**Submitted by:**

**ABDUL MOEED**

**TASK#3**

Create a neural network from scratch using Python and NumPy to classify handwritten digits from the MNIST dataset.

**Introduction**

In this tutorial, we will build a neural network from scratch using Python and NumPy to classify handwritten digits from the MNIST dataset. The MNIST dataset consists of 70,000 grayscale images of handwritten digits (0-9), each of size 28x28 pixels. We will create a simple neural network with one hidden layer, train it on the dataset, and evaluate its performance. This process will help you understand the fundamentals of neural networks, including data preprocessing, parameter initialization, forward and backward propagation, and model evaluation.

**Step 1: Load the MNIST Dataset**

We begin by loading the MNIST dataset. The tensorflow.keras.datasets module provides a convenient way to download and load this dataset. The dataset is split into a training set and a test set. We then normalize the pixel values of the images to be between 0 and 1 to help the neural network learn more effectively.

**Code Snip:**

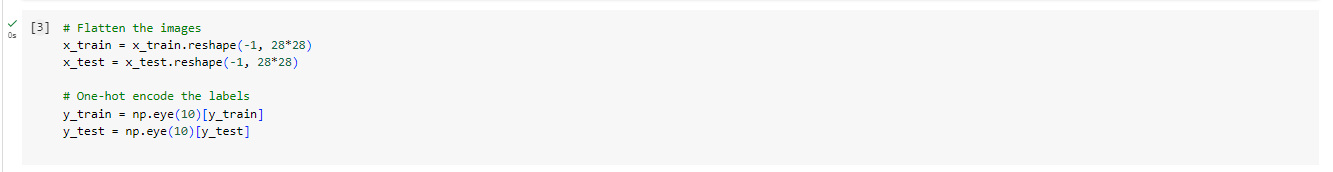


**Step 2: Preprocess the Data**

Once the data is loaded, we need to preprocess it. This involves two main steps:

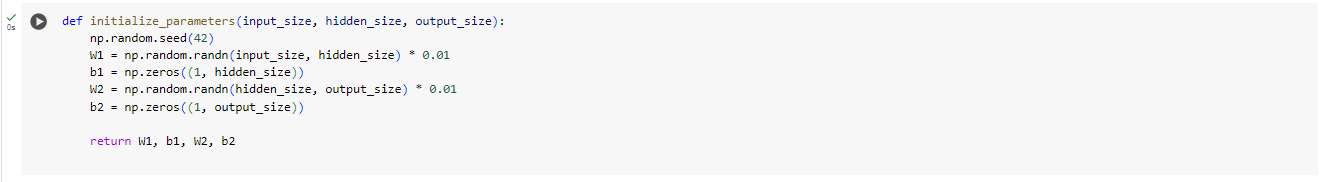
1. **Flattening the Images:** The images are originally 28x28 matrices, but the neural network expects a 1D input vector. We reshape each image into a vector of size 784 (28\*28).
2. **One-Hot Encoding the Labels:** The labels are originally integers representing the digit (0-9). One-hot encoding converts these labels into vectors of length 10, where the position corresponding to the digit is set to 1, and all other positions are set to 0. This format is required for the output layer of the neural network.

**Code Snip:**



**Step 3: Initialize the Neural Network Parameters**

We initialize the parameters of the neural network, which include weights and biases. The network consists of one input layer, one hidden layer, and one output layer. The weights are initialized randomly with small values, and the biases are initialized to zero. This step is crucial because the initial values can significantly impact the training process and the final performance of the model.



**Step 4: Define Forward and Backward Propagation**

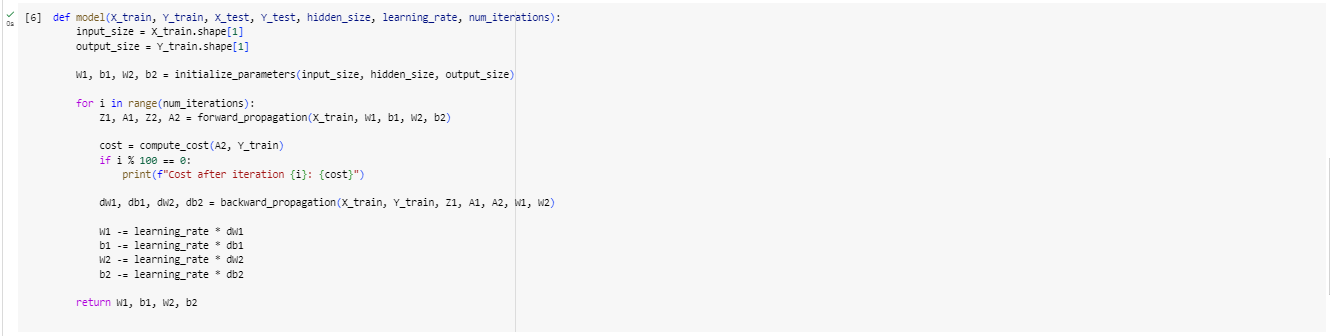
**Forward Propagation:** This step involves calculating the activations of the neurons in each layer of the network. We use the sigmoid function as the activation function for the hidden layer and the softmax function for the output layer. The sigmoid function introduces non-linearity, while the softmax function converts the output into probabilities.

**Backward Propagation:** After computing the output, we calculate the error and propagate it back through the network to update the weights and biases. This involves computing the gradients of the loss function with respect to the weights and biases using the chain rule. The sigmoid derivative is used to calculate the gradients in the hidden layer.

# 

**Step 5: Implement the Training Loop**

The training loop iterates over the dataset multiple times (epochs). In each iteration, we perform forward propagation to get the predictions, compute the cost (error), and then perform backward propagation to update the parameters. We use gradient descent to adjust the weights and biases based on the computed gradients. The learning rate determines the step size of the updates. We also print the cost every 100 iterations to monitor the training process.



**Step 6: Evaluate the Model**

After training the model, we evaluate its performance on the test set. We make predictions by performing forward propagation on the test data and then calculate the accuracy by comparing the predicted labels with the true labels. Accuracy is the proportion of correct predictions out of the total number of predictions. This step helps us understand how well the model generalizes to new, unseen data.

.