Project Report: Handwritten Digit Recognition Using Neural Networks

Authors:

Ani Aloyan

Meri Asatryan

Seda Bayadyan

Introduction

This project is focused on building and understanding a neural network that can recognize handwritten digits, allowing users to input their own digits by drawing in our drawing app, not just those from the dataset. We implemented this project in Python and used various libraries such as TensorFlow, Keras, NumPy, and Matplotlib to develop and visualize a model's learning capabilities. In this project, we aim to develop a neural network-based system for recognizing handwritten digits and compare its performance with a pre-built model from the Keras library. This report provides a comprehensive understanding of the project’s structure, including data handling, neural network architecture, training, evaluation, and external source testing.

Data Preparation

The MNIST dataset, which contains 70,000 images of handwritten digits (0-9), is used as the primary dataset. The dataset is split into three parts:

Training set (48,000 images): Used to train the neural network.

Validation set (12,000 images): Used to tune the model parameters and prevent overfitting.

Test set (10,000 images): Used to test the model’s generalization ability.

Each image in the dataset is a 28x28 pixel grayscale image. These images are flattened into a 784-dimensional vector (28x28) and normalized so that pixel values are between 0 and 1. This normalization helps in faster convergence during training.

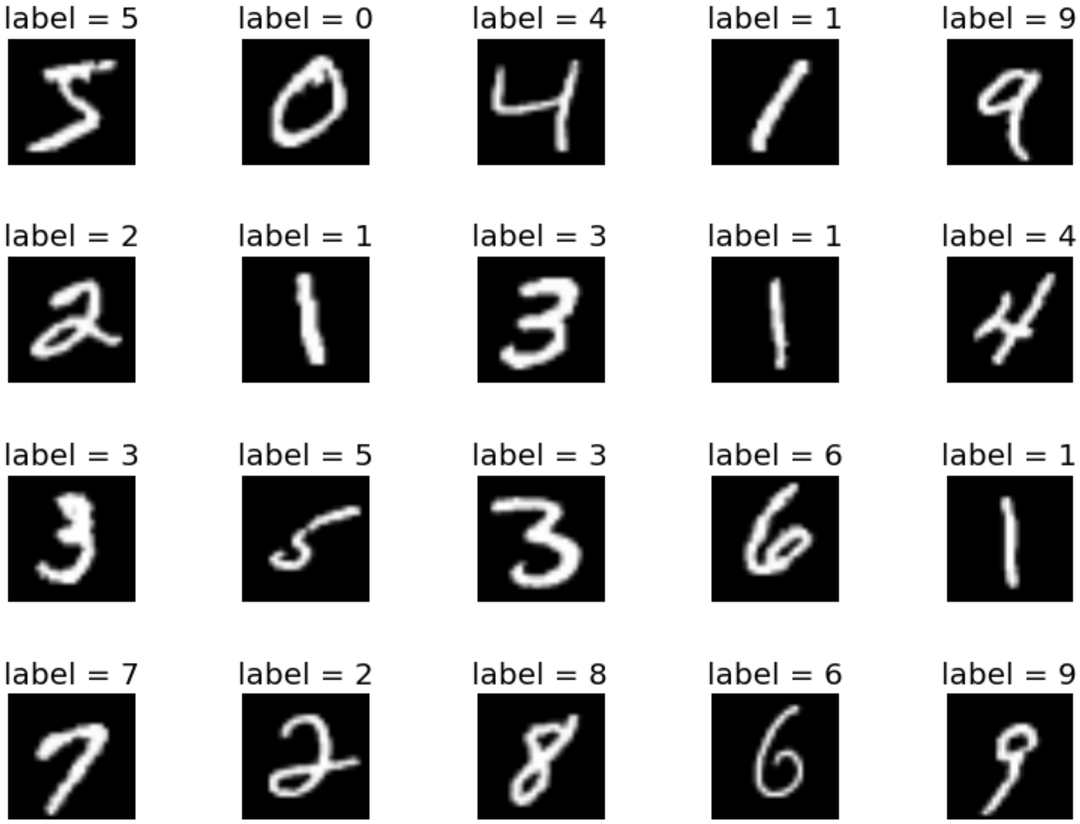


Figure 1. MNIST Dataset

Neural Network Architecture

The neural network consists of layers that are connected with each other by some activation function, following that logic our model has 3 major classes - the Layer class, the Activation function classes and the Neural Network class which combined all together. The class later receives input and output sizes for each layer. It has the forward and backward methods that receive a single layer of NN, and do the one-step forward/backward operation and return the next/previous layer. Of course for doing a one-step pass or back propagation of one step, the activation functions and their derivatives are needed. That's where the classes of activation functions become handy. There are 4 main activation functions suggested: ReLU, Leaky ReLU, Sigmoid, and, of course, Softmax that's applied at the output layer. All classes have their corresponding regular functional forms and ones for their derivatives as well. At the end, class NeuralNetwork receives the instances of layers and activations that should be applied on them to transition from layer to layer. The class combines the one-step passes into a full pass through the whole Neural Network and returns a model with optimized weights and biases that were predefined using standard normal distribution.

1. Input Layer:

The input layer serves as the entry point for data into the our network. In our case, it handles images of handwritten digits. Each image from the MNIST dataset is 28x28 pixels, totaling 784 pixels. So, the input to the neural network is a matrix of size 28x28. These images are flattened into a one dimensional array of 784 features (each representing one pixel), which is fed into the network.

2. Hidden Layers:

In this project, users have the flexibility to experiment with various parameters such as the number and size of hidden layers and their corresponding activation functions in the neural network to optimize performance. By allowing users to adjust these parameters, they can discover the most effective configurations for recognizing handwritten digits. Through experimentation, we found that a layer size of 128 neurons using the ReLU activation function provides the best results. As another instance, a configuration with two hidden layers of 128 and 64 neurons using ReLU and Sigmoid activations performed poorly in predicting the digit '3' in new, externally provided data. However, when we simplified the architecture to just one hidden layer with 128 neurons and used the ReLU activation, the network achieved significantly higher accuracy and lower loss, leading to much better overall performance on the unseen data. As discussed , each layer uses an activation function for a transition to another layer that allows the network to learn complex patterns. We provide a few options for the user for selecting activation functions, let us look at them individually.

- **ReLU (Rectified Linear Unit)**: This is the most common activation function, defined as **f(x) = max(0, x)**. It introduces non-linearity while being computationally efficient, which helps in training deep neural networks effectively.
- **Leaky ReLU**: It modifies ReLU by allowing a small, non-zero gradient when the unit is not active, defined as **f(x) = x if x > 0 else αx**. This slight adjustment helps keep the neurons active and alleviates the problem of "dying neurons" in ReLU. When choosing Leaky ReLU, the user also provides the alpha parameter which is by default 0.1 .
- **Sigmoid**: The sigmoid function outputs a value between 0 and 1, which can interpret probabilities. It is traditionally used for binary classification but can be part of multi-layer networks to help regulate the outputs as probabilities before the final layer.

User can chose which function to apply as an activation function by just entering the name of the function.

3. Output Layer:

For our digit recognition task, the output layer is where the network makes its predictions, it needs to classify each input into one of 10 categories (digits 0-9). This layer has 10 neurons, each corresponding to a digit. The output of the neural network is a 1x10 vector. Each element of this vector represents the model's predicted probability. It uses the softmax activation function, which is crucial for multi-class classification tasks like ours. The softmax function converts the outputs of the last hidden layer into probabilities by taking the exponential of each output and then normalizing these values by dividing by the sum of all exponentials. This ensures that the output values are between 0 and 1 and sum up to 1, making them a proper probabilistic distribution.

By choosing different numbers and sizes of layers, as well as different activation functions, the user can experiment with the network's structure and its ability to process and learn from the data. Each component—from the size of the hidden layers to the choice of activation function—plays a crucial role in defining how effectively the network can learn and make accurate predictions. This modularity and customization make neural networks very powerful tools for a wide range of tasks in machine learning.

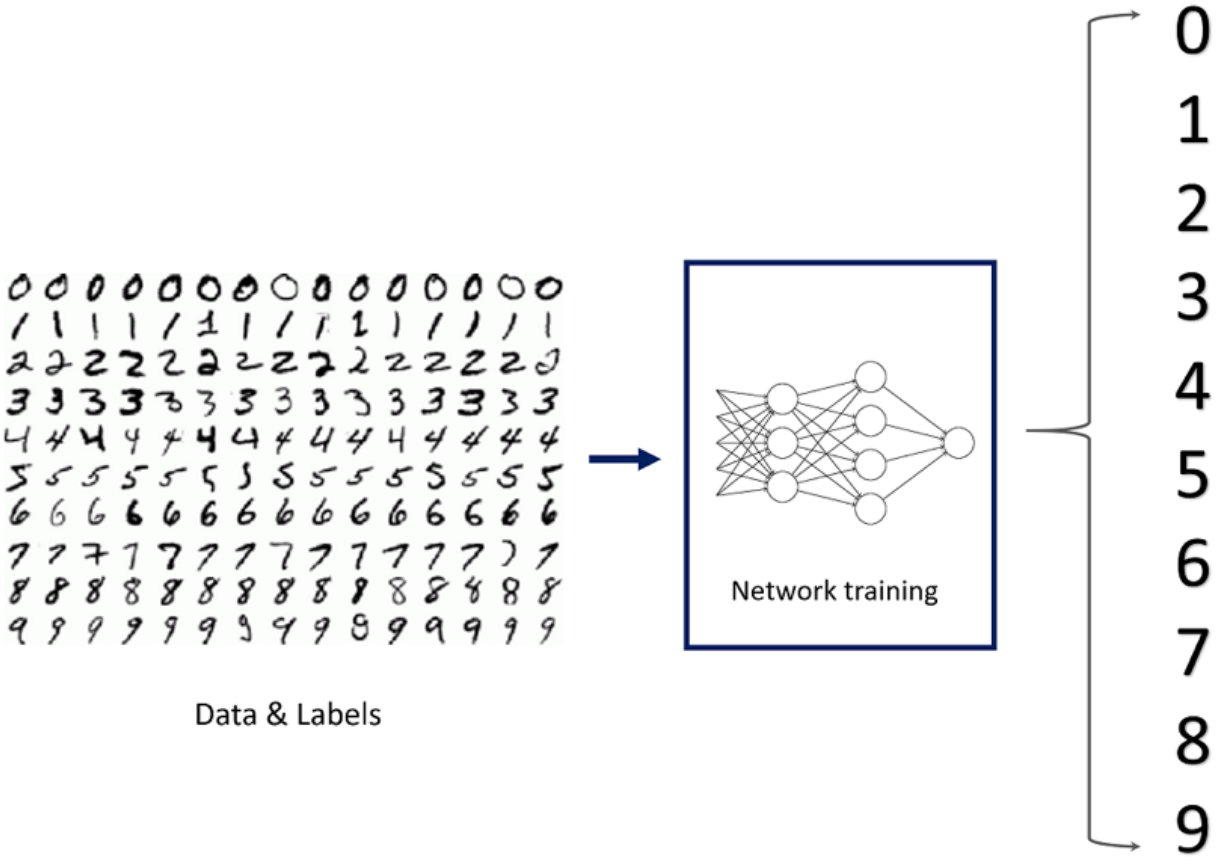


Figure 2. MNIST Dataset and Number Classification

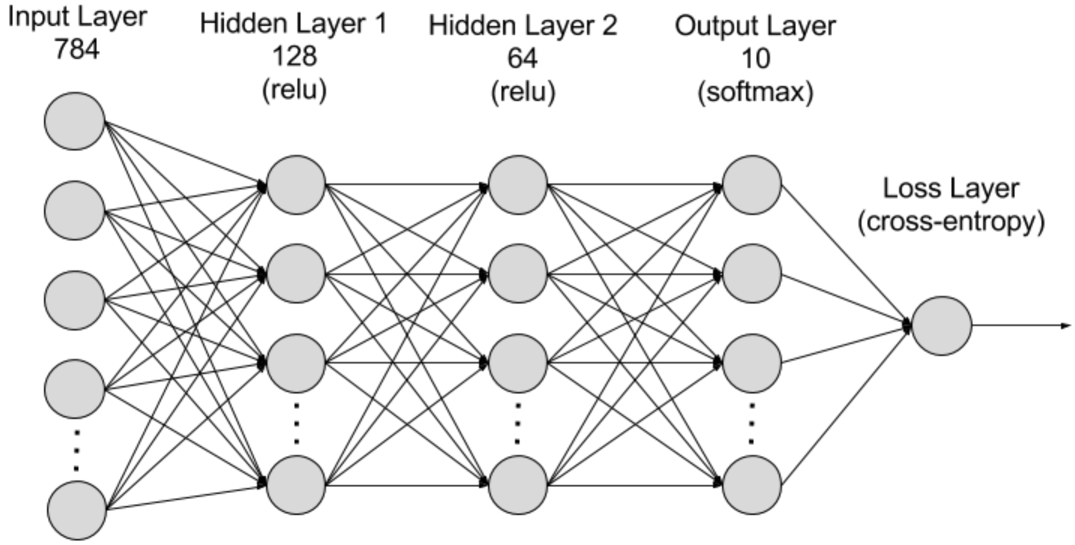


Figure 3. Network architecture for MNIST.

Training Process:

Forward Propagation

The network uses the forward propagation to compute the output.

Forward propagation involves multiple steps across layers starting from the input layer to the output layer. In each layer:

Linear Transformation: The input from the previous layer (or the initial input for the first hidden layer) is transformed using a linear equation:

Z=**XW**+**b**, where:

X is the input matrix (28×28 for an image reshaped into a single vector),

W represents the weights matrix (784×*n* as there are 784 input features, and *n* neurons in that particular layer.),

b is the bias vector (matches the number of neurons *n* in the layer to which it is applied),

Z is the result of the linear transformation (n×batch size=128).

Activation Function: The linearly transformed data **Z** is then passed through an activation function *σ*, such as ReLU or Sigmoid. This step introduces non-linearity into the model, enabling it to learn more complex patterns:

$\mathbf{A}=\sigma(\mathbf{Z})$, serves as the output of the current layer and the input to the next layer.
This process continues through all layers until the output layer, which uses a Softmax function to convert the outputs to probability distributions over predicted classes.

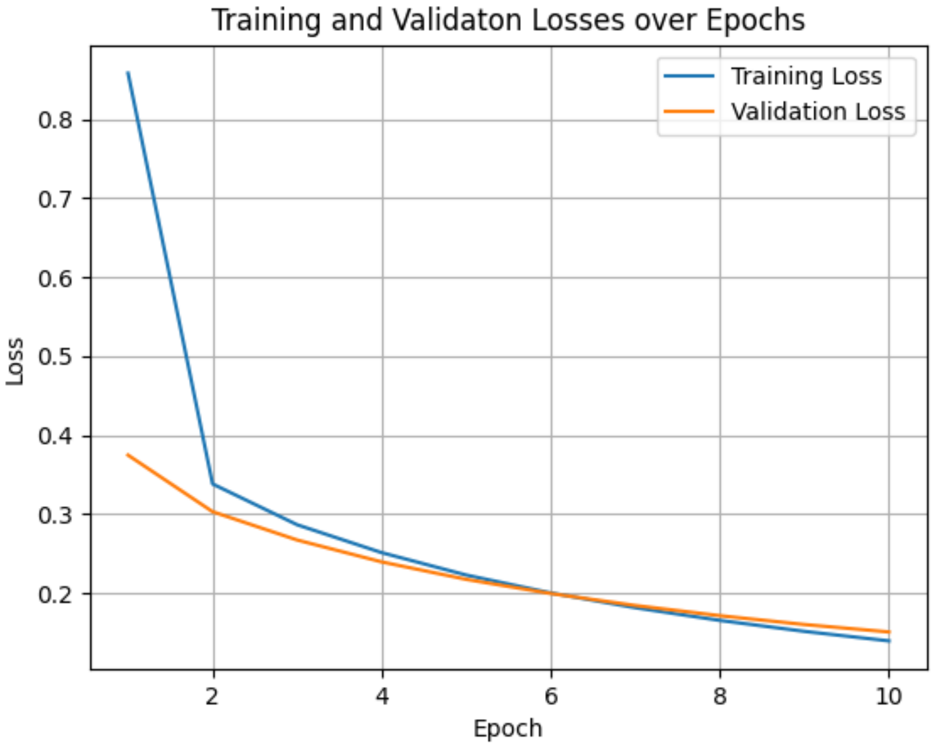
Loss Computation (Cross-Entropy)

In this project, we used cross-entropy loss to evaluate the performance of our neural network in the classification of handwritten digits. Cross-entropy loss is particularly suited for classification tasks where the model's outputs are probabilities ranging between 0 and 1. We used cross-entropy as it particularly effective for multi-class classification tasks, as in the case of our digit recognition task.

Backward Propagation

To improve the model, we used backward propagation to update the weights and biases. This involves computing the gradient of the loss function with respect to each parameter in the network:
Gradient of Loss: The derivative of the loss function is calculated with respect to each output from the last layer. This gradient will guide how much and in what direction the weights and biases should be adjusted.
Chain Rule Application: Using the chain rule, these gradients are backpropagated through the network by iteratively calculating the gradients with respect to each layer's weights and biases.
Parameter Update: Once the gradients are calculated, the weights and biases are updated using these gradients and a learning rate $\alpha=0.01$.

The cycle of forward propagation, loss computation, and backward propagation is repeated for multiple iterations (epochs=10) over the training dataset. Each iteration aims to reduce the overall loss, thus incrementally improving the model's predictions. As the model trains, we typically monitor both the training loss and validation loss to ensure that the model is not just memorizing the training data but also generalizing well to new, unseen data.



Model Evaluation on Test Data:

After training the neural network, we need to assess how well it performs on unseen data, which is crucial for understanding its effectiveness in real-world applications. This is done using a test set, which the model hasn't seen during training

- We use the test dataset to measure the model's accuracy. This tells us the percentage of test images that were correctly classified by the model.

- Accuracy is calculated by comparing the predicted labels against the true labels from the test set.

- We also track loss during training and validation. The loss function measures how well the model's predictions align with the true labels, with lower values indicating better performance.

Comparing Custom Model with Keras Model:

- We implement a similar neural network using the high-level Keras API to establish a benchmark for our custom model.

- The Keras model is also trained and evaluated on the same MNIST dataset to ensure a fair comparison.

- After training both models, we compare their accuracies to see which performs better on the test set.

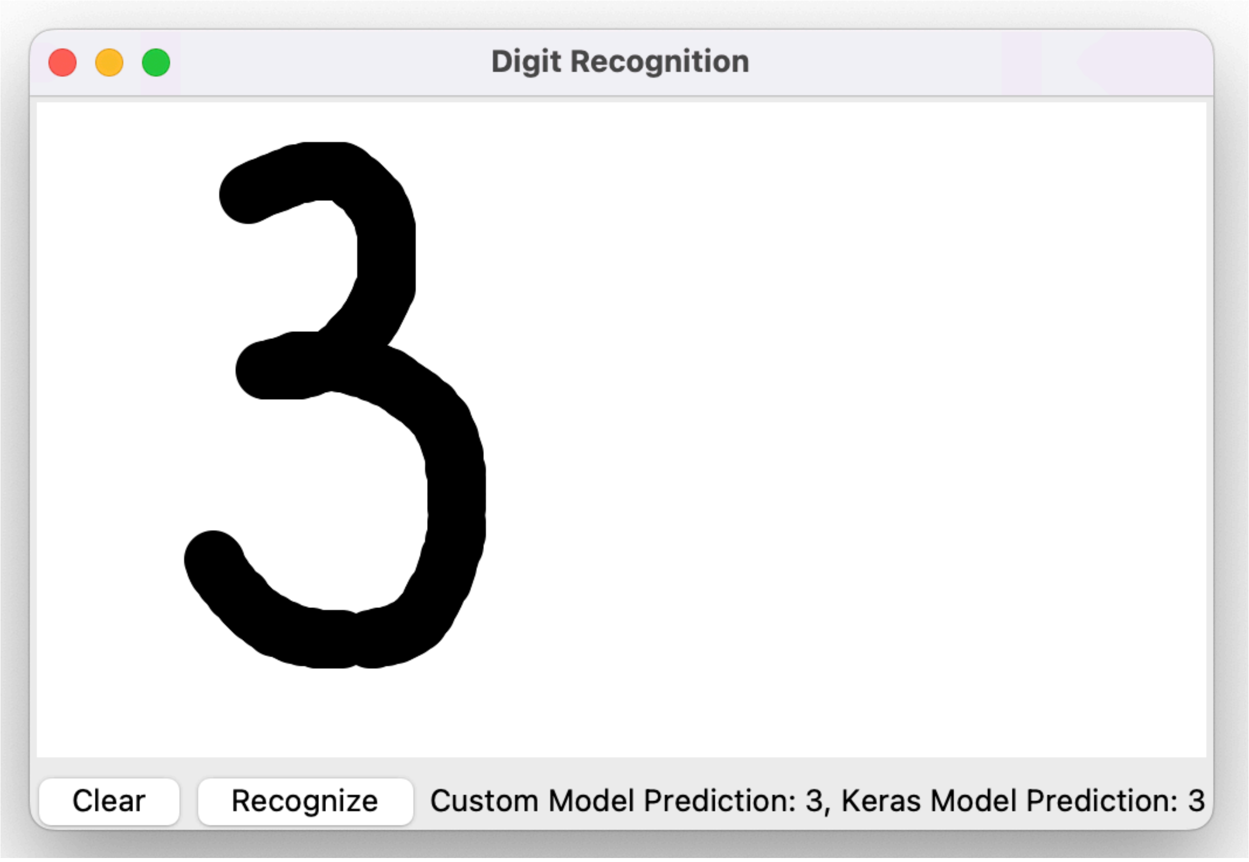
Interactive Drawing Application for Real-Time Testing:

To demonstrate the practical application of our model, we develop an interactive GUI where users can draw digits, which are then recognized by the model.
This is implemented using the **Tkinter** library in Python, where drawn images are captured, preprocessed, and fed into the model for prediction.

Key Functions for the Drawing App:

paint: Handles the drawing on the canvas.
clear_canvas: Clears the drawing canvas to allow new drawings.
save_image: Processes the drawn image, predicts the digit using the neural network, and displays the prediction.
load_and_preprocess_image: Prepares the raw images from the canvas for the neural network by resizing and normalizing them.

This comprehensive approach not only evaluates the model's performance quantitatively but also qualitatively through real-time interaction. Such testing provides insights into how the model performs under practical conditions, which is essential for applications in image recognition tasks.



Custom Model Prediction: 3, Keras Model Prediction: 3

Figure 4. Usage example of drawing app for number 3

Conclusion

After conducting a thorough evaluation and comparison of the custom-built neural network and a standard Keras model, we have insightful outcomes to consider. Our custom model achieved an impressive accuracy of 96.50% on the MNIST test set with one hidden layer of size 128 and ReLU activation function. This demonstrates a robust performance, particularly considering that the model was built from scratch using numpy arrays for operations.

On the other hand, the Keras model, which leverages TensorFlow's optimized backend operations, achieved a slightly higher accuracy of 96.85%. This increment, while modest, showcases the benefits of using a high-level framework like Keras, which includes optimized algorithms that can lead to better performance on some tasks without the need for deep customization.