# Report on Core ML

Sheil Maniar

June 2025

## 1 Outline

This is my report on the MNIST Digit Classification Project as a part of Devsoc Core Assignment 2025-26.
The MNIST dataset is a large database of handwritten digits commonly used for training and testing image processing systems, particularly in the field of machine learning. It contains 60,000 training images and 10,000 testing images (all grayscale) of digits from 0 to 9, each of which is a 28x28 pixel grayscale image.Each image is flattened into a 784-dimensional vector and the labels are one-hot encoded for classification.
The project was to build a neural network from the ground up without any external libraries like tenserflow. We had to create custom neural layers, writing forward and backward propagation manually and then perform gradient descent all while using features like activation functions, loss functions and random weight and bias initialization.

## 2 Mathematics behind this

Forward Propagation :

Forward propagation is the process where input data is passed through the neural network, layer by layer, to produce an output. Each neuron in a layer takes the outputs from the previous layer, applies a weighted sum, adds a bias, and then passes the result through an activation function.

Linear Transformation: For each layer, the input is transformed linearly by multiplying it with the layer's weights ($W$) and adding the bias ($B$).

$$Z = W \cdot X + B$$

Activation Function: The result of the linear transformation ($Z$) is then passed through an activation function ($g$). This introduces non-linearity into the model, allowing it to learn more complex patterns.

$$A = g(Z)$$

Where:

$A$ is the output of the activation function (and the input to the next layer). $g$ is the activation function (e.g., ReLU, sigmoid, softmax). In my code, there are 3 layers in total (2 activation functions used). The first layer uses the ReLU activation function, and the second layer uses the softmax activation function.

Layer 1:

$$Z_1 = W_1 \cdot X + B_1$$

$$A_1 = \text{ReLU}(Z_1) = \max(0, Z_1)$$

Layer 2:

$$Z_2 = W_2 \cdot A_1 + B_2$$

$$A_2 = \text{softmax}(Z_2) = \frac{e^{Z_2}}{\sum e^{Z_2}}$$

Backward Propagation

It involves calculating the gradient of the loss function with respect to the weights and biases. This gradient indicates how much the loss changes with respect to changes in the parameters. The difference between the predicted output ($A_2$) and the actual target values ($Y$) is calculated. In my code, the targeted values are converted to a one-hot encoded format.

$$dZ_2 = A_2 - Y_{\text{one-hot}}$$

Where $Y_{\text{one-hot}}$ is the one-hot encoded version of $Y$.

The gradients for the weights and biases of the output layer $(W_2, B_2)$ are calculated based on the error and the output of the previous layer $(A_1)$.

$$dW_2 = \frac{1}{m} dZ_2 \cdot A_1^T$$

$$dB_2 = \frac{1}{m} \sum dZ_2$$

Where $m$ is the number of training examples.

The error is propagated back to the hidden layer $(A_1)$ using the weights of the output layer $(W_2)$. The gradient for the hidden layer's linear transformation $(Z_1)$ is calculated, taking into account the derivative of the activation function (ReLU in this case).

$$dZ_1 = W_2^T \cdot dZ_2 \odot g'(Z_1)$$

Where $\odot$ denotes element-wise multiplication, and $g'(Z_1)$ is the derivative of the ReLU function, which is 1 for $Z_1 > 0$ and 0 otherwise.

Finally, the gradients for the weights and biases of the hidden layer $(W_1, B_1 W_1, B_1)$ are calculated.

Gradient Descent Gradient descent is an optimization algorithm used to minimize the loss function. It iteratively updates the weights and biases in the direction opposite to the gradient, which is the direction of steepest ascent. By moving in the opposite direction, we move towards the minimum of the loss function.

The weights and biases are updated using the calculated gradients and a learning rate $(\alpha\alpha)$. The learning rate controls the size of the steps taken in the direction of the negative gradient.

$$W \leftarrow W - \alpha \cdot dW$$

$$B \leftarrow B - \alpha \cdot dB$$

This process is repeated for a specified number of iterations or until the loss converges to a minimum.

## 3 The Code

### 3.1 Neural Architecture

The neural network built has

| **Layer** | **Configuration (Dimensions and Activation)** |
|---|---|
| Input Layer | $784 \times 1$ (Input image pixels) |
| Hidden Layer 1 | $10 \times 784$ (Weights), $10 \times 1$ (Bias), ReLU Activation |
| Output Layer | $10 \times 10$ (Weights), $10 \times 1$ (Bias), Softmax Activation |

### 3.2 Loss function used

The loss function used in this neural network is the Cross-Entropy loss. It is commonly used for multi-class classification problems.

#### Formula

The formula for the cross-entropy loss for a single sample is:

$$L = -\sum_{c=1}^{C} y_c \log(\hat{y}_c)$$

Where:

- $C$ is the number of classes.
- $y_c$ is a binary indicator (0 or 1) if class $c$ is the correct classification for the sample.

- $\hat{y}_c$ is the predicted probability of class $c$ by the model.

For a batch of $m$ samples, the average cross-entropy loss is:

$$J = -\frac{1}{m} \sum_{i=1}^{m} \sum_{c=1}^{C} y_{i,c} \log(\hat{y}_{i,c})$$

### Benefits

- **Suitable for classification:** It is particularly well-suited for classification tasks, especially when using a Softmax activation in the output layer.
- **Provides a good gradient for optimization:** The gradient of the cross-entropy loss with respect to the output layer's activations is well-behaved, which helps in the training process using gradient descent.
- **Penalizes confident incorrect predictions:** It heavily penalizes the model when it is highly confident about an incorrect prediction.

## 3.3 Training the model

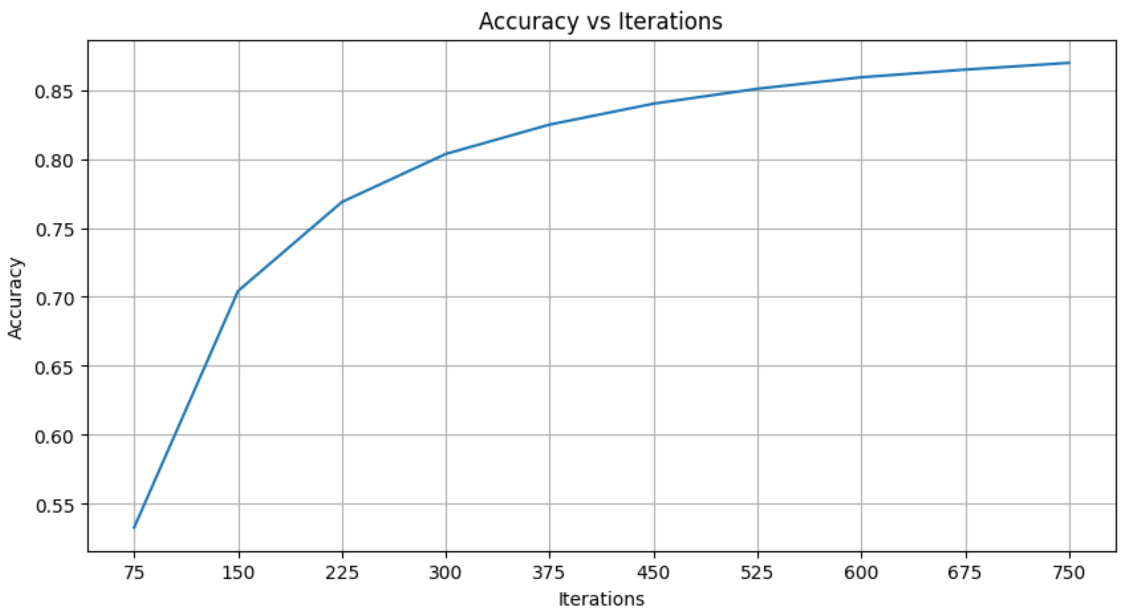
- **Number of iterations** : 750
- **Learning rate** : 0.1
- **Activation functions used** : ReLU and Softmax
- One hot encoding used

## 3.4 Results

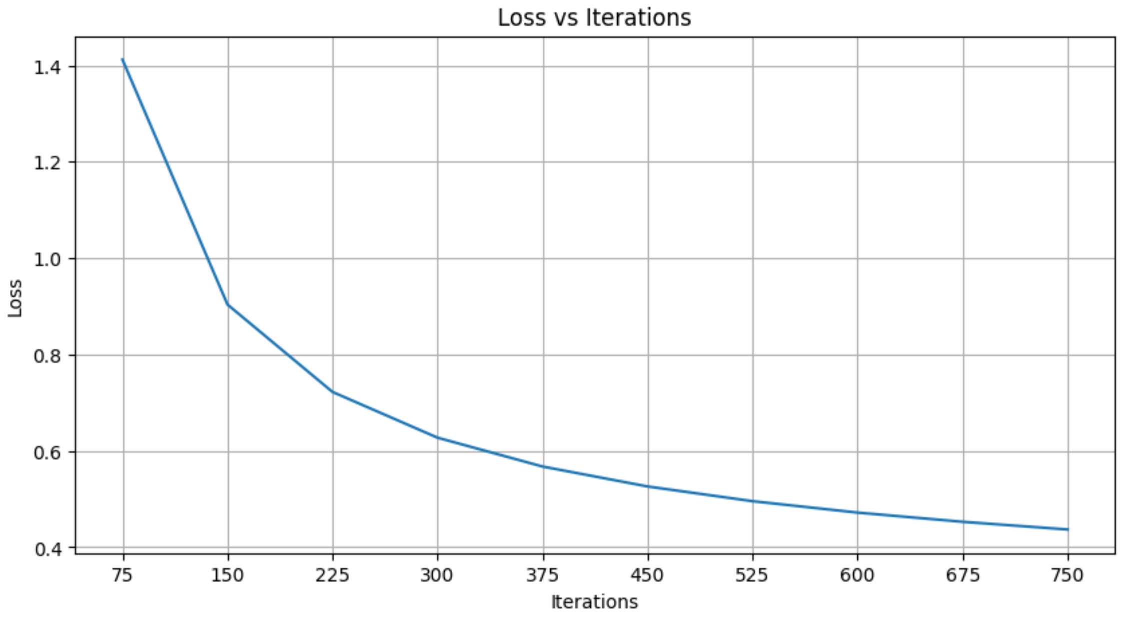
| Serial Number | Iteration Number | Loss | Accuracy |
|---|---|---|---|
| 1 | 75 | 1.412 | 0.533 |
| 2 | 150 | 0.904 | 0.704 |
| 3 | 225 | 0.722 | 0.769 |
| 4 | 300 | 0.627 | 0.804 |
| 5 | 375 | 0.567 | 0.825 |
| 6 | 450 | 0.526 | 0.840 |
| 7 | 525 | 0.495 | 0.851 |
| 8 | 600 | 0.472 | 0.859 |
| 9 | 675 | 0.453 | 0.865 |
| 10 | 750 | 0.437 | 0.870 |

**Final results : 0.437 Loss at 87% accuracy**

.



Accuracy Plot



Loss Plot