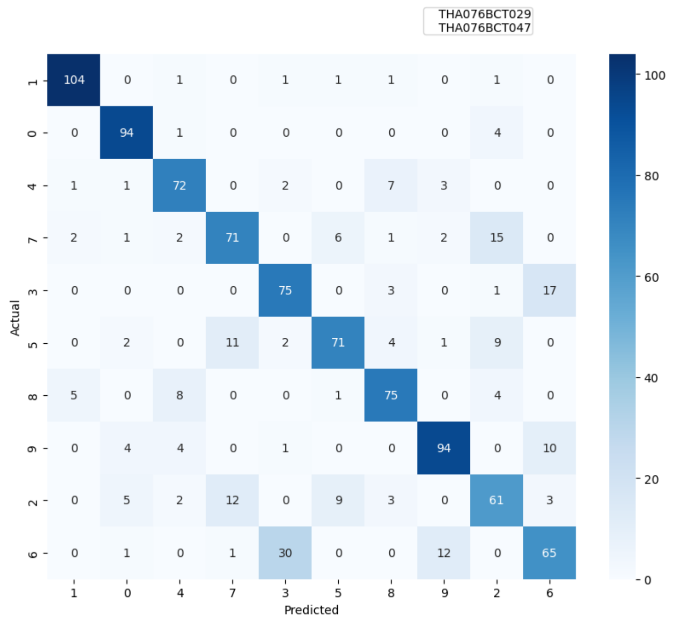


FIGURE 11. Confusion Matrix for ANN Training with Kaiming Initialization



	1	0	4	7	3	5	8	9	2	6
1	104	0	1	0	1	1	1	0	1	0
0	0	94	1	0	0	0	0	0	4	0
4	1	1	72	0	2	0	7	3	0	0
7	2	1	2	71	0	6	1	2	15	0
3	0	0	0	0	75	0	3	0	1	17
5	0	2	0	11	2	71	4	1	9	0
8	5	0	8	0	0	1	75	0	4	0
9	0	4	4	0	1	0	0	94	0	10
2	0	5	2	12	0	9	3	0	61	3
6	0	1	0	1	30	0	0	12	0	65

5) Training with Dropout

FIGURE 12. Accuracy Plot for ANN Training with Regularization

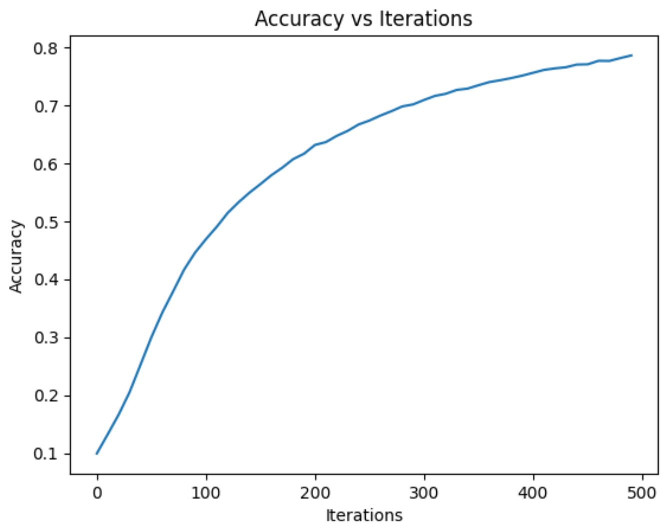
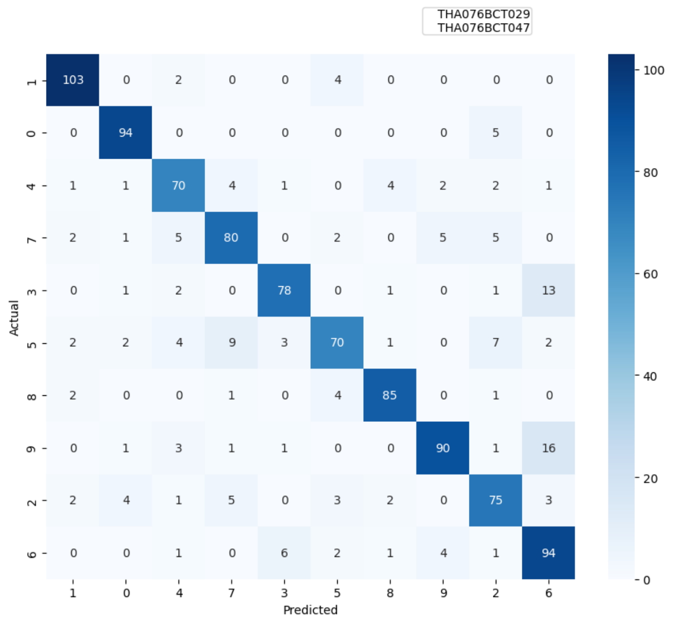


FIGURE 13. Confusion Matrix for ANN Training with Regularization



6) Training with Extra Hidden Layers

FIGURE 14. Accuracy Plot for ANN Training with Multiple Hidden Layers

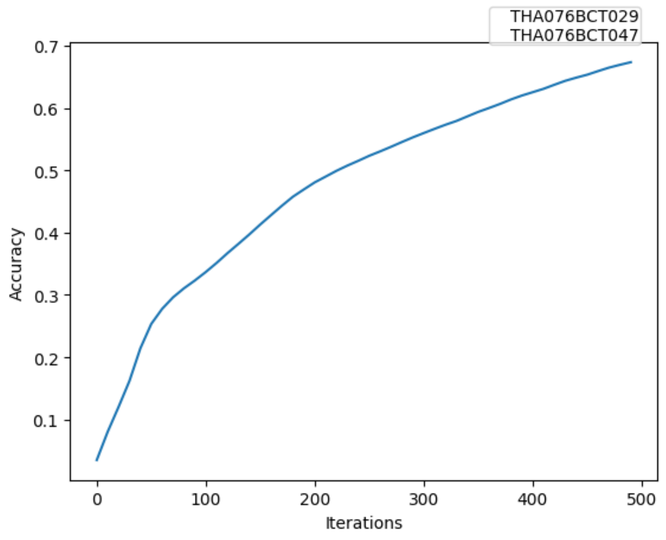
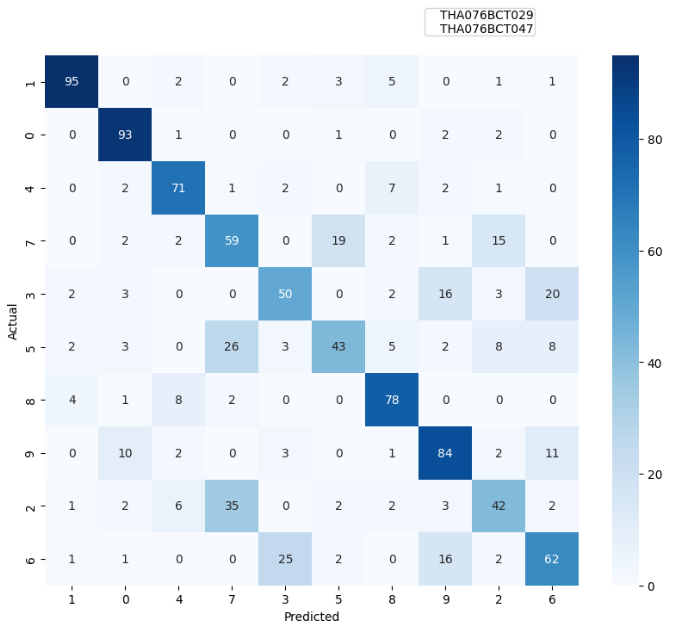


FIGURE 15. Confusion Matrix for ANN Training with Multiple Hidden Layers



B. CODING

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix ,classification_report
import random
import seaborn as sns

#Dataset Preprocessing
data1= pd.read_csv('digit.csv')
data1.head(5)

data = np.array(data1)
x,y = data.shape
print(x,y)

np.random.shuffle(data) # shuffle before splitting
data2 = data[0:1000].T
Y1 = data2[0]
X1 = data2[1:y]
X1 = X2 / 255

data_train = data[1000:x].T
Y_train = data_train[0]
X_train = data_train[1:y]
X_train = X_train / 255.
_,x_train = X_train.shape

#Visualizing dataset
rand_index = np.random.randint(0, 41000)
rand_index

image = X_train.T[rand_index]
img = np.reshape(image, (28,28))
plt.imshow(img)

def init_params():
    #Initialize weights and biases for the hidden layer
    W1 = np.random.rand(10, 784) - 0.5
    b1 = np.random.rand(10, 1) - 0.5

    # Initialize weights and biases for the output layer
    W2 = np.random.rand(10, 10) - 0.5
    b2 = np.random.rand(10, 1) - 0.5
    return W1, b1, W2, b2

#Activation Functions

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

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

```

def ReLU_deriv(Z):
    return Z > 0

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

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

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

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

#visualization of various activation functions
#ReLu
x = np.linspace(-10, 10, 100)   # Generate 100 points from -10 to 10
y = ReLU(x)

plt.plot(x, y, label='ReLU')
plt.axhline(0, color='black', linewidth=0.5, linestyle='--')  # Add x-axis
plt.axvline(0, color='black', linewidth=0.5, linestyle='--')  # Add y-axis
plt.xlabel('Input')
plt.ylabel('Output')
plt.title('ReLU Activation Function')
plt.legend()
plt.grid()
plt.show()

#Tanh
x = np.linspace(-10, 10, 100)   # Generate 100 points from -10 to 10
y = tanh(x)

plt.plot(x, y, label='ReLU')
plt.axhline(0, color='black', linewidth=0.5, linestyle='--')  # Add x-axis
plt.axvline(0, color='black', linewidth=0.5, linestyle='--')  # Add y-axis
plt.xlabel('Input')
plt.ylabel('Output')
legend_handles = [
    plt.Line2D([], [], color='black', marker='o', markersize=10, label='THA076BCT029\nTHA0
]
plt.legend(handles=legend_handles, loc='upper left', bbox_to_anchor=(0.7, 1.1), ncol=len(le
plt.grid()
plt.show()

#Sigmoid
x = np.linspace(-10, 10, 100)   # Generate 100 points from -10 to 10
y = sigmoid(x)

plt.plot(x, y, label='ReLU')
plt.axhline(0, color='black', linewidth=0.5, linestyle='--')  # Add x-axis
plt.axvline(0, color='black', linewidth=0.5, linestyle='--')  # Add y-axis
plt.xlabel('Input')
plt.ylabel('Output')
```

```
legend_handles = [
    plt.Line2D([], [], color='black', marker='o', markersize=10, label='THA076BCT029\nTHA0
]
plt.legend(handles=legend_handles, loc='upper left', bbox_to_anchor=(0.7, 1.1), ncol=len(le
plt.grid()
plt.show()

def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = softmax(Z2)
    return Z1, A1, Z2, A2

def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
    return one_hot_Y

def backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y):
    one_hot_Y = one_hot(Y)
    dZ2 = A2 - one_hot_Y
    dW2 = 1 / x * dZ2.dot(A1.T)
    db2 = 1 / x * np.sum(dZ2)
    dZ1 = W2.T.dot(dZ2) * ReLU_deriv(Z1)
    dW1 = 1 / x * dZ1.dot(X.T)
    db1 = 1 / x * np.sum(dZ1)
    return dW1, db1, dW2, db2

def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2

def get_predictions(A2):
    return np.argmax(A2, 0)

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

# accuracy_scores = [] # To store accuracy scores at different iterations
def gradient_descent(X, Y, alpha, iterations):
    W1, b1, W2, b2 = init_params()
    accuracy_scores = []   # To store accuracy scores at different iterations

    for i in range(iterations):
        Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X)
        dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W1, W2, X, Y)
        W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)
        if i % 5 == 0:
            print("Iteration: ", i)
            predictions = get_predictions(A2)
            accuracy = get_accuracy(predictions, Y)
            print(accuracy)
            accuracy_scores.append((i, accuracy))
        # Convert accuracy_scores to separate lists for plotting
    iterations, accuracies = zip(*accuracy_scores)

    # Plotting accuracy vs iterations
    plt.plot(iterations, accuracies)
    plt.grid()
    plt.xlabel('Iterations')
    plt.ylabel('Accuracy')
    plt.title('Accuracy vs Iterations')
    plt.show()
    return W1, b1, W2, b2

W1, b1, W2, b2 = gradient_descent(X_train, Y_train, 0.15, 300)

def make_predictions(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_predictions(A2)
    return predictions

def test_prediction(index, W1, b1, W2, b2):
    current_image = X_train[:, index, None]
    prediction = make_predictions(X_train[:, index, None], W1, b1, W2, b2)
    label = Y_train[index]
    print("Prediction: ", prediction)
    print("Label: ", label)

    current_image = current_image.reshape((28, 28)) * 255
    plt.gray()
    plt.imshow(current_image, interpolation='nearest')
    plt.show()

predictions = make_predictions(X1, W1, b1, W2, b2)
get_accuracy(predictions, Y1)

classes = df['label'].unique()
classes

#Confusion matrix
conf_matrix = confusion_matrix(Y1, predictions)

# Plotting confusion matrix as heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues', xticklabels=classes, yticklabe
plt.xlabel('Predicted')
plt.ylabel('Actual')
legend_handles = [
    plt.Line2D([], [], color='black', marker='o', markersize=10, label='THA076BCT029\nTHA0
]
plt.legend(handles=legend_handles, loc='upper left', bbox_to_anchor=(0.7, 1.1), ncol=len(le
plt.show()

#Kaiming Initialization
```

```
def init_params_kaiming():
    def kaiming_initializer(n_in):
        return np.random.normal(0, np.sqrt(2 / n_in))

    W1 = kaiming_initializer(784) * np.random.randn(10, 784)
    b1 = np.zeros((10, 1))

    W2 = kaiming_initializer(10) * np.random.randn(15, 10)
    b2 = np.zeros((15, 1))

    W3 = kaiming_initializer(15) * np.random.randn(12, 15)
    b3 = np.zeros((12, 1))

    W4 = kaiming_initializer(12) * np.random.randn(10, 12)
    b4 = np.zeros((10, 1))

    return W1, b1, W2, b2, W3, b3, W4, b4

#Xavier Initialization

def init_params_xavier():
    def xavier_initializer(n_in, n_out):
        bound = np.sqrt(6 / (n_in + n_out))
        return np.random.uniform(-bound, bound)

    W1 = xavier_initializer(784, 10) * np.random.rand(10, 784)
    b1 = np.random.rand(10, 1) - 0.5

    W2 = xavier_initializer(10, 15) * np.random.rand(15, 10)
    b2 = np.random.rand(15, 1) - 0.5

    W3 = xavier_initializer(15, 12) * np.random.rand(12, 15)
    b3 = np.random.rand(12, 1) - 0.5

    W4 = xavier_initializer(12, 10) * np.random.rand(10, 12)
    b4 = np.random.rand(10, 1) - 0.5

    return W1, b1, W2, b2, W3, b3, W4, b4

def forward_prop(W1, b1, W2, b2,W3,b3,W4,b4, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)

    Z2 = W2.dot(A1) + b2
    A2 = sigmoid(Z2)

    Z3 = W3.dot(A2) + b3
    A3 = tanh(Z3)

    Z4 = W4.dot(A3) + b4
    A4 = softmax(Z4)
    return Z1, A1, Z2, A2, Z3,A3,Z4,A4

def one_hot(Y):
    one_hot_Y = np.zeros((Y.size, Y.max() + 1))
    one_hot_Y[np.arange(Y.size), Y] = 1
    one_hot_Y = one_hot_Y.T
    return one_hot_Y

def backward_prop(Z1, A1, Z2, A2, Z3, A3, Z4,A4, W1, W2,W3,W4, X, Y):
    one_hot_Y = one_hot(Y)

    dZ4 = A4 - one_hot_Y
    dW4 = 1 / m * dZ4.dot(A3.T)
    db4 = 1 / m * np.sum(dZ4)

    dZ3 = W4.T.dot(dZ4) * tanh_derivative(Z3)
    dW3 = 1 / m * dZ3.dot(A2.T)
    db3 = 1 / m * np.sum(dZ3)

    dZ2 = W3.T.dot(dZ3) * sigmoid_derivative(Z2)
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)

    dZ1 = W2.T.dot(dZ2) * ReLU_deriv(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)
    return dW1, db1, dW2, db2, dW3, db3, dW4,db4

def update_params(W1, b1, W2, b2,W3,b3,W4,b4, dW1, db1, dW2, db2,dW3,db3,dW4,db4, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1

    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2

    W3 = W3 - alpha * dW3
    b3 = b3 - alpha * db3

    W4 = W4 - alpha * dW4
    b4 = b4 - alpha * db4

    return W1, b1, W2, b2, W3, b3, W4, b4

def get_predictions(A2):
    return np.argmax(A2, 0)

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

#Regularization
def dropout_forward(X, keep_prob):
    D = np.random.rand(*X.shape) < keep_prob
    A = X * D / keep_prob
    return A, D

def visualize_dropout(original_data, dropout_data, dropout_mask):
    fig, axs = plt.subplots(1, 3, figsize=(12, 4))

    axs[0].imshow(original_data.reshape(28, 28), cmap='gray')
    axs[0].set_title("Original Data")

    axs[1].imshow(dropout_mask.reshape(28, 28), cmap='gray')
    axs[1].set_title("Dropout Mask")

    axs[2].imshow(dropout_data.reshape(28, 28), cmap='gray')
    axs[2].set_title("Data with Dropout")

    for ax in axs:
        ax.axis('off')

    plt.tight_layout()
    plt.show()

#Multiple Hidden layers
def forward_prop(W1, b1, W2, b2,W3,b3,W4,b4, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLU(Z1)

    Z2 = W2.dot(A1) + b2
    A2 = sigmoid(Z2)

    Z3 = W3.dot(A2) + b3
    A3 = tanh(Z3)

    Z4 = W4.dot(A3) + b4
    A4 = softmax(Z4)
    return Z1, A1, Z2, A2, Z3,A3,Z4,A4

def backward_prop(Z1, A1, Z2, A2, Z3, A3, Z4,A4, W1, W2,W3,W4, X, Y):
    one_hot_Y = one_hot(Y)

    dZ4 = A4 - one_hot_Y
    dW4 = 1 / m * dZ4.dot(A3.T)
    db4 = 1 / m * np.sum(dZ4)

    dZ3 = W4.T.dot(dZ4) * tanh_derivative(Z3)
    dW3 = 1 / m * dZ3.dot(A2.T)
    db3 = 1 / m * np.sum(dZ3)

    dZ2 = W3.T.dot(dZ3) * sigmoid_derivative(Z2)
    dW2 = 1 / m * dZ2.dot(A1.T)
    db2 = 1 / m * np.sum(dZ2)

    dZ1 = W2.T.dot(dZ2) * ReLU_deriv(Z1)
    dW1 = 1 / m * dZ1.dot(X.T)
    db1 = 1 / m * np.sum(dZ1)

    return dW1, db1, dW2, db2, dW3, db3, dW4,db4


def update_params(W1, b1, W2, b2,W3,b3,W4,b4, dW1, db1, dW2, db2,dW3,db3,dW4,db4, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1

    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2

    W3 = W3 - alpha * dW3
    b3 = b3 - alpha * db3

    W4 = W4 - alpha * dW4
    b4 = b4 - alpha * db4

    return W1, b1, W2, b2, W3, b3, W4, b4
```

C. BACK PROPAGATION

APPENDIX. BACKPROPAGATION DERIVATION

Consider a simple neural network with one hidden layer. Let's denote the input as x, the hidden layer weights as W_h, the hidden layer bias as b_h, the hidden layer activation function as g, the output layer weights as W_o, and the output layer bias as b_o.

The forward propagation equations are:

$$Z_h = W_h \cdot x + b_h$$

$$A_h = g(Z_h)$$

$$Z_o = W_o \cdot A_h + b_o$$

$$A_o = g(Z_o)$$

Assume we have a loss function L that measures the error between the predicted output A_o and the target output y. We want to update the weights and biases to minimize the loss function using gradient descent.

The backpropagation algorithm computes the gradients of the loss with respect to the network's parameters.

The gradient of the loss with respect to the output layer activations:

$$\delta_o = \frac{\partial L}{\partial A_o} \odot g'(Z_o)$$

The gradients of the loss with respect to the output layer weights and bias:

$$\frac{\partial L}{\partial W_o} = \delta_o \cdot A_h^T$$

$$\frac{\partial L}{\partial b_o} = \delta_o$$

The gradient of the loss with respect to the hidden layer activations:

$$\delta_h = (W_o^T \cdot \delta_o) \odot g'(Z_h)$$

The gradients of the loss with respect to the hidden layer weights and bias:

$$\frac{\partial L}{\partial W_h} = \delta_h \cdot x^T$$

$$\frac{\partial L}{\partial b_h} = \delta_h$$

Finally, the weights and biases are updated using the gradients and the learning rate α:

$$W_o = W_o - \alpha \cdot \frac{\partial L}{\partial W_o}$$

$$b_o = b_o - \alpha \cdot \frac{\partial L}{\partial b_o}$$

$$W_h = W_h - \alpha \cdot \frac{\partial L}{\partial W_h}$$

$$b_h = b_h - \alpha \cdot \frac{\partial L}{\partial b_h}$$

This process of computing gradients and updating weights is repeated iteratively until convergence.

• • •