Solving the MNIST Dataset From Scratch

In Python and NumPy - A Full-Stack Deep Learning Project

Author: Andrew Fief
Date : May 21, 2025
https://github.com/andyfief/MNIST-from-scratch

Introduction

This project demonstrates a complete handwritten digit recognition system built from the ground up using only Python and NumPy. The goal was to implement a feedforward neural network for the MNIST dataset without relying on any machine learning libraries like TensorFlow or PyTorch. After creating the model architecture, preprocessing the input data, and training the model on a 60,000/10,000 train/test split, the model achieved 92% accuracy. The system was designed not only to train and test a neural network from scratch, but also to provide an interface that serves predictions via an API, complete with a React-based front end and containerized deployment using Docker. By focusing on minimal dependencies, this project was an exercise in fundamental deep learning principles, full-stack development, and DevOps.

The MNIST Dataset

The MNIST (Modified National Institute of Standards and Technology) dataset is a classic benchmark in the field of machine learning and computer vision. It consists of 70,000 grayscale images of handwritten digits: 60,000 for training and 10,000 for testing, where each is a 28 by 28 pixel grayscale image. Each image is labeled with the digit it represents, ranging from 0 to 9. Despite its simplicity, MNIST remains a widely used dataset for testing image classification models and validating new learning algorithms.

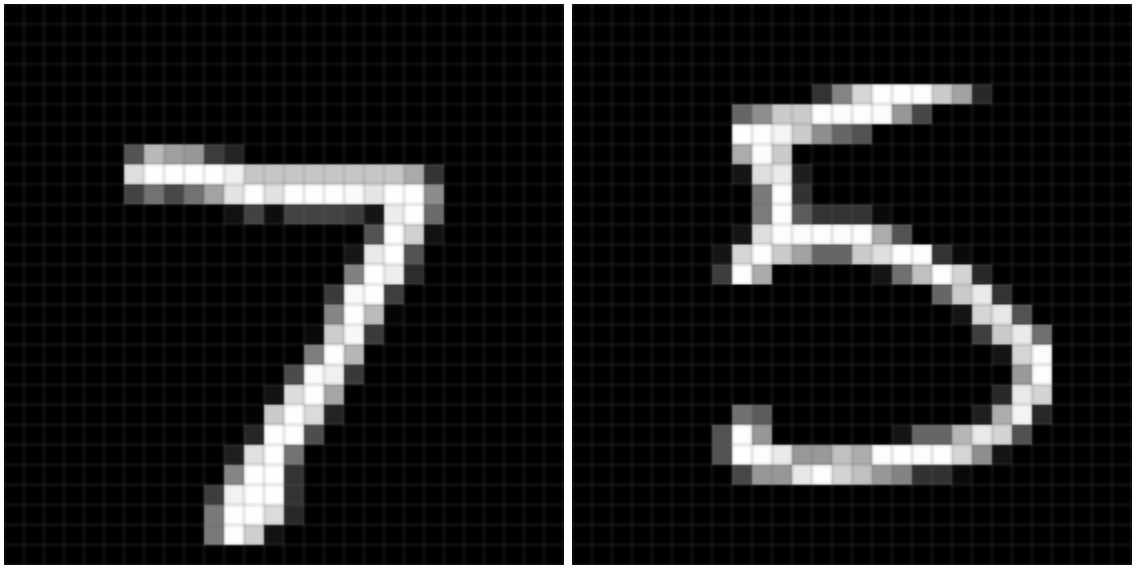

Figure 1: Two raw MNIST digit samples

Preprocessing

To improve inference quality and maintain consistency between training and prediction phases, preprocessing plays a key role. Firstly, each image is binarized (pixels are converted strictly to black or white). Next, the image is centered. This is done using an algorithm that detects the edges of an image, calculates its center, and calculates the difference between the image center and the center of the 28x28 pixel grid. The image is centered by creating a blank image, selecting the region of the original image that contains the number, and copying it into the new image, starting at the center position minus the offset. Finally, to guide the network's focus on the shape of the image, the edges are greyed.

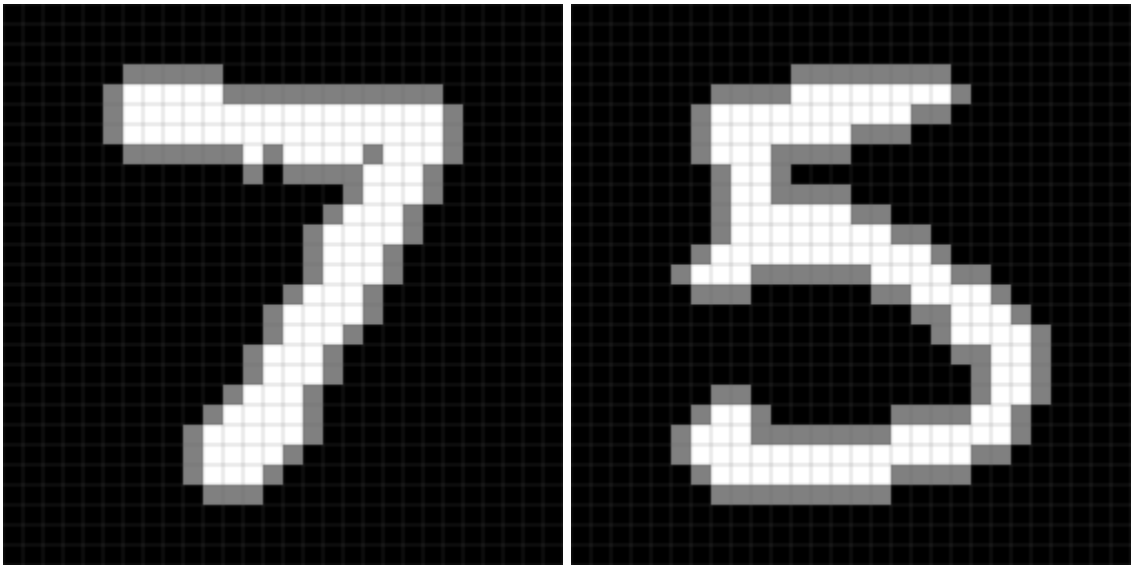

Figure 2: Two cleaned MNIST digit samples

The same preprocessing pipeline is applied to user-drawn digits in the browser to ensure consistency with the training data.

The Model

The core of the project is a deep neural network (DNN) implemented from scratch. The model uses a three-layer architecture: an input layer of 784 neurons (one per pixel), two hidden layers with 128 and 64 neurons respectively, and an output layer of 10 neurons, each representing a digit.

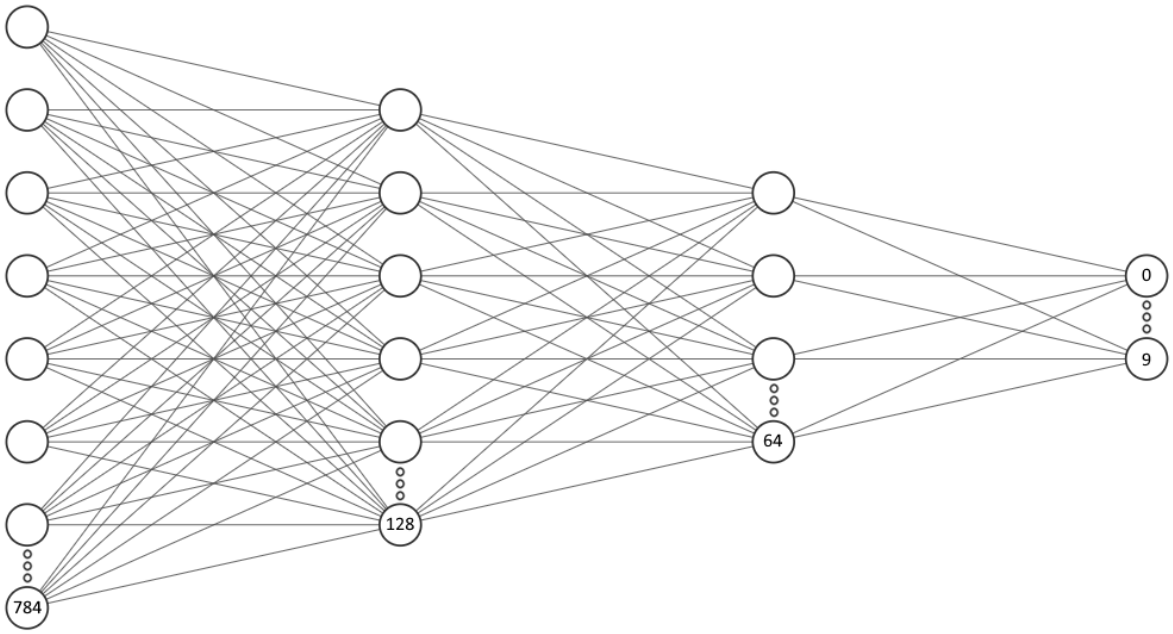

Figure 3: Representation of Neural Network Architecture

After creating the layers, each is initialized with random weights that are scaled down by the inverse square root of the number of neurons in the receiving layer. This scalar factor reduces the variance of the weights proportional to the size of the layer, helping to prevent activations from becoming too large or too small.

$$\text{weight scalar} = \sqrt{\frac{1}{layer\ size}}$$

Figure 4: Weight scalar equation

The training process of the neural network begins when a preprocessed image (flattened into a 1-dimensional input vector) is fed into the model. The input is passed through connected layers, each applying learned weights and non-linear activation functions. As the input moves forward through the network, it's processed layer by layer until it reaches the output layer, which produces a probability distribution over the ten digit classes (0 through 9). This is the **forward pass**, where the model makes its best guess based on its current understanding. To improve, the model then compares this prediction to the actual label and calculates the error. During the **backward pass**, this error is propagated backward through the network, layer by layer, adjusting the weights in a direction that reduces future errors. These updates are small and incremental, repeated across many images over 10 training cycles called "epochs". With each epoch, the model gradually improves its accuracy, learning which patterns of pixel intensities correspond to which digits. After 10 epochs, the model sees very little or no improvement.

Activation functions introduce non-linearity into a neural network, allowing it to learn and model complex patterns in data. Without them, the entire network would behave like a simple linear transformation, regardless of how many layers it has. By applying functions at different stages of

the network, we enable it to make distinctions between inputs. Put simply, activation functions give neural networks their power to learn beyond basic arithmetic.

The first activation function used in the hidden layers is the **sigmoid** function. This function sets values to be between 0 and 1, where large negative numbers are close to 0 and large positive numbers are close to 1, resulting in an S-shaped curve. The sigmoid function's curve introduces non-linearity that helps the model learn complex relationships.

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

Figure 5: The sigmoid activation function.

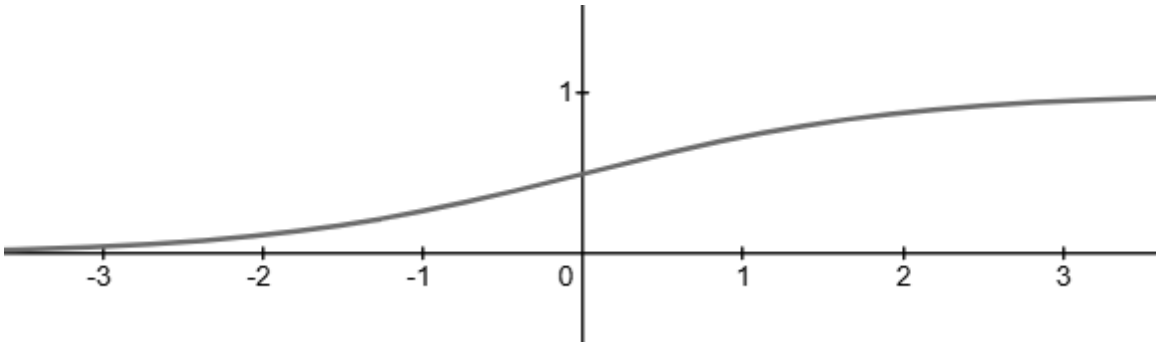

Figure 6: The sigmoid activation function graph.

The second activation function used is the **softmax** function. Softmax is used between the last hidden layer and the output layer, where it transforms the output vector of length 10 into a probability distribution, where each value represents the predicted probability of a corresponding digit class (0 through 9).

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

Figure 7: The softmax activation function.

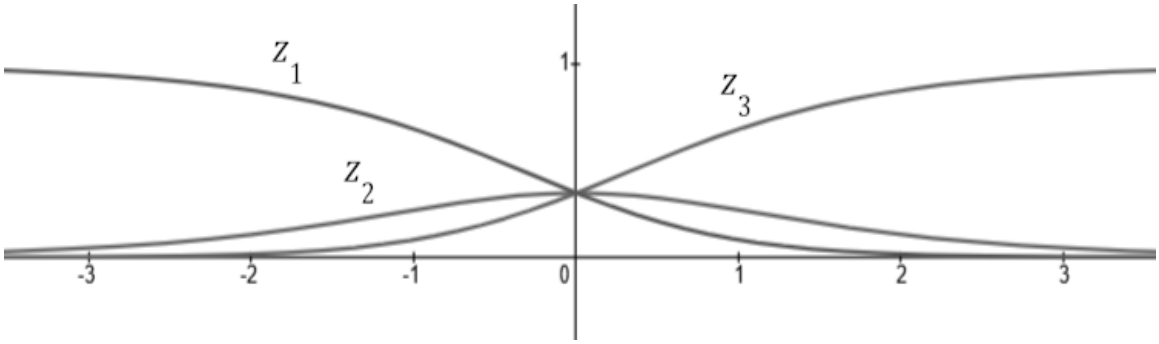

Figure 8: A so**f**tmax activation graph for 3 lines, representing a vector z of length 3 $[z_1, z_2, z_3]$ where all indices of z sum to 1. At x = 0, each of these lines intersects at y = 1/3. As a single line (or vetor index) increases, the others approach zero.

Gradient descent is the method used to iteratively improve the neural network's parameters to better match its predictions to the expected outputs, called targets. The targets represent the desired outputs for each input example, initialized close to zero (0.01) for all classes except the correct class, which is set near one (0.99), to create a low-error goal for the model's output. The loss function quantifies the difference between the network's predicted output and these targets, measuring how far off the model is. To reduce this difference, we compute gradients that describe how sensitive the loss is to each weight in the network. These gradients point in the direction where the loss increases most steeply, so by adjusting the weights in the opposite direction, the network's predictions gradually shift closer to the targets.

This is done by calculating the error at the output layer as the difference between predictions and targets, then propagating this error backward through the network layers by applying the chain rule of calculus. This is called **Backpropagation**. Each backward step computes how changes in weights influence the loss, allowing us to update weights proportionally to their contribution to the error. By repeating this process over many training examples and iterations, all parts of the network contribute to reducing the overall prediction error, guiding the model to converge on weights that generate outputs closely aligned with the target values and improve prediction accuracy.

Together, forward propagation, loss computation, backpropagation, and weight updates form the complete training loop that allows the network to learn from data. After 10 epochs of training, the model reaches a testing accuracy of 92.37%, meaning 9237/10,000 testing images were guessed correctly. The trained weights are saved using Python's pickle module to enable easy loading during inference.

The Front End

The front end of the project is built using **React**, providing a dynamic and responsive user interface where users can draw digits directly in the browser. At the core of the interface is an HTML canvas element, which allows freehand drawing and captures user input as pixel data. The canvas is configured as a 28x28 pixel grid, matching the format of MNIST dataset images, so that what the user draws can be processed in the same way as training data. Once the user presses the predict button, the canvas image is converted to a base64-encoded string, which

compactly represents the raw pixel data. This string is then sent via an API request to the back end.

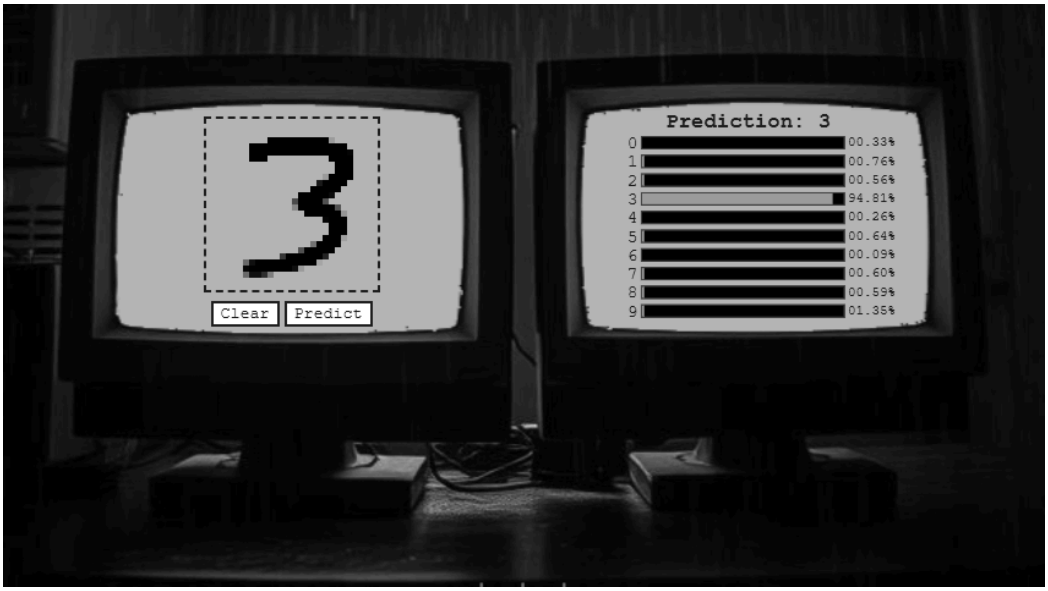

Figure 9: User Interface for digit drawing and prediction results.

The Back End

The back end is implemented with **Flask**. The front end communicates with the back end using CORS (Cross-Origin Resource Sharing) to allow cross-origin HTTP requests. When the image data reaches the backend, it's decoded, preprocessed in the same fashion as the dataset, and sent through the neural network in a forward pass, ultimately returning the model's prediction: the probability distribution. This distribution is then displayed on the front end, giving users immediate feedback on the model's performance and confidence levels.

Deployment

The project is deployed using **Docker** and **Docker Compose**, creating separate containers for the React frontend and the Flask backend. Each container is built from its own image, ensuring all dependencies are included. Docker Compose manages these containers, handling networking and startup so the frontend can easily communicate with the backend API. This setup simplifies deployment and makes the application easy to run on any system.