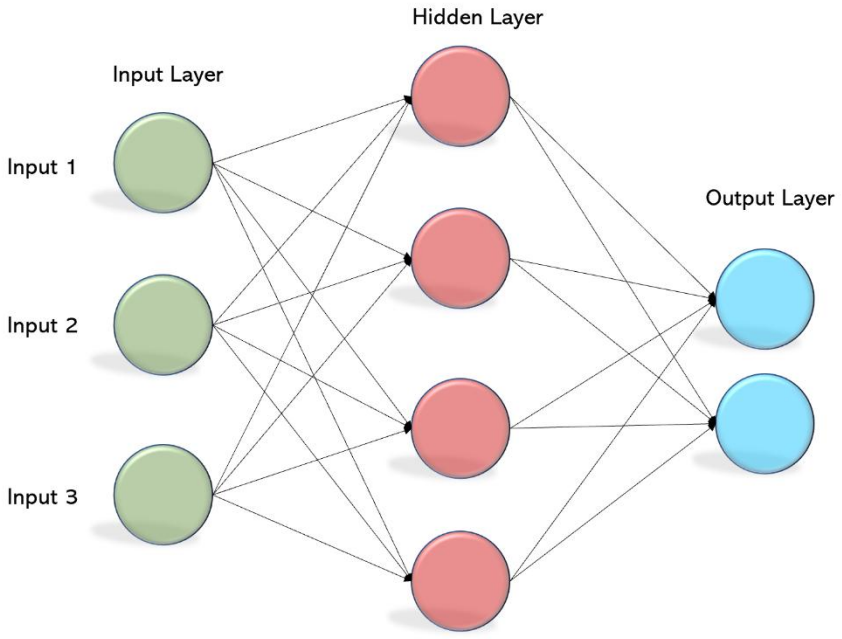
Tutorial 2:
MLP Classifier

Objectives:

- Training a Multi-Layer Perceptron
- Scaling the data
- Training the Model
- Visualizing the Curve
- Better Understanding the training process

A Multi Layer Perceptron?



Step 1 — Importing Required Libraries

The following libraries are essential:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler  # Import StandardScaler for scaling
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import classification_report, accuracy_score
import matplotlib.pyplot as plt  # Import for plotting the learning curve
```

Step 2 : Loading and Splitting the Dataset

Load the Iris dataset and split it into training and testing sets. There are 3 different classes that are Setosa (Iris setosa Versicolor (Iris versicolor) Virginica (Iris virginica)This allows us to train the model on one portion of the data and evaluate it on another.

```
# Load the Iris dataset
iris = load_iris()
X = iris.data  # Features
y = iris.target  # Labels

# Split the data into training and testing sets (70% training, 30% testing)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

Step 3 — Data Scaling

It is important for neural networks as it ensures that all features contribute equally to the learning process. The tutorial uses StandardScaler to standardize the features to have a mean of 0 and a standard deviation of 1.

```python
# Standardize the features to have mean=0 and variance=1
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)  # Fit on training data and transform
X_test_scaled = scaler.transform(X_test)        # Transform test data (using same scaling)
```

Step 4— Creating and Training the MLP Classifier

Create an MLP classifier with two hidden layers, each containing 10 neurons. Train, it is using the scaled data. The maths of it is explained with another example at the end of the document. In the context of machine learning and specifically the MLPClassifier from Scikit-learn, the random_state=42 is used to set a seed for the random number generator. This ensures reproducibility of your results.

```python
# Create an MLP classifier with two hidden layers of 10 neurons each
mlp = MLPClassifier(hidden_layer_sizes=(10 , 10), max_iter=1000, random_state=42,learning_rate_init=0.001)

# Train the MLP classifier on the scaled training data
mlp.fit(X_train_scaled, y_train)
```

✓ 0.1s

Step 5 : Making Predictions and Evaluating the Model:

Use the trained model to predict the test data and evaluate its performance using accuracy and a detailed classification report.

```python
# Predict the test set results
y_pred = mlp.predict(X_test_scaled)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')

# Print detailed classification report
print("Classification Report:\n", classification_report(y_test, y_pred))
```

Output

```
Accuracy: 1.00
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      1.00      1.00        13
           2       1.00      1.00      1.00        13

    accuracy                           1.00        45
   macro avg       1.00      1.00      1.00        45
weighted avg       1.00      1.00      1.00        45
```

Support: The number of actual samples present in the test set for each class.

Example: Class 0 (support = 19):

There are 19 samples of class 0 in the test set.

Precision, recall, and F1-score are all 1.00, meaning the model predicted all samples of class 0 correctly.

Macro Average: Macro averaging calculates the metric independently for each class and then takes the average (without considering the number of samples in each class). It treats all classes equally.

Example Calculation: Suppose you have the following precision values for three classes:

- Class 0: Precision = 0.90
- Class 1: Precision = 0.80
- Class 2: Precision = 1.00

The macro precision would be:

$$\text{Macro Precision} = \frac{0.90 + 0.80 + 1.00}{3} = 0.90$$

Weighted Average: Weighted averaging calculates the metric for each class, then takes the average weighted by the number of true instances (support) for each class. This way, classes with more samples contribute more to the final score.

Example Calculation: Suppose you have the following precision values and support for three classes:

- Class 0: Precision = 0.90, Support = 50
- Class 1: Precision = 0.80, Support = 30
- Class 2: Precision = 1.00, Support = 20

$$\text{Weighted Precision} = \frac{(0.90 \times 50) + (0.80 \times 30) + (1.00 \times 20)}{50 + 30 + 20} = \frac{45 + 24 + 20}{100} = 0.89$$

Step 6: Displaying the MLP Structure and Training Information:

Print information about the structure of the trained MLP model, such as the number of layers, the activation function used, and the number of epochs.

```
# Display the structure of the MLP classifier
print("\nMLP Structure:")
print(f"Number of layers: {mlp.n_layers_}")
print(f"Number of outputs: {mlp.n_outputs_}")
print(f"Activation function: {mlp.activation}")
print(f"Output activation function: {mlp.out_activation_}")
print(f"Number of epochs: {mlp.n_iter_}")
```

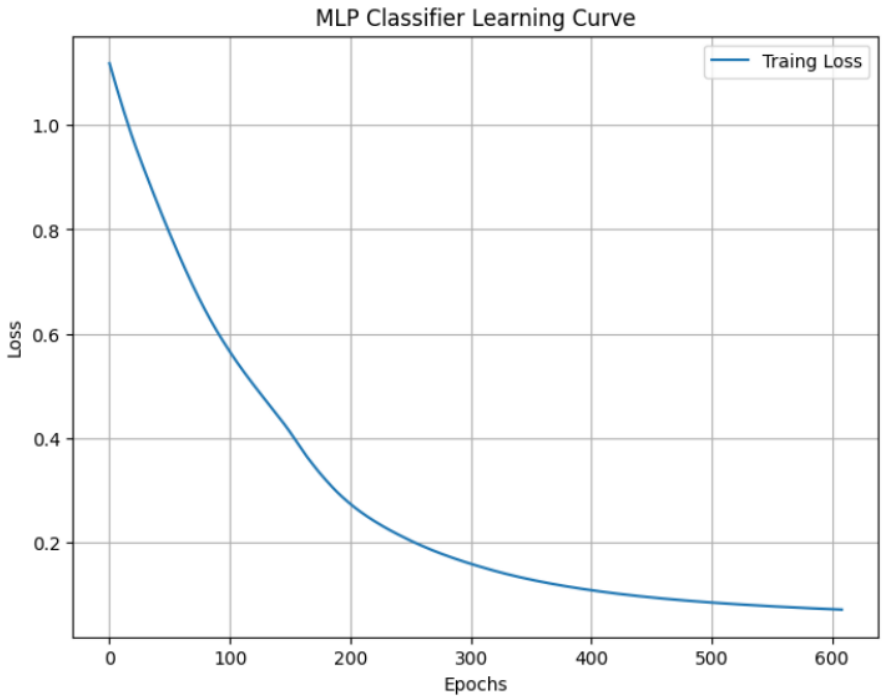
✓ 0.0s

Output:

```
MLP Structure:
Number of layers: 4
Number of outputs: 3
Activation function: relu
Output activation function: softmax
Number of epochs: 609
```

Step 7 : Visualizing the Learning Curve

```python
# Plot the loss curve
plt.figure(figsize=(8, 6))
plt.plot(mlp.loss_curve_, label='Traing Loss')
plt.title('MLP Classifier Learning Curve')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid()
plt.show()
```



Tasks:

1. **Modify the MLP model to have different numbers of hidden layers and neurons**
    - How does increasing the number of layers or neurons affect the accuracy and learning curve?
    - What configuration gives the best performance?
2. Change the learning rate of the MLP model and observe its effect on convergence.
    - Train the model and plot the learning curve.
    - How does changing the learning rate affect the loss curve and the number of epochs?

- What learning rate provides the best balance between convergence speed and model performance?

Mathematics of the MLP model:

MLP Architecture: Let us take an example of the following MLP model

- **Input Layer:** 2 neurons (for a 2-dimensional input vector).
- **Hidden Layer:** 2 neurons with sigmoid activation.
- **Output Layer:** 1 neuron with sigmoid activation (binary classification).

1. Forward Pass:

We'll start with the forward pass to compute the output and then move on to the backpropagation.

1.1. Inputs:

The input vector X is:

$$X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.1 \end{bmatrix}$$

1.2. Weights and Biases:

Weights between Input and Hidden Layer and Bias for Hidden Layer

$$W_1 = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \qquad W_1 = \begin{bmatrix} 0.4 & 0.1 \\ 0.3 & 0.2 \end{bmatrix}$$

$$b_1 = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$$

Weights between Hidden and Output Layer Bias for Output Layer:

$$W_2 = \begin{bmatrix} 0.3 \\ 0.5 \end{bmatrix}$$

$$b_2 = 0.1$$

1.3. Forward Pass Calculations:

Hidden layer Input

$$Z_1 = W_1 \cdot X + b_1 = \begin{bmatrix} 0.4 & 0.1 \\ 0.3 & 0.2 \end{bmatrix} \cdot \begin{bmatrix} 0.5 \\ 0.1 \end{bmatrix} + \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$$

$$Z_1 = \begin{bmatrix} 0.4 \cdot 0.5 + 0.1 \cdot 0.1 + 0.1 \\ 0.3 \cdot 0.5 + 0.2 \cdot 0.1 + 0.2 \end{bmatrix} = \begin{bmatrix} 0.31 \\ 0.38 \end{bmatrix}$$

Hidden layer Activation

$$A_1 = \sigma(Z_1) = \begin{bmatrix} \sigma(0.31) \\ \sigma(0.38) \end{bmatrix} = \begin{bmatrix} 0.576 \\ 0.594 \end{bmatrix}$$

Where $\sigma(x) = \frac{1}{1+e^{-x}}$.

Output layer input

$$Z_2 = W_2 \cdot A_1 + b_2 = \begin{bmatrix} 0.3 & 0.5 \end{bmatrix} \cdot \begin{bmatrix} 0.576 \\ 0.594 \end{bmatrix} + 0.1$$

$$Z_2 = 0.3 \cdot 0.576 + 0.5 \cdot 0.594 + 0.1 = 0.5402$$

Output Layer Activation (Sigmoid):

$$A_2 = \sigma(Z_2) = \sigma(0.5402) = 0.6319$$

1.4. Loss Calculation:

Assume the true label is y=1.

Loss Function (Binary Cross-Entropy Loss):

$$L = -[y \cdot \log(A_2) + (1 - y) \cdot \log(1 - A_2)]$$

$$L = -[1 \cdot \log(0.6319) + 0 \cdot \log(1 - 0.6319)] = -\log(0.6319) = 0.458$$

2. Backward Pass:

2.1. Output Layer:

1. Gradient of Loss w.r.t. Output Activation A_2

$$\frac{\partial L}{\partial A_2} = -\frac{y}{A_2} + \frac{1 - y}{1 - A_2} = -\frac{1}{0.6319} + 0 = -1.582$$

2. **Gradient of Output Activation w.r.t. Output Layer Input Z_2:** The derivative of the sigmoid function $\sigma(x)$ is $\sigma(x)(1 - \sigma(x))$.

$$\frac{\partial A_2}{\partial Z_2} = \sigma'(Z_2) = A_2 \times (1 - A_2) = 0.6319 \times (1 - 0.6319) = 0.2326$$

3. **Gradient of Loss w.r.t. Output Layer Input Z_2:**

$$\frac{\partial L}{\partial Z_2} = \frac{\partial L}{\partial A_2} \cdot \frac{\partial A_2}{\partial Z_2} = -1.582 \times 0.2326 = -0.368$$

4. **Gradient of Loss w.r.t. Weights W_2:**

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial Z_2} \cdot \frac{\partial Z_2}{\partial W_2} = \frac{\partial L}{\partial Z_2} \cdot A_1$$

$$\frac{\partial L}{\partial W_2} = -0.368 \cdot \begin{bmatrix} 0.576 \\ 0.594 \end{bmatrix} = \begin{bmatrix} -0.212 \\ -0.218 \end{bmatrix}$$

5. **Gradient of Loss w.r.t. Bias b_2:**

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial Z_2} = -0.368$$

2.2. Hidden Layer:

1. Gradient of Loss w.r.t. Hidden Layer Activations A_1:

$$\frac{\partial L}{\partial A_1} = \frac{\partial L}{\partial Z_2} \cdot \frac{\partial Z_2}{\partial A_1} = \frac{\partial L}{\partial Z_2} \cdot W_2$$

$$\frac{\partial L}{\partial A_1} = -0.368 \cdot \begin{bmatrix} 0.3 \\ 0.5 \end{bmatrix} = \begin{bmatrix} -0.1104 \\ -0.184 \end{bmatrix}$$

2. Gradient of Hidden Activation A_1 w.r.t. Hidden Layer Input Z_1:

$$\frac{\partial A_1}{\partial Z_1} = \sigma'(Z_1) = A_1 \times (1 - A_1) = \begin{bmatrix} 0.576 \times (1 - 0.576) \\ 0.594 \times (1 - 0.594) \end{bmatrix} = \begin{bmatrix} 0.244 \\ 0.241 \end{bmatrix}$$

3. Gradient of Loss w.r.t. Hidden Layer Input Z_1:

$$\frac{\partial L}{\partial Z_1} = \frac{\partial L}{\partial A_1} \cdot \frac{\partial A_1}{\partial Z_1} = \begin{bmatrix} -0.1104 \\ -0.184 \end{bmatrix} \cdot \begin{bmatrix} 0.244 \\ 0.241 \end{bmatrix}$$

$$\frac{\partial L}{\partial Z_1} = \begin{bmatrix} -0.1104 \times 0.244 \\ -0.184 \times 0.241 \end{bmatrix} = \begin{bmatrix} -0.0269 \\ -0.0443 \end{bmatrix}$$

4. Gradient of Loss w.r.t. Weights W_1:

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial Z_1} \cdot \frac{\partial Z_1}{\partial W_1}$$

$$\frac{\partial L}{\partial W_1} = \begin{bmatrix} -0.0269 \\ -0.0443 \end{bmatrix}$$

5. Gradient of Loss w.r.t bias b1:

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial Z_1}$$

Remember:

The activation function itself is not updated; instead, its derivative is used in calculating the gradient of the loss with respect to the input to the activation function.

The smaller the derivative, the less sensitive the output is to changes in the weight, meaning smaller updates. (for that reason, sigmoid is not used in hidden layers and results to vanish gradient problem)

Just replace the sigmoid with relu in hidden layer the value will not converge to zero.