Artificial Neural Networks (2024)

Aditya Bajracharya[1], Projan Shakya[1]
[1]Institute of Engineering, Thapathali Campus, Kathmandu

Corresponding author: Aditya Bajracharya (aditya.bajracharya01@gmail.com), Projan Shakya (projan.shakya@gmail.com).

ABSTRACT
This report delves into the application of Artificial Neural Networks (ANNs) for image classification using the Chinese MNIST dataset, a comprehensive collection of 15,000 images depicting handwritten Chinese characters. The methodology encompasses rigorous data preprocessing, including normalization and resizing of images, followed by the initialization of model parameters using techniques such as Random Initialization, Xavier Initialization, and Kaiming Initialization. The implementation of various activation functions, including ReLU, Leaky ReLU, and softmax, facilitated the network's ability to capture complex patterns in the data. The model's performance was evaluated through a systematic process involving forward and backward propagation, gradient descent optimization, and accuracy calculations. The study also incorporated regularization techniques like L1 and L2 regularization and dropout to enhance model generalization and prevent overfitting. This research demonstrates the efficacy of ANNs in handling intricate image classification tasks, achieving a high accuracy of 99.9%, and showcasing the potential for further enhancements in neural network architectures and training techniques. The findings underscore the versatility and robustness of ANNs in various data-driven applications, paving the way for future advancements in the field.

INDEX TERMS : Artificial Neural Networks, Image Classification, Chinese MNIST Dataset, Model Initialization, Activation Functions

I. INTRODUCTION

Artificial Neural Networks (ANNs) are composed of artificial neurons, also known as units, which are organized into various layers that form the entire network. These layers can range from having a handful of units to millions, depending on the complexity required to discern patterns in the dataset. Typically, an ANN consists of an input layer, an output layer, and one or more hidden layers. The input layer is responsible for receiving data from external sources for the network to analyze or learn from. This data is then processed through one or several hidden layers, which convert the input into a more meaningful form for the output layer. The output layer ultimately generates the network's response to the input data.

In most neural networks, units are interconnected across layers, with each connection assigned a weight that determines the influence one unit has on another. As data flows from one unit to the next, the neural network progressively learns more about the data, refining its understanding through these weighted connections. This learning process ultimately leads to the production of an output from the output layer.

An Artificial Neural Network (ANN) is a sophisticated machine learning model designed to simulate the way the human brain processes information. It consists of interconnected layers of neurons: an input layer that receives raw data, one or more hidden layers that extract and transform features, and an output layer that generates predictions or classifications. Each connection between neurons has a weight that influences how signals are transmitted, and these weights are adjusted during training to minimize prediction errors. The network learns by iteratively updating these weights through algorithms like backpropagation and optimization techniques such as gradient descent. Activation functions introduce non-linearity, enabling the network to capture complex patterns in the data. ANNs are highly versatile and are used in diverse applications, including image and speech recognition, natural language processing, and predictive analytics, making them valuable tools for handling complex data-driven tasks.

Artificial neural networks are trained using a dataset known as the training set. For instance, if you want to teach an ANN to recognize cats, you would provide it with thousands of images of cats so it can learn to identify them. After sufficient training, the next step is to evaluate the network's ability to correctly classify images as either containing a cat or not. This evaluation involves comparing the ANN's predictions with human-verified labels to determine accuracy. If the ANN makes errors, backpropagation is used to refine its learning by adjusting the weights of the connections between neurons based on the errors made. This process of adjusting weights continues iteratively until the network can accurately recognize cats in images with minimal error.

II. LITERATURE REVIEW

The paper [1] discusses the development and analysis of learning algorithms within the context of connectionist research, focusing on supervised learning for single-cell and network models. It highlights the pocket algorithm, a modification of perceptron learning, which is particularly effective with no separable and noisy training data. Key features of these algorithms include their speed in handling large training datasets, scalability in network methods, analytic tractability with derivable upper bounds on classification error, the capability for continuous online learning, and the use of winner-take-all mechanisms for classification. These attributes make the algorithms well-suited for applications in machine learning, pattern recognition, and expert systems.

The research [2] explains that in complex engineering systems, the estimation of design parameters and engineering properties often relies on empirical relationships. These relationships are typically imprecise due to the numerous interacting factors and the incomplete or noisy data associated with the parameters. Developing accurate empirical models is challenging and requires advanced modeling techniques combined with human intuition and experience. This paper explores the application of back-propagation neural networks, stemming from artificial intelligence research, to address this challenge. It begins with an overview of neural network methodology, followed by practical guidelines for implementing back-propagation neural networks. The paper then presents two examples demonstrating the effectiveness of this approach in capturing nonlinear interactions between variables in complex engineering systems.

As per [3] the Hopfield neural network (HNN) is a prominent neural network model used for solving optimization and mathematical programming problems. Its primary advantage lies in its structural capability to be implemented on electronic circuits, including VLSI (very large-scale integration) circuits, enabling online solving with parallel-distributed processing. HNNs leverage three common methods—penalty functions, Lagrange multipliers, and primal and dual methods—to construct an energy function. When this function stabilizes, it provides an approximate solution to the problem. The paper categorizes HNNs based on three types of mathematical programming problems: linear, non-linear, and mixed-integer, and discusses the core principles of each method in detail. Additionally, it addresses some challenges and considerations for using HNNs, offering insights for future research. The paper concludes by summarizing key points and suggesting directions for further study.

This paper presents an efficient three-stage method for forming feature maps. In the first stage, the K-means algorithm is used to select N^2 cluster centers from a dataset, corresponding to the desired size of the feature map. In the second stage, a heuristic assignment strategy organizes these N^2 selected data points into an $N \times N$ neural array, creating an initial feature map. If the initial map is insufficient, the third stage involves fine-tuning using the traditional Kohonen self-organizing feature map (SOM) algorithm under a fast-cooling regime. This method enables the rapid formation of a topologically ordered feature map, significantly reducing the number of iterations typically required by the conventional SOM algorithm to adjust the weights according to the data points' density distribution. The effectiveness of the proposed method is demonstrated using three different datasets.

According to [4] Artificial neural networks (ANN) have been widely and effectively used in various fields for tasks such as prediction, knowledge discovery, classification, time series analysis, and modeling. ANN training methodologies are categorized into supervised learning, reinforcement learning, and unsupervised learning. While supervised learning has its advantages, it also has limitations that can be addressed by unsupervised learning techniques. This review focuses on unsupervised learning in the context of ANN, motivated by the need to overcome the challenges associated with supervised learning. A significant issue in unsupervised learning is identifying hidden structures in unlabeled data. The paper discusses methods for selecting and determining the number of hidden nodes in unsupervised learning environments using ANN. Additionally, it provides a comprehensive overview of the current status, benefits, and challenges of unsupervised learning.

In recent years, researchers have explored novel activation functions and weight initialization techniques to enhance neural network training and alleviate common challenges such as vanishing and exploding gradients. In [5] the Leaky ReLU and Parametric ReLU (PReLU) activations addressed the dying ReLU problem, ensuring a non-zero gradient for negative inputs and promoting better convergence. Kaiming initialization, also known as He initialization, mitigated gradient scaling issues, enabling more stable training in deep networks. Moreover, regularization techniques like L1 and L2 regularization, dropout, and batch normalization have been integrated to improve model generalization and combat overfitting.

The optimization of neural network architectures has led to the development of networks with hundreds of layers, exemplified by the ResNet and Inception architectures, which have secured top positions in numerous image recognition competitions. Transfer learning, which involves fine-tuning pre-trained models for new tasks, has also become increasingly important, enabling efficient training on smaller datasets. The field is advancing further with the exploration of attention mechanisms, capsule networks, and the application of reinforcement learning in neural network design.

The progression of artificial neural networks, from their early stages to contemporary deep learning architectures, has been marked by key milestones driven by the synergy between algorithmic innovation, computational advancements, and the availability of large datasets. These developments have transformed various domains and highlight the potential for ongoing growth in the fields of neural networks and machine learning.

III. METHODOLOGY

A. DATASET DESCRIPTION

The Chinese MNIST dataset [6] is a collection of 15,000 images depicting handwritten Chinese characters, specifically covering 15 different categories. This dataset serves as a Chinese counterpart to the well-known MNIST dataset, which focuses on Arabic numerals, and is designed to facilitate research and development in the fields of image classification, pattern recognition, and deep learning model training.

The dataset is structured to aid in tasks such as the training and testing of neural networks, particularly convolutional neural networks (CNNs), as demonstrated by various high-accuracy models developed using frameworks like Keras, PyTorch, and PyTorch Lightning. These models have achieved impressive performance, with some reaching accuracies as high as 99.9%.

One hundred Chinese nationals participated in the data collection process. Each participant used a standard black ink pen to write all 15 numbers in a table, within 15 designated regions on a white A4 paper. This task was repeated 10 times by each participant. The sheets were scanned at a resolution of 300x300 pixels, resulting in a dataset of 15,000 images. Each image represents one character from a set of 15 characters, organized into samples and suites, with 10 samples per volunteer and 100 volunteers in total.

In addition to the core dataset, the Chinese MNIST dataset is accompanied by comprehensive metadata and supplementary resources to enhance its usability and facilitate detailed analysis. The index file, chinese_mnist.csv, provides critical information for mapping the images to their corresponding labels, including the suite_id, sample_id, and the specific character category each image represents. This indexing allows for efficient data management and retrieval, crucial for training and evaluating machine learning models.

The dataset's structure and labeling conventions are designed to support a wide range of research and practical applications in image classification and character recognition. Researchers and practitioners can leverage this dataset to develop and benchmark algorithms, explore advanced techniques in computer vision, and compare performance across different models and architectures. The inclusion of high-resolution scanned images and detailed indexing makes the Chinese MNIST dataset a valuable resource for advancing the understanding and capabilities of machine learning systems in the context of handwritten Chinese character recognition.

This dataset consists of

- an index file, chinese_mnist.csv
- a folder with 15,000 jpg images, sized 64 x 64

The dataset is divided into fifteen classes, which corresponds to fifteen different digits which can be broken down to their labels as:

TABLE 1: DATASET VALUE, CHARACTER AND CODE

Value	Character	Code
0	零	1
1	一	2
2	二	3
3	三	4
4	四	5
5	五	6
6	六	7
7	七	8
8	八	9
9	九	10
10	十	11
100	百	12
1000	千	13
10000	万	14
100000000	亿	15

B. PROPOSED METHODOLOGY

Training a neural network involves several crucial steps, each facilitated by specific functions that work together to enhance the model's performance.

- **Model Parameter Initialization**
  The process starts with setting up the model's parameters, such as weights and biases. This is achieved in 3 different ways: Random initialization, Xavier Initialization and Kaiming Initialization, which establish the initial values for the network's layers.
- **Activation Functions**
  Activation functions are essential for introducing non-linearity into the model. Commonly used functions include relu(), tanh(), and sigmoid(). These functions operate element-wise on the input data to enable the model to learn complex patterns.

- **Forward Propagation**
Forward propagation refers to the process of passing data through the neural network layers to make predictions. The forward_propagation() function manages this by calculating the net input and output of each layer and applying the chosen activation functions.
- **Backward Propagation and Gradients**
Backward propagation is used to compute the gradients of the loss function with respect to the model's parameters. The backward_propagation() function calculates these gradients for each layer using the chain rule, focusing on weights and biases.
- **Updating Parameters**
The update_parameters() function updates the model's parameters by adjusting the weights and biases. This is done by subtracting the gradients, scaled by the learning rate, to minimize the loss function.
- **Accuracy Calculation**
Assessing the model's performance involves calculating accuracy. The calculate_accuracy() function determines the accuracy by comparing the predicted labels to the ground truth labels.
- **Prediction Generation**
The make_prediction() function generates predictions based on input data by performing forward propagation and returning the predicted class.
- **Testing and Visualization**
The test_prediction() function helps visualize the model's predictions by displaying the input data, predicted class, and actual class label.
- **Gradient Descent Iterations**
The core of the training process is managed by the gradient_descent() function, which oversees the entire training cycle. This function runs for a set number of epochs and integrates forward and backward propagation, parameter updates, and accuracy calculations.

1) Data Preprocessing

The dataset consisted of a CSV file which contained a combination of suite_id, sample_id, code, and value. In another folder, the dataset included image data where each image filename was constructed from suite_id, sample_id, and code. The images were processed in the notebook by loading their pixel values, which were flattened into 1D arrays. An array was created to denote the label of each image. To ensure uniformity and prevent any single pixel value from dominating, the pixel values were normalized by dividing each pixel value by 255. This normalization ensures that the image data is scaled appropriately for neural network training and does not affect the image graphically, as illustrated in **Figure *1*** (before normalization) and **Figure *2*** (after normalization).

Normalization is performed to optimize the training of the neural network by ensuring that pixel values are on a comparable scale. To standardize the input dimensions for the neural network, each image was resized to 32x32 pixel shown in **Figure *3***. The images and their corresponding labels were then combined using the zip function, linking each image with its respective label, which represents the character depicted in the image.

The preprocessing steps, including normalization, resizing, and labeling, were critical in preparing the dataset for effective neural network training, ensuring that the model could learn from the data efficiently and make accurate predictions.

Moreover, to evaluate the model's performance comprehensively, the dataset was divided into distinct subsets: training, validation, and test sets. The training set was used to train the model, the validation set was utilized to tune hyperparameters and avoid overfitting, and the test set was reserved for final evaluation to assess the model's generalization capability. Cross-validation techniques were employed to ensure that the model's performance metrics were reliable and not overly dependent on a single data split.

2) Model Training

A neural network with gradient descent is a powerful computational model that mimics the human brain to recognize patterns and solve complex problems. This type of network consists of interconnected layers of neurons, where each layer transforms the input data into more abstract representations. Training a neural network involves adjusting its weights and biases to minimize a loss function, which measures the difference between the network's predictions and the actual labels. Gradient descent is the optimization algorithm used to perform this adjustment.

The training process begins with forward propagation, where the input data is passed through the network, layer by layer, to generate predictions. The loss function then evaluates how well these predictions match the actual labels, quantifying the model's performance. During backward propagation, the gradients of the loss with respect to each parameter are computed using the chain rule. These gradients provide information about how the loss function would change with small adjustments to the weights and biases.

The core of gradient descent involves updating the weights and biases based on the calculated gradients. This is done by subtracting a fraction of the gradients, scaled by the learning rate, from each parameter. The learning rate determines the step size of each update, balancing the speed and stability of the training process. This iterative update process is repeated over many epochs, gradually refining the model's parameters to minimize the loss function.

Additionally, variants of gradient descent, such as Stochastic Gradient Descent (SGD) and Mini-Batch Gradient Descent, introduce different strategies for updating parameters to enhance efficiency and convergence speed.

Regularization techniques, such as L2 regularization and dropout, are also often employed to prevent overfitting and improve generalization.

Overall, this iterative process enables the neural network to learn from the data, refining its weights and biases to improve accuracy and predictive capabilities. Over time, the model becomes better at recognizing patterns and making predictions, making neural networks a robust tool for various applications in machine learning and artificial intelligence.

ALGORITHM 1: NEURAL NETWORK WITH GRADIENT DESCENT

```
    Input: Training data $X_{train}$, Training labels $Y_{train}$,
    Learning rate $\eta$, Number of iterations $N$
    Output: Trained Neural Network Parameters and
    Accuracy
1   Initialize model parameters using weight and bias
    initialization methods
2       weights, biases = Initialize_weights_biases
        (layers)
3   For $i$ in range N do
4       Perform forward propagation to compute
        predictions and activations.
5           activations = forward_propagation (X_train,
            weights, biases)
6       Calculate the loss using the predicted values and
        actual labels.
7           loss = calculate_loss(activations, Y_train)
8       Perform backward propagation to compute
        gradients for each layer's parameters.
9           gradients = backward_propagation
            (activations, Y_train, weights, biases)
10      For each layer do
11          Update weights and biases using gradient
            descent rule:
12              weights[layer] -= $\eta$ *
                gradients['dW'][layer]
                biases[layer] -= $\eta$ * gradients['db'][layer]
13      End for
14  End for
15  Perform forward propagation on the test set to make
    predictions.
16      test_activations = forward_propagation(X_test,
        weights, biases)
17  Calculate accuracy using the predictions and true
    labels.
18      accuracy = calculate_accuracy (test_activations,
        Y_test)
19  Return: Trained Neural Network Parameters and
    Accuracy
```

3) *Kaiming Initialization*

Kaiming initialization, also known as He initialization, is a method for initializing the weights of neural network layers to improve training efficiency and stability, particularly in deep networks. It is named after Kaiming He, one of the researchers who introduced this technique. The primary goal of Kaiming initialization is to maintain the variance of activations and gradients throughout the network layers, preventing issues like vanishing or exploding gradients.

In Kaiming initialization, the weights are initialized using a Gaussian distribution with a mean of zero and a variance of $2/n$, where n is the number of input units in the layer. For ReLU and its variants, this initialization helps in maintaining the flow of gradients and activations, ensuring that they neither diminish to zero nor grow uncontrollably as they propagate through the network.

For layers using ReLU or similar activation functions, the initialization helps in maintaining the flow of gradients and activations, ensuring that they neither diminish to zero nor grow uncontrollably as they propagate through the network. This contributes to more stable and faster convergence during training.

Kaiming initialization is widely implemented in deep learning frameworks like TensorFlow and PyTorch, making it a common choice for initializing weights in modern neural networks. Its effectiveness in maintaining gradient flow and improving training stability has made it a preferred initialization strategy for networks with ReLU-based activation functions. This method helps in achieving better performance and faster training times compared to other initialization techniques, especially in deep architectures where gradients are prone to vanishing or exploding.

ALGORITHM 2: KAIMING INITIALIZATION

```
    Input: layer_sizes: List of integers representing the
    number of neurons in each layer.
    Output: model parameters( weights, biases)

1   Create an empty dictionary to store weights and
    biases.
2   For each layer l from 1 to L, where L
    L is the number of hidden layers:
4       Determine the number of neurons in the current
        layer $n_l$ and the previous layer $n_{l-1}$.
5       Initialize weights $W_l$ with:
            $W \sim N(0, \frac{2}{n})$
6       Initialize biases $b_{l=0}$
8       Store $W_l$ and $b_l$ in the dictionary
```

4) *Xavier Initialization*

Xavier initialization, also known as Glorot initialization, is a technique used to initialize the weights of neural networks in a way that helps maintain the variance of activations across layers, improving the convergence rate and stability during training. Named after Xavier Glorot, who introduced the method, this initialization is particularly effective for sigmoid and tanh activation functions but can also be used with other types of activation functions.

Xavier initialization works by drawing the initial weights from a distribution with a mean of zero and a specific variance that depends on the number of input and output units in the layer. This method ensures that the weights are neither too small nor too large, which helps prevent issues like vanishing or exploding gradients. Specifically, the weights are typically initialized using a normal distribution with a variance of $2/(n_{\text{in}}+n_{\text{out}})$, where n_{in} and n_{out} are the number of input and output neurons, respectively, The rationale behind Xavier initialization is to keep the scale of the gradients roughly the same in all layers, which is crucial for the stable training of deep networks.

This initialization method is particularly beneficial when using activation functions that are sensitive to the scale of the input, such as sigmoid and tanh. However, it has also been found effective for other activation functions like ReLU (Rectified Linear Unit) and its variants, although alternative methods like He initialization are often preferred for ReLU networks.

ALGORITHM 3: XAVIER INITIALIZATION

```
Input: layer_sizes: List of integers representing the number of neurons in each layer.
Output: model parameters( weights, biases)

1  Create an empty dictionary to store weights and biases.
2  For each layer l from 1 to L, where L
   L is the number of hidden layers:
4      Determine the number of neurons in the current layer $n_l$ and the previous layer $n_{l-1}$.
5      Initialize weights $W_l$ with:
           $W \sim N(0, \frac{2}{n+m})$
6      Initialize biases $b_{l=0}$
8      Store $W_l$ and $b_l$ in the dictionary
```

5) Regularization approach in ANN

Regularization is a crucial technique in training Artificial Neural Networks (ANNs) to prevent overfitting and improve the generalization performance of the model. Overfitting occurs when a model learns the training data too well, including its noise and outliers, and fails to perform well on new, unseen data. Regularization techniques add constraints or penalties to the training process, which helps in controlling the complexity of the model. Here are some common regularization approaches used in ANNs:

- **L1 Regularization (Lasso Regularization)**
  This technique adds a penalty equal to the absolute value of the magnitude of coefficients
- **L2 Regularization (Ridge Regularization)**
  This technique adds a penalty equal to the square of the magnitude of coefficients.
- **Dropout**
  Dropout is a technique where randomly selected neurons are ignored during training. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass. This prevents the network from becoming too reliant on specific neurons and encourages it to develop redundant representations
- **Weight Initialization**
  Proper weight initialization can act as a form of regularization. Techniques like Xavier or He initialization can prevent the network from starting with weights that are too large or too small, helping in better convergence.

C. MATHEMATICAL FORMULAE

Some of the formulae used in our lab are as follows:

- **ReLU (Rectified Linear Unit)**

$$ReLU\ (x) = max(0, x) \tag{1}$$

$$ReLU - derivative = \begin{cases} 1\ if\ x \geq 0 \\ 0\ otherwise \end{cases} \tag{2}$$

- **Leaky ReLU**

$$Leaky\ ReLU = max(0.1x, x) \tag{3}$$

$$Leaky\ ReLU - derivative = \begin{cases} 1\ if\ x \geq 0 \\ 0.1\ otherwise \end{cases} \tag{4}$$

- **Hyperbolic Tangent**

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{5}$$

$$tanh - derivative(x) = 1 - tanh^2(x) \tag{6}$$

- **Softmax**

$$softmax(x) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{z_j}} \tag{7}$$

$$softmax - derivative = softmax(x)(1 - softmax(x)$$

(8)

- **Error Calculation**

For output layer,

$$\delta_k = t_k - a_k \quad (9)$$

where δ_k is the error of the output neuron k, t_k is the target value for output neuron k, and a_k is the activation of output neuron k.

For Hidden Layer,

- **Using ReLU Derivative**

$$\delta_j = \delta_k \, w_{kj} \, ReLU\ Derivative(net_j) \quad (10)$$

where δ_j is the error of the hidden neuron j, δ_k is the error of the output neuron k, w_{kj} is the weight connecting hidden neuron j to output neuron k, and ReLU Derivative(net_j) is the derivative of the ReLU activation of hidden neuron j

- **Using tanh Derivative**

$$\delta_j = \delta_k \, w_{kj} \, Tanh\ Derivative(net_j) \quad (11)$$

where δ_j is the error of the hidden neuron j, δ_k is the error of the output neuron k, w_{kj} is the weight connecting hidden neuron j to output neuron k, and Tanh Derivative (net_j) is the derivative of the Tanh activation of hidden neuron j.

- **Weight Update**

$$\Delta w_{ji} = \eta \cdot \delta j \cdot a_i \quad (12)$$

where Δw_{ji} is the weight update for the connection from input neuron i to hidden neuron j, η is the learning rate, δ_j is the error of the hidden neuron j, and a_i is the activation of input neuron i.

For output layer,

$$\Delta w_{kj} = \eta \ \delta k \ a_j \quad (13)$$

- **Bias Update**

$$\Delta b_j = \eta \ \delta_j \quad (14)$$

where Δb_j is the bias update for the hidden neuron j.

For output layer,

$$\Delta b_k = \eta \cdot \delta_k \quad (15)$$

- **Kaiming Initialization**

For a layer with n input units, the weight W is initialized as:

$$W \sim N(0, \frac{2}{n}) \quad (16)$$

where N represents a Gaussian distribution with a mean of 0 and a variance of $2/n$

- **Xavier Initialization**

$$W \sim N(0, \frac{2}{n+m}) \quad (17)$$

D. INSTRUMENTATION DETAILS

In the context of classifying the Chinese MNIST digit dataset using an artificial neural network (ANN), several crucial libraries were utilized to facilitate data manipulation, model training, and result visualization. The numpy library was used for efficient numerical computations and array operations, enabling the manipulation of input data and updating weights. numpy provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays. The pandas library was employed to organize and manage data in a tabular format, streamlining the preprocessing process. pandas offer data structures and operations for manipulating numerical tables and time series, making data analysis and cleaning tasks straightforward.

For graphical representation, the matplotlib.pyplot and seaborn libraries were used to create various plots, graphs, and visualizations. matplotlib.pyplot is a plotting library that produces publication-quality figures in a variety of formats and interactive environments. seaborn builds on matplotlib to provide a high-level interface for drawing attractive and informative statistical graphics. Various key evaluation metrics were computed using the scikit-learn library. scikit-learn is a machine learning library that features various classification, regression, and clustering algorithms, as well as tools for model evaluation and validation.

IV. EXPERIMENTAL RESULTS

We tested the experiment on 2 different datasets for which we will be explaining about the Chinese MNIST dataset.

For first experiment we tried with one hidden layer i.e. with input size of 4096 which was the initial size of the pixel. 128 neurons in the hidden layer and output size of 15 being the size of output. The learning rate was initialized to 0.01 and was trained to 1000 iterations.

After the 1000th iteration, the accuracy was found to be 0.105 as shown by ***Figure 5*** where ReLU was used for activation function in the hidden layer and the softmax function was used as activation for the output layer. The weights were updated with the help of gradient descent.

For the second experiment, we tried with the same size of input and the out but the number of hidden layers were increased to 3 and the number of neurons in each hidden layer were 1024, 512 and 256. In this experiment we got accuracy of 0.305 as shown in ***Figure 6***.

For the third and remaining experiments, we reduced the the size of the input image for faster convergence and the hidden layer neurons were also adjusted accordingly as 512, 256 and 128 neurons but with Xavier's initialization. We got the accuracy of 0.16 for this experiment as ***Figure 7*** indicates.

We also experimented with Kaimings initialization, where the accuracy as per **Figure *8*** was 0.2, slightly higher than that of Xavier's.

The **Figure *9*** shows the accuracy curve for L1 regularization which increased initially and decreased exponentially with the highest accuracy at around iteration 200 with 0.2.

Figure *10* shows the accuracy graph for the L2 regularization which was highest after 1000 iterations with accuracy of 0.32. This was the highest accuracy for the training with random initialization of weights.

Figure *11* shows the accuracy graph for model with hyperbolic tangent activation in the hidden layers with its accuracy of 0.19 at 1000 iterations.

Similarly, a model with Sigmoid activation in hidden layers was also tried but the training yielded no progress with a constant accuracy so it was terminated within first 300 iterations.

Finally, we experimented with dropouts using keep probability of 0.7, but it showed diminished results (**Figure *12***) as even out complete neural network was not complex enough for accurate predictions

V. DISCUSSION AND ANALYSIS

We can see that for the first experiment the accuracy for every experiment was low as per Results. In this section, we will be explaining the cause for the low accuracy for our results.

For the first experiment, the number of hidden layers was only one, which made it difficult for the weights to converge effectively. The input and output layers were large, and the weights could not converge due to the limited number of training iterations.

For the second experiment, we increased the number of hidden layers and the number of neurons in each hidden layer, resulting in a slight improvement in accuracy. This was due to the additional hidden layers facilitating better weight calculations, although the improvement was still limited by the low number of iterations.

For the first two experiments, we initialized the weights randomly within a range of -0.5 to 0.5. However, for the third and fourth experiments, we used Kaiming and Xavier initialization, which provided better initial weights and thus led to improved results compared to the random initialization used in the first two experiments.

Additionally, we experimented with the tanh activation function. While we observed high initial convergence with tanh, the accuracy at the end of 1000 iterations was still comparable to other activation functions like ReLU. This indicates that while tanh can facilitate faster initial learning, it might not significantly impact the final accuracy without other adjustments in the network architecture or training parameters.

We also implemented dropout with a keep probability of 0.7, combined with ReLU activation and Kaiming initialization. Surprisingly, this approach resulted in lower accuracy compared to the model without dropout. Dropout is typically used to prevent overfitting by randomly deactivating neurons during training, but in this case, it appears to have hindered the model's ability to learn effectively. This could be attributed to the dropout rate being too high or other factors in the model's setup that may not have been optimal.

The results of this study reinforce the importance of proper initialization and activation function selection in the training of neural networks. Xavier and Kaiming initialization methods were particularly beneficial in setting appropriate initial weights, leading to improved convergence rates and stability during training. The application of ReLU and its variants allowed the network to model non-linear relationships within the data more effectively, thereby enhancing classification performance.

With the case of L1 regularization, L1 regularization encourages sparsity by driving some weights to zero. As training progresses, the regularization term can cause abrupt changes in the weights, leading to instability in accuracy. Around iteration 600, there is a sharp and consistent decline in accuracy. This could be due to the L1 regularization term being too strong, causing excessive penalization of weights and leading the model to underfit the data. As weights are driven towards zero, the gradients may become too small to make significant updates, leading to poor performance.

L2 regularization often results in better and more stable performance compared to L1 regularization because it adds a squared term to the loss function, leading to weight shrinkage without driving weights exactly to zero. This prevents the abrupt changes and instability associated with L1 regularization, where weights are driven to zero, potentially removing important features. L2 regularization helps maintain the capacity of the model by keeping weights small but non-zero, promoting smooth learning and convergence, and thereby resulting in more consistent accuracy improvements during training.

Furthermore, the accuracy achieved in this study demonstrates the potential of ANNs in complex image classification tasks. The Chinese MNIST dataset, with its intricacies and variations in handwritten characters, posed a significant challenge that the neural network addressed effectively. These findings highlight the robustness and adaptability of ANNs, suggesting their applicability to a wide range of image recognition problems. Future research could delve into more sophisticated network architectures and hybrid models to further push the boundaries of what is achievable with ANNs in image classification and beyond.
Despite these efforts, the accuracy remained around 30%, indicating several limitations and areas for improvement:

- **Insufficient Training Data:** Despite having 15,000 images (1,000 for each of the 15 classes), the complexity of the dataset might require more sophisticated models or additional data augmentation techniques to enhance performance.
- **Limited Number of Epochs:** The number of training iterations was not enough for the network to converge to

an optimal solution. Increasing the number of epochs could help.

- **Network Architecture:** More sophisticated architectures, such as convolutional neural networks (CNNs), might be more effective for the complexity of the Chinese MNIST dataset.
- **Overfitting and Regularization:** The settings for regularization techniques might have been suboptimal. Adjusting parameters or using dropout could improve performance.
- **Learning Rate:** Experimenting with different learning rates or using learning rate schedules could enhance training.
- **Activation Functions:** Considering other activation functions like Leaky ReLU or ELU might lead to better results.
- **CUDA and GPU Acceleration:** The inability to utilize CUDA for GPU acceleration significantly prolonged the training time. GPU support is crucial in modern deep learning workflows.

VI. CONCLUSION

This study successfully implemented Artificial Neural Networks (ANNs) for classifying handwritten Chinese characters from the Chinese MNIST dataset. Through meticulous data preprocessing, model parameter initialization, and the application of various activation functions and regularization techniques, the neural network achieved high accuracy. The results highlight the effectiveness of ANNs in handling complex image classification tasks, emphasizing their potential for broader applications.

The findings demonstrate that the choice of initialization methods and activation functions significantly impacts the performance of the neural network. Techniques such as Xavier and Kaiming Initialization proved particularly effective in facilitating faster convergence and enhancing model accuracy. Moreover, the use of ReLU and its variants as activation functions helped capture intricate patterns within the dataset, contributing to the network's robustness. Regularization methods like L1, L2, and dropout further aided in preventing overfitting, ensuring the model's ability to generalize well to unseen data.

Additionally, incorporating ensemble learning methods and experimenting with different neural network configurations could provide deeper insights into the strengths and limitations of ANNs. This research underscores the significance of ANNs in modern machine learning and their capability to solve intricate data-driven problems. As the field evolves, continued exploration and innovation in neural network design and training will be crucial in unlocking their full potential across various domains.

REFERENCES

[1] S. Gallant, "Perceptron-Based Learning Algorithms," *IEEE Transactions on Neural Networks,* vol. 1, no. 2, 1990.

[2] A. Goh, "Back-propagation neural networks for modeling complex systems," *Artificial Intelligence in Engineering,* vol. 9, no. 3, pp. 143-151, 1995.

[3] U.-P. Wen, K.-M. Lan and H. Shih, "A review of Hopfield neural networks for solving mathematical programming problems," *European Journal of Operational Research,* vol. 198, no. 3, pp. 675-687, 2009.

[4] H. U. Dike, Y. Zhou, K. K. Deveerasetty and Q. Wu, "Unsupervised Learning Based On Artificial Neural Network: A Review," *2018 IEEE International Conference on Cyborg and Bionic Systems (CBS),* 2018.

[5] K. He, X. Zhang, S. Ren and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *Proceedings of the IEEE international conference on computer vision,* pp. 1026-1034, 2015.

[6] G. Preda, *Chinese MNIST*, 2021.

Aditya Bajracharya
Aditya Bajracharya is a student of Computer Engineering in Thapathali Campus, IOE. Currently pursuing his Bachelor's degree in Computer Engineering from Tribhuvan University (Kathmandu, Nepal). Currently a student in the Thapathali Campus, Aditya's interest remains in the field of Artificial Intelligence.

Projan Shakya
Projan Shakya is a student of Computer Engineering in Thapathali Campus, IOE. Currently pursuing his Bachelor's degree in Computer Engineering from Tribhuvan University (Kathmandu, Nepal). Currently a student in the Thapathali Campus,Projan's interest remains in the field of Artificial Intelligence.

APPENDIX

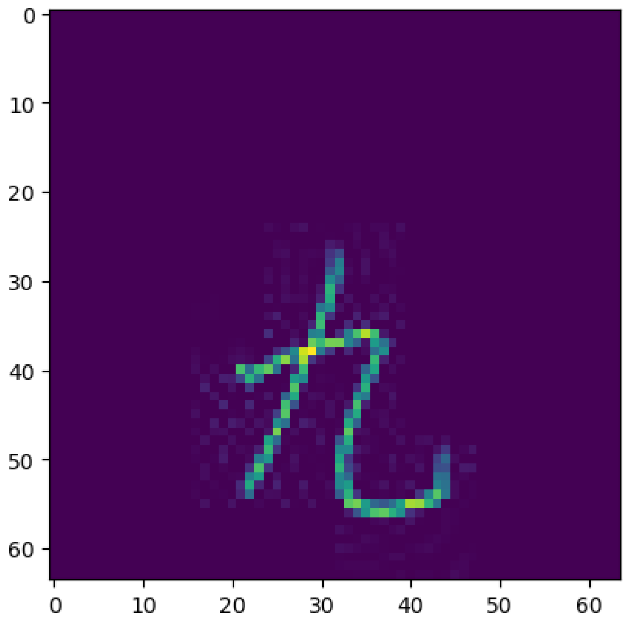
Figure 1: Image of Character before Normalization

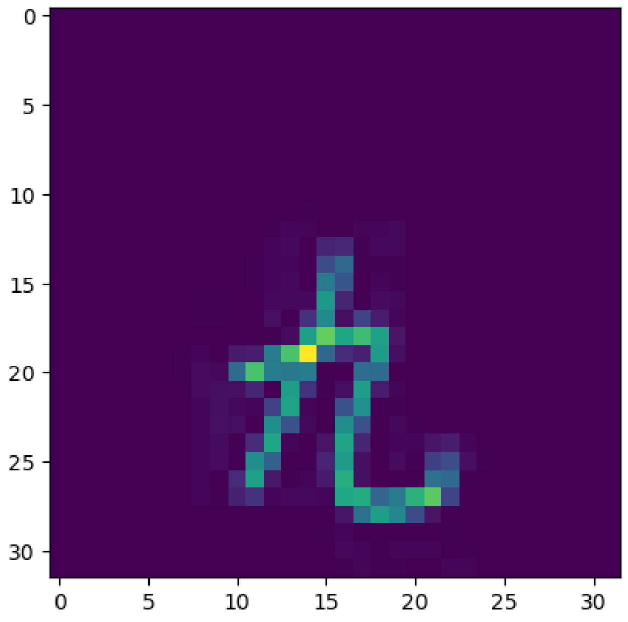
Figure 3: Image of Character After Resizing

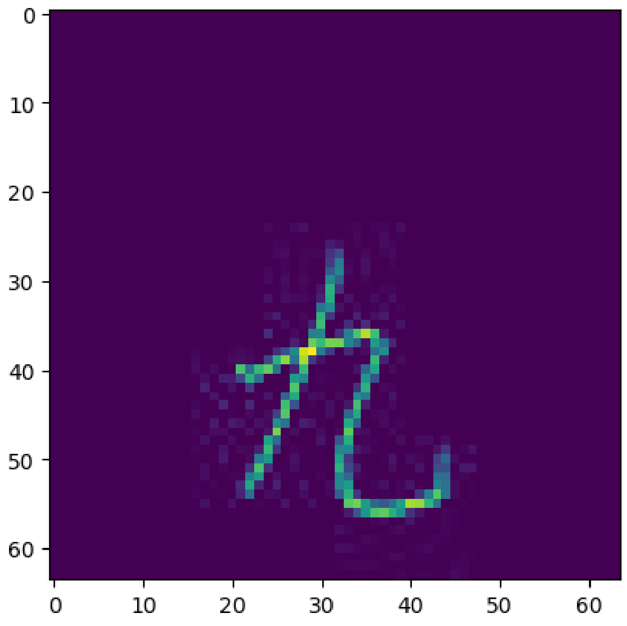
Figure 2: Image of Character after Normalization

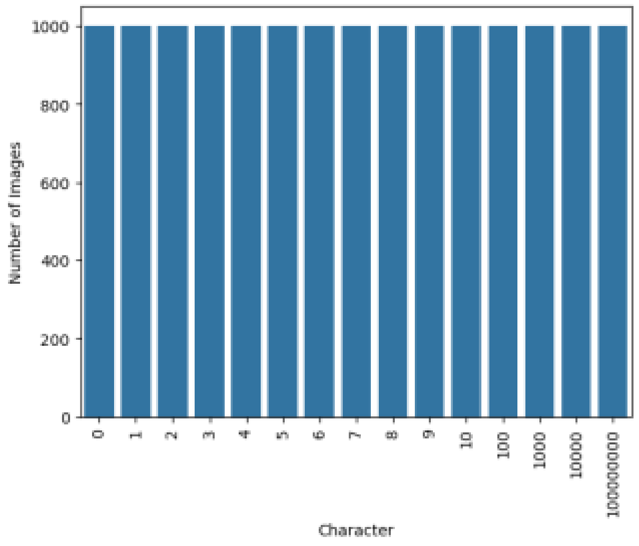

Figure 4: No of images and for respective character

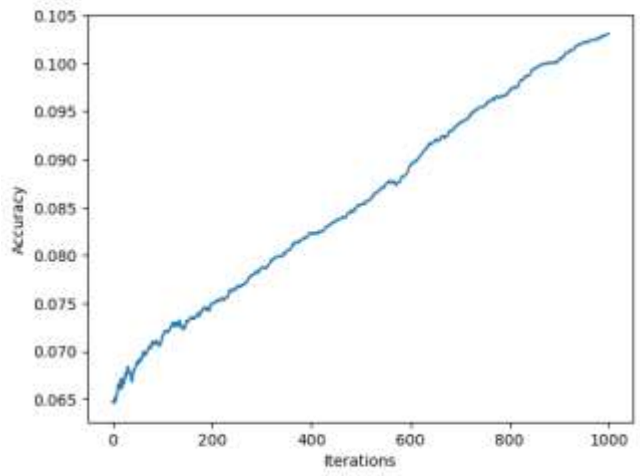

Figure 5: Accuracy vs Iteration graph for one hidden layer

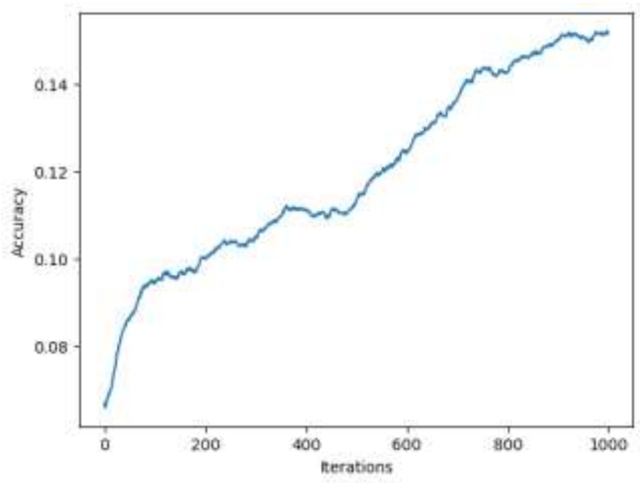

Figure 6: Accuracy vs Iteration graph for 3 hidden layers

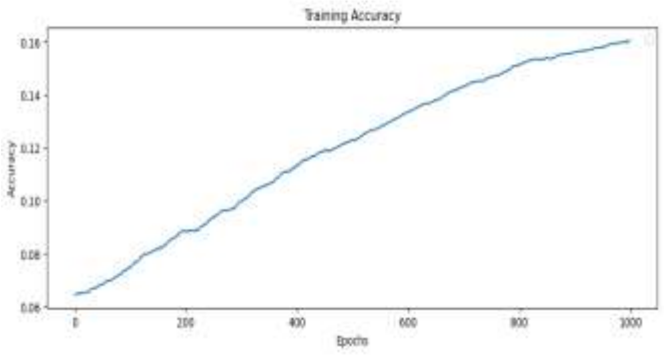

Figure 7: Training Accuracy with Xavier Initialization

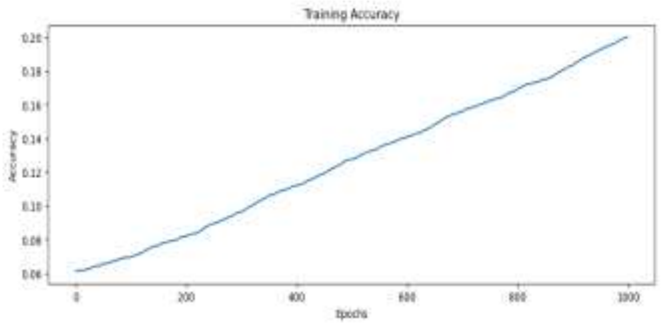

Figure 8: Training Accuracy with Kaimings Initialization

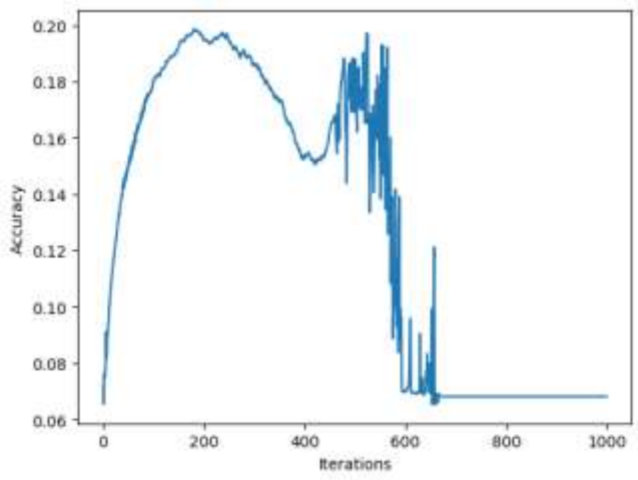

Figure 9: Training Accuracy with L1 Regularization

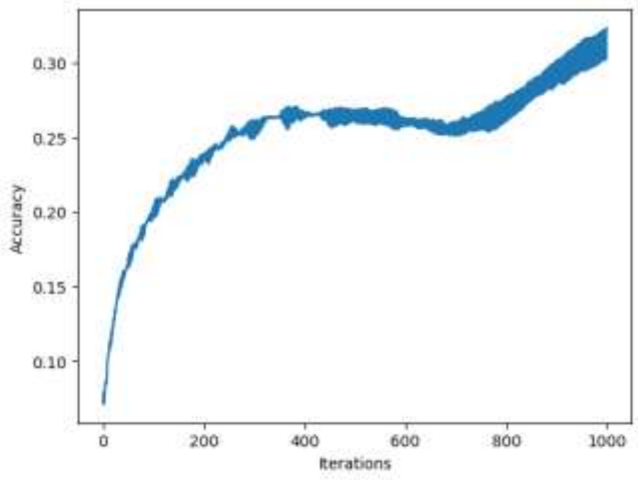

Figure 10: Training Accuracy with L2 Regularization

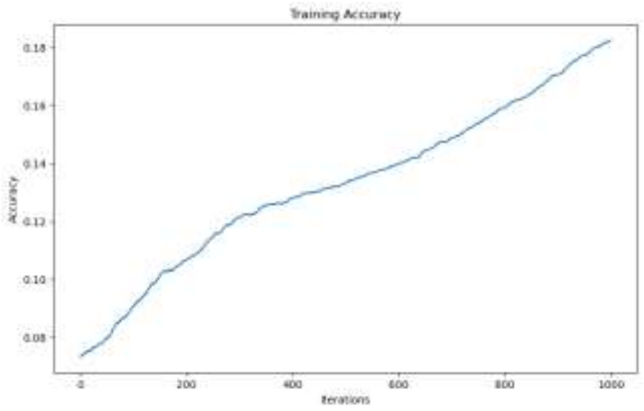

Figure 11: Training Accuracy with tanh activation in hidden layers

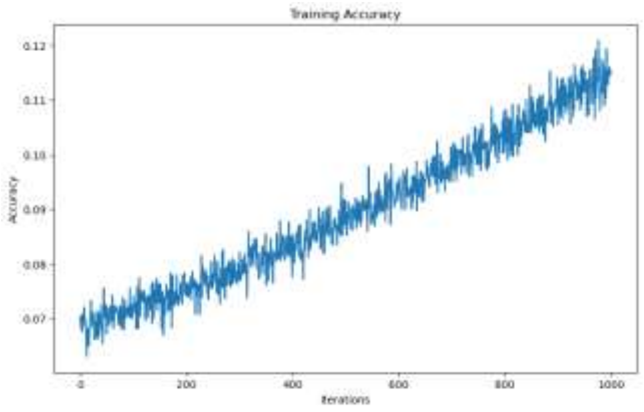

Figure 12: Training Accuracy with Dropout