Perceptron and Multi-Layer Perceptron Activity

Notebook 1: Simple Perceptron

```
# Simple Perceptron Implementation

import numpy as np

def step_function(x):
    return 1 if x >= 0 else 0

class Perceptron:
    def __init__(self, input_size, learning_rate=0.1):
        self.weights = np.zeros(input_size + 1)

    def predict(self, x):
        x = np.insert(x, 0, 1)
        return step_function(np.dot(self.weights, x))

    def train(self, X, y, epochs=10):
        for _ in range(epochs):
            for xi, target in zip(X, y):
                xi = np.insert(xi, 0, 1)
                prediction = step_function(np.dot(self.weights, xi))
                error = target - prediction
                self.weights += 0.1 * error * xi

X = np.array([[0,0], [0,1], [1,0], [1,1]])
y = np.array([0, 1, 1, 1])

p = Perceptron(input_size=2)
p.train(X, y)

for x in X:
    print(f"Input: {x}, Prediction: {p.predict(x)}")
```

Notebook 2: Perceptron AND Operator

```
# Perceptron AND Logic Gate

import numpy as np

def step(x):
    return 1 if x >= 0 else 0

class Perceptron:
    def __init__(self, input_size, lr=0.1):
        self.weights = np.zeros(input_size + 1)
        self.lr = lr

    def predict(self, x):
        x = np.insert(x, 0, 1)
```

```
        return step(np.dot(self.weights, x))

    def train(self, X, y, epochs=10):
        for _ in range(epochs):
            for xi, yi in zip(X, y):
                xi = np.insert(xi, 0, 1)
                y_hat = self.predict(xi[1:])
                error = yi - y_hat
                self.weights += self.lr * error * xi

X = np.array([[0,0], [0,1], [1,0], [1,1]])
y = np.array([0, 0, 0, 1])

model = Perceptron(input_size=2)
model.train(X, y)

print("Testing AND gate:")
for x in X:
    print(f"{x} -> {model.predict(x)}")
```

Notebook 3: Multi-layer Perceptron (Sigmoid)

```
# Multi-Layer Perceptron with Sigmoid

import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])

np.random.seed(1)

input_layer_neurons = X.shape[1]
hidden_neurons = 4
output_neurons = 1

hidden_weights = 2 * np.random.random((input_layer_neurons, hidden_neurons)) - 1
output_weights = 2 * np.random.random((hidden_neurons, output_neurons)) - 1

for epoch in range(10000):
    hidden_input = np.dot(X, hidden_weights)
    hidden_output = sigmoid(hidden_input)

    final_input = np.dot(hidden_output, output_weights)
    final_output = sigmoid(final_input)
```

```
    error = y - final_output
    d_output = error * sigmoid_derivative(final_output)

    error_hidden = d_output.dot(output_weights.T)
    d_hidden = error_hidden * sigmoid_derivative(hidden_output)

    output_weights += hidden_output.T.dot(d_output)
    hidden_weights += X.T.dot(d_hidden)

print("Final Predictions after training:")
print(final_output.round(3))
```

Reflection

Through this activity, I have learned the key differences between a simple perceptron and a multi-layer perceptron (MLP). The simple perceptron works well for linearly separable problems such as the AND and OR gates. It uses a step activation function and updates weights based on errors using a simple learning rule. However, it cannot solve non-linear problems like XOR.

The MLP, enhanced with a sigmoid activation function and backpropagation, can solve complex, non-linear problems by using one or more hidden layers. I also understood the importance of choosing the right architecture and dataset, and how model complexity relates to data patterns. This activity gave me hands-on experience using Jupyter Notebooks, and running neural network models, which deepened my understanding of machine learning fundamentals.