ASSIGNMENT 2: FEED FORWARD NEURAL NETWORK USING KERAS AND TENSORFLOW

A **Feedforward Neural Network (FNN)** is one of the simplest types of artificial neural networks used in deep learning. It processes information in one direction, from input to output, without loops or cycles, making it ideal for supervised learning tasks like classification and regression.

1. Neurons and Layers

- **Neuron (Node)**: The fundamental unit of a neural network, analogous to biological neurons. Each neuron receives inputs, applies a weighted sum, adds a bias, and passes the result through an activation function.
- **Input Layer**: The first layer in the network where data is fed into the model. It does not perform computations; it just holds the input data.
- **Hidden Layers**: Layers between the input and output layers where computations occur. These layers transform inputs into outputs through weights, biases, and activation functions.
- **Output Layer**: The last layer that provides the final predictions or outputs of the model. In a classification task, it outputs the probability distribution of each class.

2. Weights and Biases

- **Weights**: The parameters that connect neurons in one layer to neurons in the next. Weights determine the strength and direction (positive or negative) of the input signal. They are adjusted during training to minimize error.
- **Biases**: Additional parameters added to each neuron to shift the activation function. They help improve model accuracy by allowing neurons to activate even when input values are zero.

4. Feedforward Propagation

- **Definition**: The process by which input data is passed forward through the network to make a prediction. Each layer processes inputs, applies weights and biases, passes the result through an activation function, and sends it to the next layer.

6. Gradient Descent

- **Purpose**: An optimization algorithm that minimizes the loss function by adjusting weights and biases.
- **Types of Gradient Descent**:
    - **Batch Gradient Descent**: Uses the entire dataset to compute gradients in each step.
    - **Stochastic Gradient Descent (SGD)**: Uses one data sample at a time, making it faster but noisier.
    - **Mini-batch Gradient Descent**: Compromise between Batch and SGD, uses small batches of data.

8. Epochs and Iterations

- **Epoch**: One complete pass through the entire training dataset.
- **Iteration**: A single update of weights and biases, typically occurring after a batch of data has been processed.
- **Batch Size**: The number of training samples processed before the model updates its parameters.

CODE:

```
[2]  # Implementing feedforward neural networks with Keras and TensorFlow
     # import the necessary packages
     import numpy as np
     import tensorflow as tf
     from tensorflow.keras.datasets import mnist
     from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import Dense
     import matplotlib.pyplot as plt
```

This code demonstrates how to implement a feedforward neural network using Keras and TensorFlow to classify digits in the MNIST dataset. The network has multiple fully connected (dense) layers with ReLU activations and is trained using the SGD optimizer.

Numpy (np): Useful for handling data and performing mathematical operations.

TensorFlow and Keras modules:

- **mnist**: Provides the MNIST dataset, a standard for digit classification.
- **Sequential:** A model type in Keras where layers are stacked sequentially.
- **Dense:** A fully connected (feedforward) layer.

Matplotlib: Used for visualizing training history (loss and accuracy over epochs).

```
print("[INFO] accessing MNIST...")
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.reshape((x_train.shape[0], -1)).astype('float32') / 255
x_test = x_test.reshape((x_test.shape[0], -1)).astype('float32') / 255
```

 Reshape: Converts each image (28x28 pixels) into a single 784-dimensional vector so that it can be input into the neural network.

 Normalization: Scales pixel values from [0, 255] to [0, 1] by dividing by 255, which helps with training efficiency and stability.

```
# One-hot encode the labels
y_train = tf.keras.utils.to_categorical(y_train, num_classes=10)
y_test = tf.keras.utils.to_categorical(y_test, num_classes=10)
```

One Hot Encoding is a method for converting categorical variables into a binary format. It creates new binary columns (0s and 1s) for each category in the original variable. Each category in the original column is represented as a separate column, where a value of 1 indicates the presence of that category, and 0 indicates its absence.

```
# Step 4: Compile the model
model.compile(optimizer='sgd', loss='categorical_crossentropy', metrics=['accuracy'])

H = model.fit(x_train, y_train, epochs=15, batch_size=32, validation_data=(x_test, y_test))
```

Purpose: Converts integer labels (0-9) into one-hot encoded vectors for multi-class classification.

- **tf.keras.utils.to_categorical** transforms each label (e.g., 5 becomes [0, 0, 0, 0, 0, 1, 0, 0, 0, 0] for the digit "5").
- The final output layer of the network will predict probabilities for each class.

```
# Step 3: Define the network architecture
model = Sequential([
    Dense(64, activation='relu', input_shape=(784,)),
    Dense(64, activation='relu'),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
```

- **Sequential Model**: A linear stack of layers.
- **Layers**:
  - Dense(64, activation='relu'): Each hidden layer has 64 neurons and uses the ReLU (Rectified Linear Unit) activation function.
  - **Input Layer**: The first Dense layer specifies input_shape=(784,), indicating each input sample is a 784-dimensional vector.
  - **Output Layer**: Dense(10, activation='softmax'): The final layer has 10 neurons (one for each class) and uses the softmax activation to output a probability distribution across the 10 classes.

Rest of the code is self explanatory…