**Assignment 2**

**Implementing Feedforward Neural Networks with Keras and TensorFlow**

**Step 1: Importing Required Libraries**

*import tensorflow as tf*

*from tensorflow import keras*

*import pandas as pd*

*import numpy as np*

*import matplotlib.pyplot as plt*

*import random*

*%matplotlib inline*

**TensorFlow and Keras**: Used for building and training the neural network.

**Pandas**: Useful for data manipulation and analysis.

**NumPy**: Used for numerical operations on arrays.

**Matplotlib**: Helps visualize data and results.

**%matplotlib inline**: Ensures that plots are displayed within the notebook.

**Step 2: Loading and Preparing the MNIST Dataset**

**About MNIST**:  
The **MNIST dataset** contains **70,000 images** of handwritten digits (0–9), each with a **28x28 pixel size** (784 features). It is divided into **60,000 training images** and **10,000 testing images**. Each image’s pixel value ranges from **0 (white)** to **255 (black)**.

*# Load the dataset*

*mnist = tf.keras.datasets.mnist*

*(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()*

**x\_train**: Images used for training the model (60,000 samples).

**y\_train**: Labels corresponding to the training images.

**x\_test**: Images used for testing the model (10,000 samples).

**y\_test**: Labels corresponding to the test images.

Check the Dataset Size and Structure:

*# Size of training and testing datasets*

*len(x\_train) # Output: 60000*

*len(x\_test) # Output: 10000*

*# Shape of the datasets*

*x\_train.shape # (60000, 28, 28)*

*x\_test.shape # (10000, 28, 28)*

*# Display a sample image from the training data*

*plt.matshow(x\_train[0])*

**x\_train[0]**: Shows the pixel values of the first training image.

**plt.matshow**: Displays the first image visually.

**Step 3: Normalizing the Data**

To ensure consistent learning, we **normalize the pixel values** from their original range (0–255) to **0–1**.

*# Normalize pixel values to the range [0, 1]*

*x\_train = x\_train / 255.0*

*x\_test = x\_test / 255.0*

**Why normalization?**  
It helps the network converge faster by ensuring that the input values are small and consistent.

**Step 4: Defining the Neural Network Architecture**

Using **Keras**, we define the structure of our **feedforward neural network**.

*model = keras.Sequential([*

*keras.layers.Flatten(input\_shape=(28, 28)), # Flatten the input image*

*keras.layers.Dense(128, activation='relu'), # Hidden layer with 128 neurons*

*keras.layers.Dense(10, activation='softmax') # Output layer with 10 classes*

*])*

**Sequential model**: This allows us to stack layers sequentially.

**Flatten layer**: Converts the 2D image (28x28) into a 1D vector of size 784.

**Dense layer (Hidden)**: A fully connected layer with **128 neurons** using **ReLU activation**.

**Dense layer (Output)**: The output layer with **10 neurons** (one for each class) using **softmax activation** to generate probabilities.

**Step 5: Compiling the Model**

We configure the model with the **optimizer, loss function, and evaluation metric**.

*model.compile(optimizer='sgd',*

*loss='sparse\_categorical\_crossentropy',*

*metrics=['accuracy'])*

**Optimizer**: Stochastic Gradient Descent (**SGD**) is used for optimization.

**Loss function**: We use **sparse categorical crossentropy** since this is a multi-class classification problem with integer labels.

**Metrics**: We track **accuracy** as the evaluation metric.

**Step 6: Training the Model**

We train the model on the **training data** by specifying the number of **epochs** and **batch size**.

*history = model.fit(x\_train, y\_train, epochs=5, batch\_size=32, validation\_split=0.1)*

**epochs=5**: The model will go through the entire dataset 5 times.

**batch\_size=32**: During each step, 32 samples will be processed.

**validation\_split=0.1**: 10% of the training data is used for validation during training.

**Step 7: Evaluating the Model on Test Data**

After training, we evaluate the model’s performance using the **test dataset**.

*test\_loss, test\_accuracy = model.evaluate(x\_test, y\_test)*

*print(f'Test Accuracy: {test\_accuracy}')*

**model.evaluate()**: Returns the **loss** and **accuracy** on the test data.

**Test accuracy**: Gives an idea of how well the model generalizes to unseen data.

**Step 8: Plotting Training Loss and Accuracy**

We can visualize the **training progress** by plotting the **loss** and **accuracy** over epochs.

*# Plot training loss*

*plt.plot(history.history['loss'], label='Training Loss')*

*plt.plot(history.history['val\_loss'], label='Validation Loss')*

*plt.title('Loss vs Epochs')*

*plt.xlabel('Epochs')*

*plt.ylabel('Loss')*

*plt.legend()*

*plt.show()*

*# Plot training accuracy*

*plt.plot(history.history['accuracy'], label='Training Accuracy')*

*plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')*

*plt.title('Accuracy vs Epochs')*

*plt.xlabel('Epochs')*

*plt.ylabel('Accuracy')*

*plt.legend()*

*plt.show()*

**Training Loss vs. Validation Loss**: Helps identify **overfitting** or **underfitting**.

**Training Accuracy vs. Validation Accuracy**: Shows how well the model is learning over time.