🧩 Code Explanation — Step by Step

---

a. Import the necessary packages

```
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
import numpy as np
```

Explanation:

- `tensorflow` → the main deep learning library (used to build and train neural networks).
- `keras` → a high-level API inside TensorFlow that makes creating models easier.
- `matplotlib.pyplot` → used for plotting graphs (like accuracy/loss curves).
- `numpy` → handles numerical operations, arrays, etc.

---

b. Load the training and testing data (MNIST dataset)

```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
```

Explanation:

- Loads the **MNIST dataset**, which has **60,000 training images** and **10,000 testing images** of handwritten digits (0–9).
- Each image is **28×28 pixels (grayscale)**.
- `x_train`, `y_train` → training images and their corresponding digit labels.
- `x_test`, `y_test` → test images and their labels (used for evaluation).

Normalize pixel values

```
x_train = x_train / 255
x_test = x_test / 255
```

Explanation:

- Pixel values originally range from **0 to 255**.

- Dividing by 255 scales them to **0–1 range**, which helps the model train faster and more accurately.
- This is called **normalization**.

---

c. Define the network architecture using Keras

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation="relu"),
    keras.layers.Dense(10, activation="softmax")
])
```

Explanation:
This defines a **feedforward neural network** step by step:

1. **`Flatten(input_shape=(28, 28))`**
    - Converts each 28×28 image into a 1D vector of 784 values (28×28 = 784).
    - Needed because the dense layer expects a flat input.
2. **`Dense(128, activation="relu")`**
    - A **fully connected (dense) layer** with **128 neurons**.
    - Each neuron uses **ReLU (Rectified Linear Unit)** activation:
      f(x)=max(0,x)f(x) = \max(0, x)f(x)=max(0,x)
    - This helps the model learn complex non-linear patterns.
3. **`Dense(10, activation="softmax")`**
    - Output layer with **10 neurons**, one for each digit (0–9).
    - **Softmax** converts raw scores into **probabilities** that sum to 1.

---

d. Compile the model

```
model.compile(optimizer="sgd",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
```

Explanation:
This step prepares the model for training:

- `optimizer="sgd"` → uses *Stochastic Gradient Descent* to update weights during training.
- `loss="sparse_categorical_crossentropy"` → suitable for multi-class classification (integer labels 0–9).
- `metrics=["accuracy"]` → tracks how often predictions match labels.

---

e. Train the model

```
history = model.fit(x_train, y_train,
                    validation_data=(x_test, y_test),
                    epochs=10)
```

Explanation:

- `fit()` trains the model for a given number of **epochs** (full passes over the dataset).
- `validation_data` checks performance on unseen data after every epoch.
- The output shows:
  - Training and validation accuracy
  - Training and validation loss

The results in your file:

```
accuracy: 0.9550 - val_accuracy: 0.9529
```

👉 means ~95% correct predictions on both training and test data.

---

f. Evaluate the model

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Test Accuracy:", test_acc)
print("Test Loss:", test_loss)
```

Explanation:

- `evaluate()` checks model performance on test data.
- It returns **loss** and **accuracy**.
- Printed values confirm the model generalizes well.

Example output:

```
Test Accuracy: 0.9529
Test Loss: 0.1583
```

---

g. Plot the accuracy graph

```
plt.plot(history.history["accuracy"])
plt.plot(history.history["val_accuracy"])
plt.title("Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend(["Train", "Validation"])
plt.show()
```

Explanation:

- Plots accuracy curves for **training** and **validation** sets across all epochs.
- Helps visualize whether the model is improving and if it's overfitting.

---

h. Plot the loss graph

```
plt.plot(history.history["loss"])
plt.plot(history.history["val_loss"])
plt.title("Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend(["Train", "Validation"])
plt.show()
```

Explanation:

- Similar to above, but plots **loss** over epochs.
- Lower loss means better fit to data.

---

i. Display sample prediction

```
index = 0
plt.imshow(x_test[index], cmap="gray")
plt.show()

prediction = model.predict(x_test)
```

```
print("Predicted digit:", np.argmax(prediction[index]))
print("Actual digit:", y_test[index])
```

Explanation:

- `index = 0` → chooses the first test image.
- `imshow()` displays it in grayscale.
- `model.predict()` → outputs probabilities for each digit (0–9).
- `np.argmax()` → picks the digit with the **highest probability**.
- Compares prediction with actual label.

🧠1️⃣Feedforward Neural Network (FNN)

Definition (simple + clear):

A **Feedforward Neural Network (FNN)** is an artificial neural network where the information moves **only in one direction — from input to output** — through a series of layers of neurons, **without any loops or backward connections**.

Structure:

```
Input Layer → Hidden Layer(s) → Output Layer
```

Each neuron in a layer is connected to every neuron in the next layer.
The output from one layer becomes the input to the next.

Working:

1. Input data is passed to the first layer.
2. Each neuron computes a **weighted sum** of its inputs and adds a **bias**.
3. An **activation function** is applied to introduce non-linearity.
4. The result passes forward to the next layer until the output is produced.

🧩 Formula for one neuron:

y=f(Wx+b)y = f(Wx + b)y=f(Wx+b)

where

- WWW = weights,
- bbb = bias,
- fff = activation function,
- xxx = input.

Key Point:

In an FNN, data **flows forward only — there are no feedback loops** (unlike RNNs).

Applications:

- Handwritten digit recognition (MNIST)
- Image and speech classification
- Function approximation
- Simple regression or prediction problems