Task 12: Building a Multilayer Perceptron (MLP) using TensorFlow and Keras

Objective: In this assignment, you will learn to build, train, and evaluate Multilayer Perceptron (MLP) models using sklearn, TensorFlow and Keras. You will work with two datasets—the Iris datasets—and experiment with different MLP architectures and hyperparameters to observe their effects on classification performance.

Part 1: MLP on the Digits Dataset

1. Dataset: Use the `load_digits` dataset from `sklearn.datasets`. This dataset contains images of handwritten digits (0–9), each represented as an 8x8 pixel grid (64 features).

2. Task: Build, train, and evaluate an MLP model to classify the digits.

3. Steps:

Data Preprocessing:

- Load the digits dataset using `sklearn.datasets.load_digits()`.
- Scale the data using `StandardScaler` to normalize the feature values.

Model Architecture:

- Define an MLP model using Keras with the following structure:
- Input Layer: 64 features (input shape).
- Hidden Layers: At least two hidden layers, with 128 and 64 neurons, respectively, each with ReLU activation.
- Output Layer: 10 neurons with softmax activation (for the 10-digit classes).
- ```
  mlp = MLPClassifier(hidden_layer_sizes=(128, 64), activation='relu', solver='adam', max_iter=100)
  mlp.fit(X_train, y_train)
  ```

Compile the Model:

- Use the Adam optimizer with a learning rate of 0.001.
  ```

- For the loss function, use categorical cross-entropy.

Training:

- Train the model for 30 epochs with a batch size of 16.

Evaluation:

- Evaluate the model on a test split (e.g., 20%) and report the test accuracy.

Visualization:

- Plot the training and validation accuracy over epochs to show model learning.
- Show the original images and model predictions on 5 samples from the test set.

4. Hints:

- Use `train_test_split` from `sklearn.model_selection` to split the data.

- Use `LabelBinarizer` to one-hot encode the target labels.

Part 2: MLP on the Iris Dataset

1. Dataset: Use the `load_iris` dataset from `sklearn.datasets`. This dataset consists of 150 samples of iris flowers, classified into three species based on four features (sepal and petal dimensions).

```
import numpy as np

import tensorflow as tf

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler, LabelBinarizer

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from tensorflow.keras.optimizers import Adam

import matplotlib.pyplot as plt
```

2. Task: Build, train, and evaluate a smaller MLP model to classify the iris species.

3. Steps:

Data Preprocessing:

- Load the Iris dataset using `sklearn.datasets.load_iris()`.
- Standardize the data using `StandardScaler`.
- Binarize the data.

```
label_binarizer = LabelBinarizer()

y_train = label_binarizer.fit_transform(y_train)

y_test = label_binarizer.transform(y_test)
```

Model Architecture:

- Define an MLP model with the following structure:
- Input Layer: 4 features (input shape).
- Hidden Layers: Use one hidden layer with 64 neurons and ReLU activation.
- Output Layer: 3 neurons with softmax activation (for the three classes of iris).

```
# Define the MLP model

model = Sequential([

  Dense(64, input_shape=(X_train.shape[1],), activation='relu'),  # First hidden layer

  Dense(32, activation='relu'),  # Second hidden layer

  Dense(3, activation='softmax')  # Output layer with 3 classes (Iris Setosa, Versicolor, Virginica)

])
```

Compile the Model:

- Use the Adam optimizer with a learning rate of 0.01.
- Set the loss function to categorical cross-entropy.

```
# Compile the model

model.compile(optimizer=Adam(learning_rate=0.01),

        loss='categorical_crossentropy',
```

```
                metrics=['accuracy'])
```

Training:

- Train the model for 50 epochs with a batch size of 8.

```
# Train the model

history = model.fit(X_train, y_train, epochs=50, batch_size=8,
validation_split=0.1, verbose=1)
```

Evaluation:

- Evaluate model accuracy on a test set (use a 20% test split) and report accuracy.

```
# Evaluate the model

test_loss, test_accuracy = model.evaluate(X_test, y_test, verbose=0)

print(f"Test Accuracy: {test_accuracy:.2f}")
```

Visualization:

- Plot the training and validation accuracy over epochs.
- Display a confusion matrix for the test predictions.

```
# Plot training and validation accuracy over epochs

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

plt.plot(history.history['val_accuracy'], label='Validation Accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.show()
```

4. Hints:

One-hot encode the labels with `LabelBinarizer`.

Use the `plot_confusion_matrix` function from `sklearn.metrics` to create the confusion matrix.

Experiment with different numbers of layers and neurons in and observe how model complexity impacts performance and training time.