2. Implementing Feed-forward Neural Networks with Keras and TensorFlow

Aim: To implement a feed-forward neural network for image classification on the MNIST dataset using Keras and TensorFlow.

Theory: A **feed-forward neural network (FFNN)**, also known as a Multilayer Perceptron (MLP), is a foundational neural network architecture. It consists of an input layer, one or more hidden layers, and an output layer. Information flows in one direction from input to output without any loops. Each neuron in a layer is connected to every neuron in the next layer, forming a fully connected network.

The **MNIST dataset** is a widely used benchmark for image classification. It comprises 60,000 training images and 10,000 testing images of handwritten digits (0-9), each of size 28x28 pixels.

Steps and Explanation:

a. **Import Necessary Packages**: This step involves importing the required libraries and modules from TensorFlow, Keras, and Matplotlib for plotting.

b. **Load the Training and Testing Data**: The MNIST dataset is readily available within Keras. It needs to be loaded and then preprocessed. A common preprocessing step is to **normalize** the pixel values from the range [0, 255] to [0, 1] by dividing by 255.0. This helps in faster and more stable training.

c. **Define the Network Architecture using Keras**: * We use Sequential model, which allows building a model layer by layer. * The input images are 28x28 pixels. We first use Flatten to convert each 2D image into a 1D vector of 784 pixels. * A Dense layer with 128 neurons and a **Rectified Linear Unit (ReLU)** activation function is added as a hidden layer. ReLU is common for hidden layers due to its computational efficiency and ability to mitigate the vanishing gradient problem. * The output Dense layer has 10 neurons (one for each digit class) with a **softmax** activation function. Softmax outputs a probability distribution over the classes, indicating the model's confidence for each digit.

d. **Train the Model using SGD**: * The model is **compiled** with an optimizer, a loss function, and evaluation metrics. * **Stochastic Gradient Descent (SGD)** is chosen as the optimizer with a learning rate of 0.01. SGD updates the model's weights based on the gradient of the loss function computed on a mini-batch of data. * sparse_categorical_crossentropy is the loss function used for multi-class classification when the true labels are integers (like in MNIST). * accuracy is chosen as the metric to monitor during training and evaluation. * The fit method trains the model for a specified number of **epochs** (iterations over the entire training dataset). validation_data is provided to monitor performance on unseen data during training.

e. **Evaluate the Network**: After training, the evaluate method is used on the test set to get the final loss and accuracy, providing an unbiased estimate of the model's performance on unseen data.

f. **Plot the Training Loss and Accuracy**: Matplotlib is used to visualize the training and validation loss and accuracy over epochs. This helps in understanding how the model is learning, identifying issues like overfitting (when training accuracy is much higher than validation accuracy) or underfitting (when both accuracies are low).

Code:

Python

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.datasets import mnist
from tensorflow.keras.optimizers import SGD
import matplotlib.pyplot as plt

# 1. Import necessary packages (Done above)

# 2. Load the training and testing data
(x_train, y_train), (x_test, y_test) = mnist.load_data()

# Preprocessing: Normalize pixel values to be between 0 and 1
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

print(f"Training data shape: {x_train.shape}")
print(f"Testing data shape: {x_test.shape}")

# 3. Define the network architecture using Keras
model = Sequential([
    # Flatten the 28x28 images to a 784-dimensional vector
    Flatten(input_shape=(28, 28)),
    # A hidden layer with 128 neurons and ReLU activation
    Dense(128, activation='relu'),
    # The output layer with 10 neurons (for 10 classes) and softmax activation
    Dense(10, activation='softmax')
])

model.summary()
```

```
# 4. Train the model using SGD
# Compile the model
model.compile(optimizer=SGD(learning_rate=0.01),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Train the model for 5 epochs
print("\nTraining the model...")
history = model.fit(x_train, y_train, epochs=5, validation_data=(x_test, y_test))

# 5. Evaluate the network
print("\nEvaluating the model...")
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f'\nTest loss: {test_loss:.4f}')
print(f'Test accuracy: {test_acc:.4f}')

# 6. Plot the training loss and accuracy
plt.figure(figsize=(12, 4))

# Plotting Accuracy
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Training and Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.grid(True)
```

```
# Plotting Loss
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Training and Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```

Conclusion: This practical successfully demonstrated the implementation of a feed-forward neural network for image classification. We learned how to load and preprocess data, define a network architecture using Keras, train the model with SGD, and evaluate its performance. Visualizing the training history provides insights into the learning process and helps identify potential areas for improvement.