IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication August, 27

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Building a Neural Network from Scratch for MNIST Classification

RASHMI KHADKA[1] (THA076BCT035), SHIWANI SHAH[2] (THA076BCT042)
[1]Department of Electronics and Computer Engineering, Thapathali Campus (e-mail: rashmikhadka6255@gmail.com)
[2]Department of Electronics and Computer Engineering, Thapathali Campus (e-mail: shahshiwani70@gmail.com)

ABSTRACT

One of the major problems in the field of artificial intelligence (AI) is the use of machine learning model as a black box, even though it might be helpful in a few cases but understanding the internal structure and the operating mechanism will assist the user to tweak the variables in a more efficient and productive manner. This paper shows the working of an artificial neural network (ANN) from scratch. The entire mathematics behind the working of neural network along with the different evaluation metrics required to assess the performance of the model are discussed in this paper. The research encompasses the development of a custom neural network from scratch, applied to a digit dataset. Moreover, it extends to constructing neural networks with multiple hidden layers and dropout layers, while also illuminating the impact of Kaiming/Xavier initialization versus random initialization. This endeavor enhances comprehension of ANN mechanics, enabling more informed architectural decisions and heightened optimization potential.

INDEX TERMS Artificial Neural Network, ANN, Digit Dataset, MNIST

I. INTRODUCTION

ARTIFICIAL Neural Networks (ANNs), inspired by the human brain's neural structure, are adept at capturing complex patterns and relationships within data. By interconnecting layers of neurons and employing activation functions, ANNs process input information to yield meaningful output. Artificial Neural Networks (ANNs) have been a subject of extensive exploration, with a goal to achieve human-like performance over the years. Visualize an ANN as a microcosm within the cerebral cortex, mirroring certain aspects of neurons through simplified mathematical models. These models emulate dendrites, cell bodies, and axons, effectively portraying the essence of biological neurons. Notably, the input nodes in an artificial neuron are analogous to dendrites in a biological neuron, while signal strength and transfer in biological neurons echo a step function-like threshold in artificial ones.Understanding the mechanics of an ANN can be likened to mapping functions between spaces, linearly or via higher-order relations, thereby addressing classification and regression challenges.

At its core, an ANN comprises three layers: input, hidden, and output. The Perceptron, a basic ANN structure, starts this journey with input and output layers, representing one-to-one mappings. Yet, the Perceptron's limitations soon surfaced, as it struggled with non-linear classification and continuous functional approximations. This was surmounted by stacking layers or utilizing fully connected perceptron layers, paving the way for more complex architectures.This paper aims to offer a distilled comprehension of ANN functionality for classification and regression tasks. While training neural networks has become routine, the intricacies of associated mathematics often elude understanding. This study bridges that gap, unfolding the mathematical framework underlying neural networks, followed by hands-on implementation.

Furthermore, the report outlines the objectives of this laboratory endeavor. The primary aim is to demystify the intricate mechanics of ANNs through hands-on implementation. Through this journey, we will construct a custom neural network and deploy it on a digit dataset. This process entails designing and training the network, followed by an assessment of its predictive capabilities. Additionally, the laboratory encompasses the creation of ANNs with multiple hidden layers and the integration of dropout layers, a regularization technique. Lastly, we delve into the comparative analysis of initialization techniques, exploring the effects of Kaiming/Xavier initialization versus random initialization.

II. METHODOLOGY

A. BUILDING ARTIFICIAL NEURAL NETWORK FROM SCRATCH

Artificial Neural Networks (ANNs) are computational models inspired by the structure and functioning of the human brain. They are designed to mimic the way biological neurons work, allowing machines to learn and make decisions based on data. ANNs have gained immense popularity in the field of artificial intelligence due to their ability to solve a wide range of complex problems, from image recognition to language processing.

In general, a neural network contains three layers :

- **Input layer:** It takes the input vectors from the user and multiplies the respective branch weights to it.
- **Hidden layer:** Hidden layer is the set of neurons where all the computations are performed on the input data. It is responsible for learning more complex patterns from the data.
- **Output layer:** It gives the final predicted output value based on the input features.

This section outlines the step-by-step methodology employed to construct an Artificial Neural Network (ANN) from scratch, delving into the intricacies of its implementation along with the relevant equations.

- **Data Preprocessing:**
  The process starts with data preprocessing, involving loading the digit dataset and partitioning it into training and testing sets. The images are normalized by scaling their pixel values to a range between 0 and 1, enhancing convergence during training.
- **Initializing Parameters:** Parameters, including weights (W) and biases (b), are randomly initialized to facilitate the learning process. The initialization is crucial as it affects convergence and performance. We adopted three initialization methods i.e Random initialization, Xavier initialization and Kaiming Initialization and tested results for each.
- **Forward Propagation:** The forward propagation phase computes the activations of hidden and output layers, culminating in the network's predictions. We apply the ReLU activation function for the hidden layer and the Softmax function for the output layer. The equations are as follows:

$$Z^{[1]} = W^{[1]}X + b^{[1]} \tag{1}$$

$$A^{[1]} = g_{\text{ReLU}}(Z^{[1]}) \tag{2}$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]} \tag{3}$$

$$A^{[2]} = g_{\text{softmax}}(Z^{[2]}) \tag{4}$$

where:

$W^{[1]}$ weight matrix of the first hidden layer
$b^{[1]}$ bias vector of the first hidden layer
$Z^{[1]}$ linear output of the first hidden layer
$A^{[1]}$ activation of the first hidden layer
$g_{\text{ReLU}}(z^{[1]})$ is the ReLU activation function
$W^{[2]}$ weight matrix of the output layer
$b^{[2]}$ bias vector of the output layer
$Z^{[2]}$ linear output of the output layer
$A^{[2]}$ activation of the output layer
$g_{\text{softmax}}(z^{[2]})$ softmax activation function

- **Backward Propagation:** Backpropagation computes gradients of the loss function with respect to the network parameters, enabling optimization. The derivatives are calculated using chain rule. Equations for gradients are:

$$dZ^{[2]} = A^{[2]} - Y \tag{5}$$

$$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T} \tag{6}$$

$$dB^{[2]} = \frac{1}{m} \sum dZ^{[2]} \tag{7}$$

$$dZ^{[1]} = W^{[2]T} dZ^{[2]} \cdot g^{[1]\prime}(Z^{[1]}) \tag{8}$$

$$dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]T} \tag{9}$$

$$dB^{[1]} = \frac{1}{m} \sum dZ^{[1]} \tag{10}$$

where:

α is the learning rate
m is the number of training examples
$A^{[1]}$ is the activation of the first hidden layer
$A^{[0]}$ is the input features
$dZ^{[2]}$ is the derivative of the loss w.r.t. $Z^{[2]}$
$dW^{[2]}$ is the gradient of the loss w.r.t. $W^{[2]}$
$db^{[2]}$ is the gradient of the loss w.r.t. $b^{[2]}$
$dZ^{[1]}$ is the derivative of the loss w.r.t. $Z^{[1]}$
$dW^{[1]}$ is the gradient of the loss w.r.t. $W^{[1]}$
$db^{[1]}$ is the gradient of the loss w.r.t. $b^{[1]}$
$g^{[1]\prime}(z^{[1]})$ derivative of the activation function
Y is the ground truth one-hot encoded vector

- **Updating Parameters:** Parameters are updated using gradient descent, with the learning rate controlling the step size towards optimization. The equations are:

$$W^{[2]} := W^{[2]} - \alpha dW^{[2]} \tag{11}$$

$$b^{[2]} := b^{[2]} - \alpha db^{[2]} \tag{12}$$

$$W^{[1]} := W^{[1]} - \alpha dW^{[1]} \tag{13}$$

$$b^{[1]} := b^{[1]} - \alpha db^{[1]} \tag{14}$$

where:

α is the learning rate

$dW^{[2]}$ and $db^{[2]}$gradients of the loss w.r.t $W^{[2]}$ and $b^{[2]}$

$dW^{[1]}$ and $db^{[1]}$gradients of the loss w.r.t $W^{[1]}$ and $b^{[1]}$

- **Training the Network:** The network undergoes training iteratively, refining its parameters to minimize the loss function. The forward and backward propagation steps are executed in each iteration.
- **Making Predictions:** With trained parameters, the network predicts labels for unseen data instances. The class with the highest probability is considered the predicted class.
- **Evaluating Performance:** Performance evaluation involves computing accuracy by comparing predicted labels with ground truth labels. Accuracy is defined as the ratio of correct predictions to the total number of instances.

B. ACTIVATION FUNCTION

Activation functions are crucial components of artificial neural networks (ANNs) that introduce non-linearity into the network's computations. They play a significant role in enabling ANNs to learn and represent complex relationships in data, making them capable of handling a wide range of tasks, from image recognition to language processing. Activation functions determine whether a neuron should be activated (fire) or not based on the input it receives.

- **Rectified Linear Unit (ReLU):** ReLU is one of the most widely used activation functions in deep learning. It computes the output as the maximum of zero and the input value. Mathematically, ReLU is defined as:

$$\text{RELU} = \max(0, x) \tag{15}$$

ReLU provides computational efficiency and helps mitigate the vanishing gradient problem by preventing gradients from becoming too small during backpropagation.

- **Sigmoid:** Sigmoid is another common activation function that maps input values to a range between 0 and 1. It has a smooth S-shaped curve and is mathematically defined as:

$$\text{Sigmoid} = \frac{1}{1 + e^{-x}} \tag{16}$$

Sigmoid is useful for binary classification problems as it squeezes the output between 0 and 1, representing probabilities.

- **Hyperbolic Tangent (tanh):** Similar to the sigmoid function, the tanh function maps input values to a range between -1 and 1. It has a symmetric S-shaped curve and is mathematically defined as:

$$\tanh = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{17}$$

- **Softmax** The softmax function is a specialized activation function commonly used in the output layer of artificial neural networks (ANNs) for multiclass classification problems. It transforms the raw output scores of the network into probability distributions, allowing the network to assign a probability to each possible class label. This makes it suitable for scenarios where an input belongs to one of multiple classes, and the network needs to provide a probability distribution over those classes. It is mathematically defined as:

$$\text{Softmax} = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}} \text{ for } i = 1, 2, \ldots, N \tag{18}$$

C. PARAMETER INITIALIZATION METHODS

In the realm of training artificial neural networks, choosing the right weight initialization method is a critical aspect that can profoundly influence the network's convergence speed and overall performance.There are various ways to initialize parameters while making the neural network from scratch. Most commonly used method is random initialization. In this project we have implemented the neural network from three initialization methods i.e random, Kaiming and Xavier initialization methods.

- **Random Initialization:** Random initialization is a basic approach where the weights of the neural network are initialized with random values drawn from a specified distribution. While simple, this method can lead to challenges during training. If the weights are too large, it can result in exploding gradients, causing unstable training. Conversely, if the weights are too small, the gradients can vanish, leading to slow convergence or a complete halt in learning.
- **Xavier Initialization:** Xavier initialization, also known as Glorot initialization, was designed to address the challenges posed by random initialization. It takes into account the number of input and output units in a layer to determine the scale of the initial weights. By normalizing the initialization variance, Xavier initialization ensures that the variance of the activations remains relatively constant throughout the network. It works well with activation functions like sigmoid and hyperbolic tangent, which have bounded outputs.
  The formula for Xavier initialization is:

$$\text{Var} = \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}} \tag{19}$$

In Xavier initialization, weights are randomly sampled from a distribution with zero mean and variance as described above.

- **Kaiming Initialization:** the Rectified Linear Unit (ReLU) activation function. ReLU introduces non-linearity by setting negative values to zero, which can lead to the "dying ReLU" problem if not initialized properly. Kaiming initialization addresses this by taking into account the slope of the ReLU activation. This

method helps prevent neurons from becoming inactive during training, facilitating better gradient flow and learning.The formula for Kaiming initialization is:

$$\text{Var} = \sqrt{\frac{2}{n_{\text{in}}}} \tag{20}$$

In Kaiming initialization, weights are randomly sampled from a distribution with zero mean and variance as described above, but the scaling factor is adjusted to better suit the properties of ReLU.

D. DROPOUT LAYER IN ARTIFICIAL NEURAL NETWORKS

Dropout is a regularization technique used in artificial neural networks to prevent overfitting and improve the generalization of the model. It works by randomly "dropping out" or deactivating a certain percentage of neurons during each forward and backward pass of training. This prevents the network from relying too heavily on any individual neuron, making it more robust and reducing the risk of overfitting. These are the implementation steps:

- **Introduction of Dropout:** During training, dropout randomly sets a fraction of the neurons' outputs to zero. The fraction to be dropped out is determined by a hyperparameter called the dropout rate, typically between 0.2 and 0.5.
- **Forward Pass:** During the forward pass, dropout masks are generated for each layer with dropout. A dropout mask is a binary matrix of the same shape as the layer's output. Each element in the mask is set to 0 with probability (1 - dropout rate) or 1 with probability (dropout rate). The layer's output is then element-wise multiplied by the mask.
- **Backward Pass:** During the backward pass, the same dropout mask is applied to the gradients. Neurons that were dropped out during the forward pass will have zero gradients. This effectively prevents those neurons from contributing to the weight updates.

E. INSTRUMENTATION

In the case of implementing an Artificial Neural Network (ANN) from scratch, we can use various libraries and tools to aid in the process. Here are some tools and libraries commonly used for implementing ANNs:

- **Python:** The primary programming language for implementing ANNs from scratch.
- **NumPy:** A fundamental library for numerical computations. It provides support for multi-dimensional arrays and mathematical functions, making it essential for matrix operations in ANNs.
- **Matplotlib or Seaborn:** These libraries are used for data visualization, including plotting loss curves, accuracy trends, and other metrics over epochs.
- **Jupyter Notebooks:** Interactive notebooks that allow you to run code in segments, visualize data, and analyze results. They are great for experimenting and documenting our implementation.
- **Pandas:** Useful for data manipulation and analysis. It helps to organize and preprocess data efficiently.

F. EVALUATION METRICES

The metrics below are commonly used to evaluate the performance of implementation of ANN from scratch. They provide insights into the model's accuracy, precision, recall, and overall predictive capability.

Confusion Matrix:

	Predicted Class	
Actual Class	Positive	Negative
Positive	TP	FN
Negative	FP	TN

Accuracy:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \tag{21}$$

Precision:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \tag{22}$$

Recall:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \tag{23}$$

F1 Score:

$$\text{F1 Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{24}$$

G. SYSTEM ARCHITECTURE

The system architecture for ANN is shown in the FIGURE 1. These are the steps shown in the system architecture:

- **Collect Data:** Begin by collecting your dataset, which should consist of input features and corresponding target labels.
- **Analyze Data:** Understand the characteristics of your dataset, including the distribution of features and labels, potential outliers, and any preprocessing steps required.
- **Train-Test Split:** Split your dataset into two subsets: a training set and a testing/validation set. The training set is used to train the model, while the testing/validation set is used to evaluate its performance.
- **Choose Number of Layers in Network:** Decide on the architecture of your neural network, including the number of hidden layers. The architecture can vary based on the complexity of the problem.
- **Choose Number of Neurons for Each Layer:** Determine the number of neurons in each hidden layer. This choice can impact the network's ability to learn patterns in the data.
- **Choose Activation Function for Each Layer:** Select appropriate activation functions for each hidden layer. Common choices include ReLU (Rectified Linear Unit)

for intermediate layers and softmax for the output layer in classification tasks.

- **Choose Feature Initializer:** Decide on a weight initialization technique for the network's parameters. Options include random initialization, Xavier initialization, and Kaiming initialization.
- **Initialize Parameters:** Initialize the weights and biases for each layer based on your chosen weight initialization technique.
- **Perform Forward Pass:** For each data point in the training set, perform the forward pass through the network. Calculate the weighted sum of inputs, apply activation functions, and generate predictions.
- **Perform Backward Pass:** Calculate the loss between predicted and actual values for each data point. Perform the backward pass to compute gradients using backpropagation. This involves computing the gradients of the loss with respect to each parameter.
- **Update Parameters:** Use the computed gradients to update the network's parameters (weights and biases). The learning rate determines the step size in the parameter update.
- **Perform Gradient Descent:** Iterate through the entire training set, performing forward and backward passes, and updating parameters. This is the core of the training process. Repeat this process for multiple epochs.
- **Make Predictions:** After training, use the trained network to make predictions on the testing/validation set or new unseen data.
- **Evaluate Performance:** Evaluate the model's performance on the testing/validation set using appropriate evaluation metrics (accuracy, precision, recall, F1-score, etc.).

III. RESULT

A. DATASET

The MNIST dataset used in the implementation of the artificial neural network from scratch consists of a total of 42,000 instances, each represented as a row in a Pandas DataFrame. The dataset has 785 columns, with each column corresponding to a specific attribute or feature. Some of the characteristics of dataset are given below:

- **Total Instances:** The dataset contains 42,000 instances, making it a reasonably sized dataset for training and testing machine learning models.
- **Features:** Each instance in the dataset is associated with 785 features. These features include both the pixel values of the grayscale images and the label assigned to each image. The label is present in the "label" column, while the pixel values are spread across the "pixel0" to "pixel783" columns.
- **Data Type:** The dataset contains integer values, as indicated by the "int64" data type for all columns.
- **Memory Usage:** The dataset consumes approximately 251.5 megabytes of memory, which is manageable for most modern computing systems.

The dataset's primary purpose is to serve as the training and testing data for building an artificial neural network (ANN) from scratch. The pixel values in the "pixel" columns represent the intensity of the grayscale color in the range of 0 to 255. These pixel values collectively form the visual representation of the handwritten digits.

The "label" column holds the ground truth labels for each instance, indicating the actual digit that the image represents. This label information is crucial for training the ANN to predict and classify the handwritten digits accurately. Given the nature of the dataset, the goal of the implementation is to develop an artificial neural network that can learn from the pixel values and associated labels, enabling it to correctly classify handwritten digits. The dataset's large number of instances and its comprehensive set of features provide ample training data for the neural network to learn and generalize from.

MNIST dataset used in this implementation provides a rich and diverse collection of handwritten digit images, each with its associated label, making it an ideal choice for practicing the creation of an artificial neural network from scratch.

B. ANALYSIS OF ANN FROM SCRATCH IN DIGIT DATASET (MNIST DATASET)

We began by exploring various activation functions, including ReLU, Tanh, Sigmoid, and Softmax, along with their derivatives, which are visualized in the figures FIGURE 2, FIGURE 3, FIGURE 4, FIGURE 5, FIGURE 6, FIGURE 7 and FIGURE 8. Each activation function has its characteristics and plays a crucial role in shaping the network's performance.

Moving on to the initial implementation of the neural network without multiple hidden layers and dropout, we achieved a training accuracy of 85%. The accuracy versus iteration curve is illustrated in the FIGURE 9, showcasing how the model's performance improves over training iterations. This curve showcased the model's progression towards convergence and underscored the improvement in accuracy over successive iterations. Importantly, the testing accuracy aligned with the training accuracy at 85%, indicating a model that was successfully generalizing its learned patterns to unseen data.

However, when we introduced multiple hidden layers to the network architecture and employed sigmoid and tanh activation functions, a drop in accuracy ensued, bringing it down to 70% which can be seen in FIGURE 10. This phenomenon was rooted in the vanishing gradient problem, where gradients diminish as they propagate through numerous layers, causing a slowdown in learning. Consequently, the testing accuracy displayed a parallel decrease, settling at 68.4%. This outcome highlighted the challenges of training deeper networks and the need to address gradient-related issues.

Furthering our analysis, we delved into the realm of weight initialization techniques. By employing Xavier initialization, we observed a notable reduction in accuracy to 25% as seen

in FIGURE 12. This stark difference underlined the pivotal role of proper weight initialization in ensuring a stable and effective training process. Conversely, Kaiming initialization emerged as a game-changer, boosting the accuracy substantially to 87% as seen in FIGURE 13. This emphasized the effectiveness of initializing weights while considering the specific activation functions in use.

In our pursuit of refining model robustness, we introduced dropout layers into the network architecture. This regularization technique introduced controlled randomness during training, preventing overfitting and enhancing the model's ability to generalize. As a result, the accuracy improved to 77% as seen in FIGURE 11, underscoring the significance of dropout in enhancing model generalization.

In summary, our study unveiled the critical role of activation functions, initialization methods, and architectural choices in shaping the performance of artificial neural networks. Through systematic exploration and analysis, we gained insights into strategies for optimizing accuracy and achieving robust models.

IV. DISCUSSION AND ANALYSIS

This project unveiled a multifaceted landscape of insights and findings that contribute to a comprehensive understanding of artificial neural networks (ANNs) and their intricate dynamics. Delving into the nuances of our implementation, we unearthed valuable observations and considerations that shed light on the underlying mechanisms of ANNs and their impact on performance.

Firstly, the exploration of various activation functions revealed their distinct characteristics and effects on network behavior. The Rectified Linear Unit (ReLU) stood out for its ability to mitigate the vanishing gradient problem and enhance training speed. Sigmoid and Tanh functions exhibited saturation issues, impairing their efficacy in deeper architectures. Meanwhile, the Softmax function demonstrated its importance in multi-class classification scenarios, enabling probabilistic predictions across multiple classes.

The initial implementation of the neural network, without multiple hidden layers and dropout, showcased commendable training accuracy of 85%. However, this achievement masked the potential challenges of overfitting and limited generalization capabilities, which were addressed in subsequent stages of the project.

Introducing multiple hidden layers into the architecture revealed the intricacies of training deep networks. The decline in accuracy to 70% emphasized the hurdles of gradients vanishing in the deeper layers. This led to the exploration of dropout layers, which provided a noteworthy improvement in testing accuracy to 77%. Dropout proved instrumental in preventing overfitting and bolstering model generalization, highlighting its importance as a regularization technique.

Weight initialization strategies, particularly Xavier and Kaiming, emerged as critical factors affecting model convergence and performance. Xavier initialization struggled to provide effective weight initialization, leading to a stark drop in accuracy to 25%. In contrast, Kaiming initialization resulted in a substantial accuracy boost to 87%, underscoring its compatibility with specific activation functions and network architectures.

The analysis of our project also underscored the significance of hyperparameter tuning. Parameters such as learning rate, batch size, and the number of iterations critically influenced training dynamics and final accuracy. The delicate balance between avoiding convergence stagnation and overshooting optimal parameters was highlighted through iterative adjustments.

Beyond technical aspects, the project offered broader insights into the challenges and intricacies of building ANNs from scratch. The journey revealed the importance of understanding activation functions, weight initialization, regularization techniques, and hyperparameters. It also underscored the necessity of experimentation, continuous learning, and adaptation to optimize model performance.

V. CONCLUSION

In conclusion, the implementation of an Artificial Neural Network (ANN) from scratch has unraveled the intricate layers of deep learning and solidified our understanding of its underlying mechanisms. By precisely crafting each component, from activation functions to weight initialization methods, we have not only constructed a functional neural network but also gained invaluable insights into the nuances that dictate its behavior. This project has underscored that while the conceptual framework of ANNs might appear straightforward, their successful implementation requires proper knowledge and attention to detail.

Through our experimentation, we have witnessed the impact of different architectural choices on the network's performance. The initial baseline implementation, even without multiple hidden layers and dropout, demonstrated a training accuracy of 85%, showcasing the potential of our handcrafted network. However, upon introducing multiple hidden layers and varying activation functions, we encountered the delicate balance between model complexity and overfitting. The accuracy drop to 70% with sigmoid and tanh activations revealed the challenges posed by gradient vanishing and exploding issues. Leveraging Xavier and kaiming weight initializations highlighted the pivotal role of proper initialization techniques in overcoming these hurdles, yielding accuracy improvements. The incorporation of dropout layers, as a regularization technique, showcased the potential for combating overfitting, evident in the 77% accuracy achieved. In essence, this project has not only equipped us with a functional neural network but has also underscored the iterative nature of model development and the importance of informed decision-making at each stage.

References

[1] N. Kushawaha and A. Roy, *"A study of Artificial Neural Network and its implementation from scratch"*, June 2022.
[2] K. Pykes, *"Algorithms From Scratch: Artificial Neural Network. Detailing and Building An Artificial Neural Network From Scratch"*, Pykes Techni-

cal Notes,[Online]. Available: https://medium.com/pykes-technical-notes/algorithms-from-scratch-artificial-neural-network-1967e446b29
[3] Nagesh Singh Chauhan, *"Build an Artificial Neural Network From Scratch: Part 1"*, November 2019, [Online]. Available: https://www.kdnuggets.com/2019/11/build-artificial-neural-network-scratch-part-1.html
[4] Samson Zhang, *"Simple MNIST NN from scratch (numpy, no TF/Keras)"*, [Online]. Available: https://www.kaggle.com/code/wwsalmon/simple-mnist-nn-from-scratch-numpy-no-tf-keras/notebook.

RASHMI KHADKA is a driven individual currently pursuing a Bachelor's degree in Computer Engineering at Thapathali Campus. She possesses a strong passion for machine learning and data mining, consistently seeking to delve into the latest advancements in these domains. While Rashmi may not have amassed notable achievements at this point, her enthusiasm and determination to learn and apply state-of-the-art technologies make her a promising and ambitious figure in the field of computer engineering.

SHIWANI SHAH is a dedicated individual currently studying Bachelor's in Computer Engineering at Thapathali Campus. With a strong passion for research and innovation, Shiwani actively engages in various projects and practical applications to apply her knowledge. Her continuous quest for learning drives her to stay updated with the latest advancements and trends in the field. Equipped with expertise in machine learning and data science, Shiwani's relentless determination, inquisitive mindset, and commitment to technology position her to make significant contributions to the ever-evolving landscape of the industry.

APPENDIX A: FIGURES

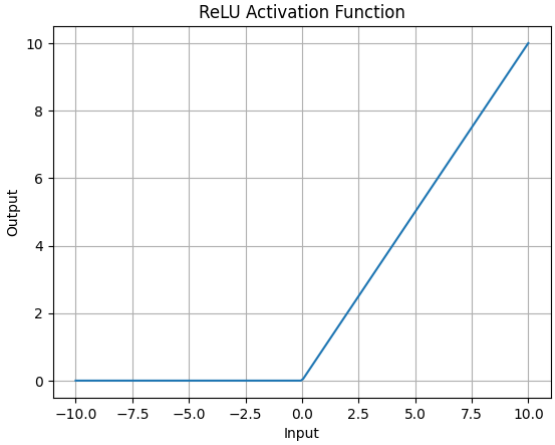

FIGURE 2: RELU activation function

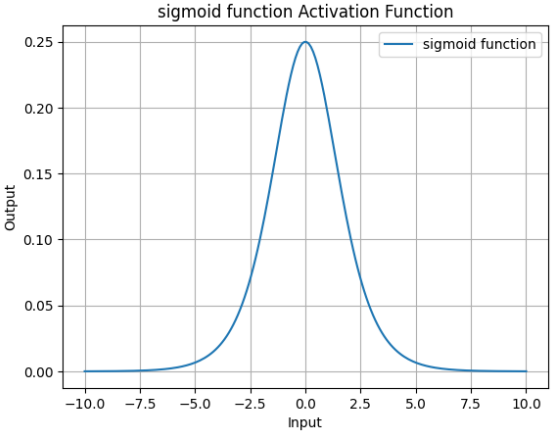

FIGURE 5: Derivative of sigmoid function

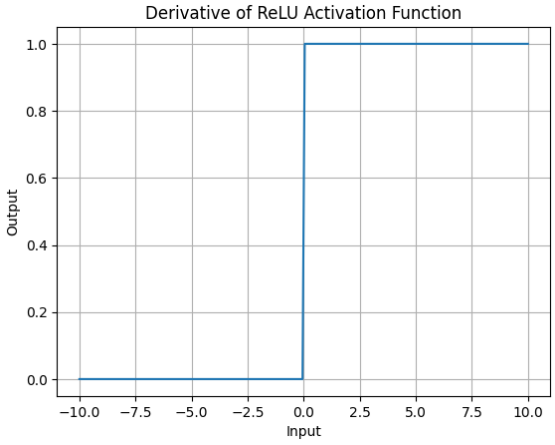

FIGURE 3: Derivative of RELU activation function

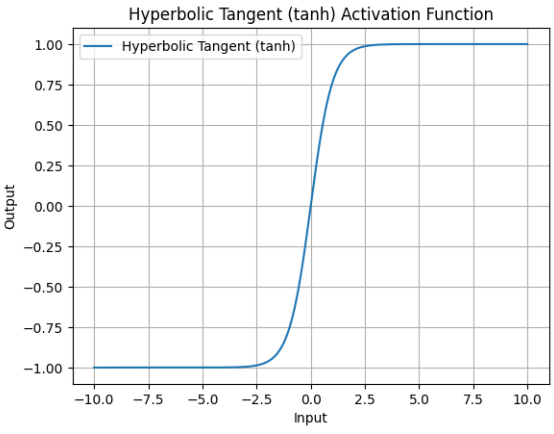

FIGURE 6: Hyperbolic tangent (tanh) function

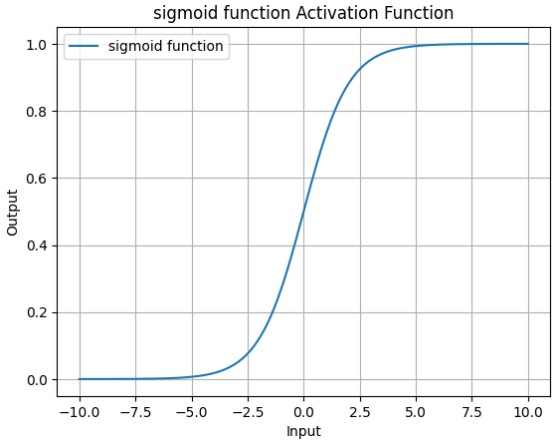

FIGURE 4: Sigmoid function

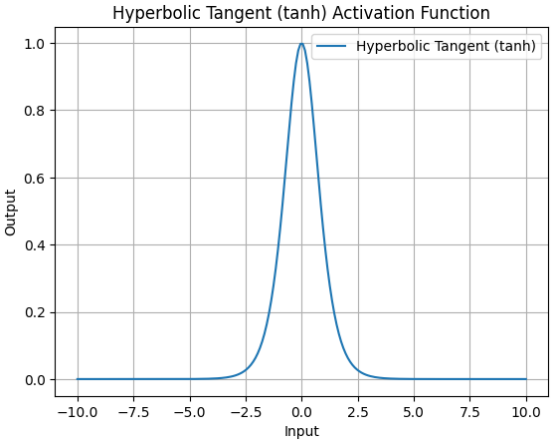

FIGURE 7: derivative of hyperbolic tangent function

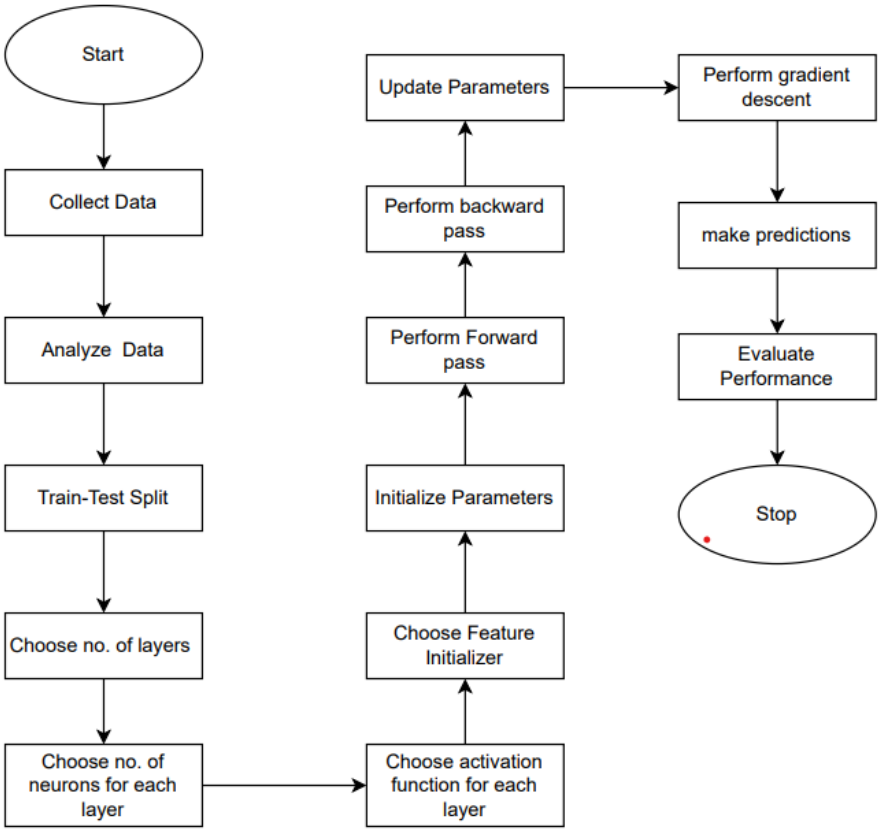

FIGURE 1: System Diagram of ANN

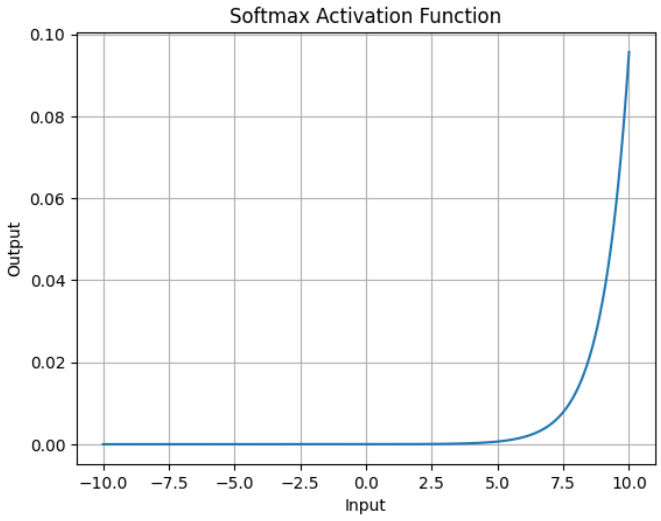

FIGURE 8: Softmax Function

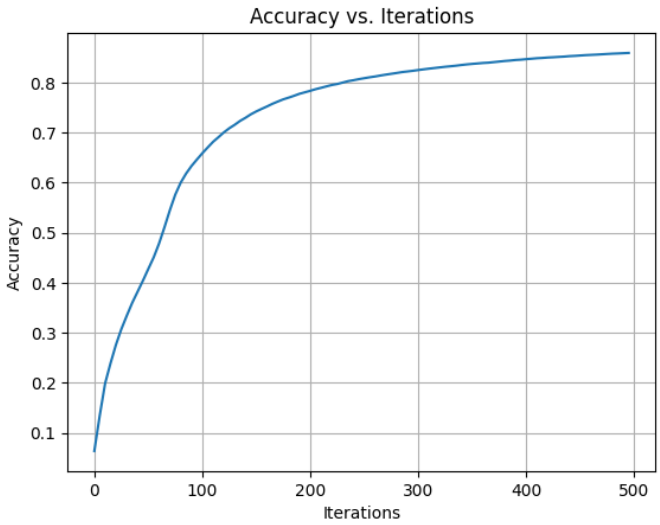

FIGURE 9: Plot for accuracy vs iteration

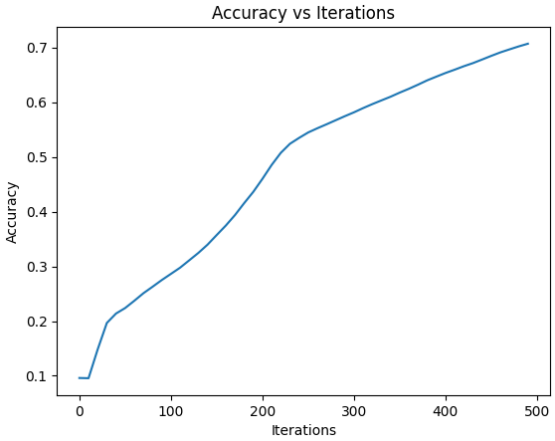

FIGURE 10: Plot for accuracy vs iterations for multiple hidden layer

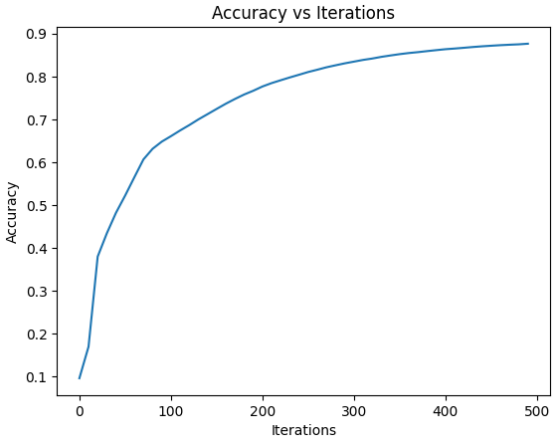

FIGURE 13: Plot for accuracy vs iterations with kaiming initialization

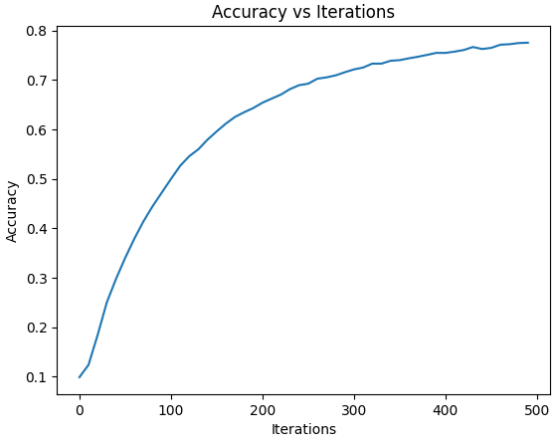

FIGURE 11: Plot for accuracy vs iterations in network with dropout layer

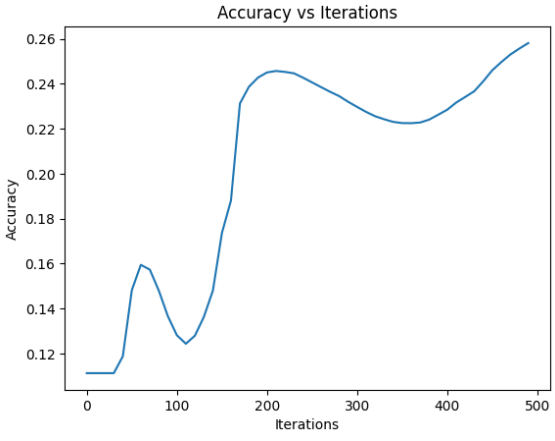

FIGURE 12: Plot for accuracy vs iterations with Xavier initialization

APPENDIX B:DERIVATION OF BACKWARD PROPAGATION

Equations for forward pass are:

$$z^{[1]} = w^{[1]}A^{[0]} + b^{[1]}$$

$$A^{[1]} = \text{ReLU}(z^{[1]})$$

$$z^{[2]} = w^{[2]}A^{[1]} + b^{[2]}$$

$$A^{[2]} = \text{Softmax}(z^{[2]})$$

The Cross-Entropy loss is calculated as:

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

Where:

y_{ik} true probability value for k-th class$i$-th data instance

$A_{ik}^{[2]}$predicted probability value for$k$-th class for$i$-th data

$L_i$loss value for$i$-th data instance

The Cost function is calculated as:

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

Where:

m is the total number of data points

The 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 the Softmax function, we need to calculate the derivative of Softmax which is:

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

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

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

Since y_{ik} is the true label and $A_{ik}^{[2]}$ is the predicted probability, the equation simplifies to:

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

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 consists 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 upcoming value as a singular data point.

Now the change in weight for $w^{[2]}$ for a single data point is calculated as:

$$\Delta w^{[2]} = \frac{\partial C}{\partial w^{[2]}} = \frac{\partial C}{\partial A^{[2]}} \cdot \frac{\partial A^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial w^{[2]}} = \frac{1}{m}\Delta z^{[2]} \cdot A^{[1]}$$

The weights are updated when the whole dataset is passed through the network. Considering them as a matrix which consists of all the data points, we must update the weight by taking the average change in weight for each data point. $\frac{1}{m}$ is introduced as we need to calculate the average weight value as the weight value remains the 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 to ensure the dimensions are consistent.

Similar calculations can be done for the bias term:

$$\Delta b^{[2]} = \frac{\partial C}{\partial b^{[2]}} = \frac{1}{m}\sum \Delta z^{[2]}$$

For the calculation of $\Delta z^{[1]}$, we can apply a similar chain rule:

$$\Delta z^{[1]} = \frac{\partial C}{\partial z^{[1]}} = [w^{[2]}]^T \Delta z^{[2]} \cdot \text{ReLU}'(z^{[1]})$$

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

$$\Delta w^{[1]} = \frac{\partial C}{\partial w^{[1]}} = \frac{1}{m}\Delta z^{[1]}[A^{[0]}]^T$$

$$\Delta b^{[1]} = \frac{\partial C}{\partial b^{[1]}} = \frac{1}{m}\sum \Delta z^{[1]}$$