Challenges-6

Implementing a simple perceptron with two inputs and a sigmoid activation function in Python, and then use the perceptron learning rule to train it for the NAND and XOR binary logic functions.

1. Sigmoid Activation Function

The sigmoid function is defined as:

$\sigma(x)=1+e-x1$

Its output is between 0 and 1, making it suitable for binary classification tasks.

Python Implementation of Sigmoid:

Python

```
import numpy as np

def sigmoid(x):
    """The sigmoid activation function."""
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    """Derivative of the sigmoid function."""
    s = sigmoid(x)
    return s * (1 - s)
```

2. Perceptron Structure

Our perceptron will have:

- Two inputs (x1, x2)
- Two corresponding weights (w1, w2)
- A bias term (b) with a corresponding weight (wb=1)
- A weighted sum: $z=w1x1+w2x2+b$
- An output: $a=\sigma(z)$

3. Perceptron Learning Rule

The perceptron learning rule (with gradient descent for a sigmoid activation) updates the weights based on the error between the predicted output and the target output. The weight update rule is:

$\Delta w_i = \alpha \cdot (\text{target} - \text{output}) \cdot \text{input}_i \quad \Delta b = \alpha \cdot (\text{target} - \text{output}) \cdot 1$

where:

- α is the learning rate (a small positive constant).
- target is the desired output for a given input.
- output is the perceptron's prediction.
- input_i is the i-th input.

For the bias, the input is effectively 1.

Python Implementation of the Perceptron:

Python

```
class Perceptron:
    def __init__(self, num_inputs):
        self.weights = np.random.rand(num_inputs)
        self.bias = np.random.rand(1)[0]
        self.learning_rate = 0.1

    def predict(self, inputs):
        weighted_sum = np.dot(inputs, self.weights) + self.bias
        return sigmoid(weighted_sum)

    def train(self, training_inputs, targets, epochs):
        for epoch in range(epochs):
            for inputs, target in zip(training_inputs, targets):
                prediction = self.predict(inputs)
                error = target - prediction

                # Update weights
                for i in range(len(self.weights)):
                    self.weights[i] += self.learning_rate * error * inputs[i]

                # Update bias
                self.bias += self.learning_rate * error

            if epoch % 1000 == 0:
                loss = np.mean((targets - [self.predict(x) for x in training_inputs]) ** 2)
                print(f"Epoch {epoch}, Loss: {loss:.4f}")
```

4. Training for NAND Function

The truth table for NAND is:

Input 1	Input 2	Output
0	0	1
0	1	1

1	0	1
1	1	0

Let's train our perceptron:

Python

```
# Training data for NAND
nand_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
nand_targets = np.array([1, 1, 1, 0])

# Create and train the perceptron
nand_perceptron = Perceptron(num_inputs=2)
nand_perceptron.train(nand_inputs, nand_targets, epochs=10000)

# Test the trained perceptron for NAND
print("\nNAND Gate Results:")
for inputs in nand_inputs:
    prediction = nand_perceptron.predict(inputs)
    print(f"Input: {inputs}, Prediction: {prediction:.4f}, Target: {nand_targets[np.where((nand_inputs ==
inputs).all(axis=1))[0][0]]}")
```

5. Training for XOR Function

The truth table for XOR is:

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Let's train our perceptron:

Python

```
# Training data for XOR
xor_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
xor_targets = np.array([0, 1, 1, 0])

# Create and train the perceptron
xor_perceptron = Perceptron(num_inputs=2)
xor_perceptron.train(xor_inputs, xor_targets, epochs=20000) # XOR might need more epochs

# Test the trained perceptron for XOR
print("\nXOR Gate Results:")
for inputs in xor_inputs:
    prediction = xor_perceptron.predict(inputs)
```

```
    print(f"Input: {inputs}, Prediction: {prediction:.4f}, Target: {xor_targets[np.where((xor_inputs == inputs).all(axis=1))[0][0]]}")
```

Explanation:

- We define the sigmoid and its derivative (although the derivative isn't strictly needed for the basic perceptron learning rule used here, it's crucial for more advanced training algorithms like backpropagation).
- The Perceptron class initializes weights and bias randomly.
- The predict method calculates the weighted sum of inputs and applies the sigmoid activation.
- The train method iterates through the training data for a specified number of epochs, updating the weights and bias based on the error.
- We then create instances of the Perceptron and train them on the NAND and XOR datasets.
- Finally, we test the trained perceptrons on all possible input combinations and print the predictions alongside the targets.

Observations:

You'll likely observe that the perceptron trained on the **NAND** function converges relatively well, and the predictions will be close to the target binary outputs (0 or 1).

Challenge -7:

Visualizing the learning process of a **single-layer perceptron** (as the 2D plane visualization of a separating line is straightforward for this case) trained on a linearly separable function like **NAND**. Visualizing the decision boundaries of a multi-layer perceptron in a 2D plane for XOR is more complex as the separation isn't a single line but rather regions defined by the hidden layer's transformations.

1. Visualizing the Learning Process (Single-Layer Perceptron for NAND)

For a single-layer perceptron with two inputs (x1,x2), the decision boundary is a line defined by the equation:

$w1x1+w2x2+b=0$

We can rewrite this as:

$x2=−w2w1x1−w2b$

This is the equation of a line where the slope is −w2w1 and the y-intercept is −w2b. During training, the perceptron learning rule adjusts w1, w2, and b, causing this line to move and rotate in the 2D input space until it correctly separates the data points.

Python Implementation with Visualization (using Matplotlib):

Python

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

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

class SimplePerceptronVisualizer:
    def __init__(self, num_inputs):
        self.weights = np.random.rand(num_inputs) - 0.5
        self.bias = np.random.rand(1)[0] - 0.5
        self.learning_rate = 0.1
        self.history_weights = []
        self.history_bias = []

    def predict(self, inputs):
        weighted_sum = np.dot(inputs, self.weights) + self.bias
        return 1 if sigmoid(weighted_sum) >= 0.5 else 0

    def train_step(self, inputs, target):
        prediction = self.predict(inputs)
        error = target - prediction
        self.weights += self.learning_rate * error * inputs
        self.bias += self.learning_rate * error
        self.history_weights.append(self.weights.copy())
        self.history_bias.append(self.bias)

    def plot_decision_boundary(self, ax, x_min, x_max, y_min, y_max):
        if self.weights[1] == 0:
            return  # Avoid division by zero
        x = np.linspace(x_min, x_max, 100)
        y = (-self.weights[0] * x - self.bias) / self.weights[1]
        ax.plot(x, y, 'r--')

# Training data for NAND
nand_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
nand_targets = np.array([1, 1, 1, 0])

# Initialize perceptron and lists for animation
perceptron_visualizer = SimplePerceptronVisualizer(num_inputs=2)
fig, ax = plt.subplots()
ax.set_xlim(-0.5, 1.5)
ax.set_ylim(-0.5, 1.5)
scatter = ax.scatter(nand_inputs[:, 0], nand_inputs[:, 1], c=nand_targets, cmap=plt.cm.RdYlBu, s=100)
```

```
line, = ax.plot([], [], 'g-', linewidth=2)
title = ax.text(0.5, 1.1, '', transform=ax.transAxes, ha='center')

def animate(i):
    perceptron_visualizer.weights = perceptron_visualizer.history_weights[i]
    perceptron_visualizer.bias = perceptron_visualizer.history_bias[i]
    if perceptron_visualizer.weights[1] != 0:
        x = np.linspace(-0.5, 1.5, 100)
        y = (-perceptron_visualizer.weights[0] * x - perceptron_visualizer.bias) / perceptron_visualizer.weights[1]
        line.set_data(x, y)
    title.set_text(f"Iteration: {i+1}, Weights: [{perceptron_visualizer.weights[0]:.2f},
{perceptron_visualizer.weights[1]:.2f}], Bias: {perceptron_visualizer.bias:.2f}")
    return line, title

def train_and_animate(perceptron, inputs, targets, epochs):
    perceptron.history_weights.append(perceptron.weights.copy())
    perceptron.history_bias.append(perceptron.bias)
    for _ in range(epochs):
        for input_data, target in zip(inputs, targets):
            perceptron.train_step(input_data, target)

# Train for a few epochs to see the movement
train_and_animate(perceptron_visualizer, nand_inputs, nand_targets, epochs=20)

ani = animation.FuncAnimation(fig, animate, frames=len(perceptron_visualizer.history_weights), interval=500,
blit=True)
plt.xlabel("Input 1")
plt.ylabel("Input 2")
plt.title("Perceptron Learning for NAND")
plt.show()
```

Explanation of the Visualization Code:

1. **SimplePerceptronVisualizer:** This class is similar to the `Perceptron` but stores the history of weights and bias during training. It also has a `plot_decision_boundary` method.
2. **Training Data:** We use the NAND truth table.
3. **Initialization:** We initialize the perceptron and Matplotlib figure and axes. The NAND data points are plotted with colors representing their target values.
4. **animate(i):** This function is called for each frame of the animation. It updates the perceptron's weights and bias to their values at the `i`-th step of training and then recalculates and plots the decision boundary line. The title also updates to show the current weights and bias.
5. **train_and_animate:** This function runs the training for a specified number of epochs, storing the weights and bias at each step.
6. **animation.FuncAnimation:** This creates the animation by repeatedly calling the `animate` function.

When you run this code, you'll see a scatter plot of the NAND data points. A red dashed line (initially random) will move and rotate as the perceptron learns, eventually trying to separate the (0,0), (0,1), and (1,0) points (target 1) from the (1,1) point (target 0).

Visualizing XOR is More Complex for a Single Line:

Since XOR is not linearly separable, a single straight line cannot perfectly divide the input space. To visualize the learning process for a multi-layer perceptron solving XOR, you would need to:

- **Visualize the decision regions** created by the hidden layer's neurons. Each hidden neuron creates its own separating line, and their combined output defines more complex regions in the input space.
- Potentially visualize the **output of the hidden layer** as a transformation of the input space. The hidden layer maps the non-linearly separable XOR problem into a space where it becomes linearly separable for the output layer. This visualization would likely involve plotting the output of the two hidden neurons as a 2D space where the XOR classes might be separable by a line.

Creating a clear and intuitive animation of this process in 2D is significantly more challenging and often involves visualizing the activation maps of the hidden neurons or the transformation of the input space.

Multi-Layer Feed-Forward Perceptron for XOR (Already Implemented)

where we implemented a multi-layer feed-forward perceptron with backpropagation to solve the XOR problem. The Python code for that solution is provided in the previous response. You can run that code to train the network and see that it learns the XOR function.

To summarize the visualization part:

- We successfully visualized the learning process of a **single-layer perceptron** for a linearly separable problem (NAND) by showing the movement of its decision boundary line in the 2D input space.
- Visualizing the learning process of a **multi-layer perceptron** for a non-linearly separable problem like XOR in a simple 2D plane is much harder and would require visualizing the transformations learned by the hidden layers, rather than just a single separating line.

Challenge -8:

implement a multi-layer feed-forward perceptron network with two input neurons, two hidden neurons, and one output neuron in Python. We'll use the backpropagation algorithm to train it to solve the XOR logical function.

1. Network Architecture

Our network will have the following structure:

- **Input Layer:** 2 neurons (corresponding to the two inputs of XOR).
- **Hidden Layer:** 2 neurons with sigmoid activation.
- **Output Layer:** 1 neuron with sigmoid activation (producing an output between 0 and 1, which we'll interpret as a binary output).

2. Sigmoid Activation Function and its Derivative

We'll reuse the sigmoid function and its derivative from the previous example:

Python
```
import numpy as np

def sigmoid(x):
    """The sigmoid activation function."""
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    """Derivative of the sigmoid function."""
    s = sigmoid(x)
    return s * (1 - s)
```

3. Neural Network Implementation

Python
```
class MultiLayerPerceptron:
    def __init__(self, num_inputs, num_hidden, num_outputs):
        self.num_inputs = num_inputs
        self.num_hidden = num_hidden
        self.num_outputs = num_outputs

        # Initialize weights with random values
        self.weights_input_hidden = np.random.rand(self.num_inputs, self.num_hidden) - 0.5
        self.bias_hidden = np.random.rand(1, self.num_hidden) - 0.5
        self.weights_hidden_output = np.random.rand(self.num_hidden, self.num_outputs) - 0.5
        self.bias_output = np.random.rand(1, self.num_outputs) - 0.5
        self.learning_rate = 0.1

    def forward(self, inputs):
        # Hidden layer
        self.hidden_layer_input = np.dot(inputs, self.weights_input_hidden) + self.bias_hidden
        self.hidden_layer_output = sigmoid(self.hidden_layer_input)
```

```
        # Output layer
        self.output_layer_input = np.dot(self.hidden_layer_output, self.weights_hidden_output) + self.bias_output
        self.output_layer_output = sigmoid(self.output_layer_input)

        return self.output_layer_output

    def backward(self, inputs, targets, outputs):
        # Output layer error
        output_error = targets - outputs
        output_delta = output_error * sigmoid_derivative(self.output_layer_input)

        # Hidden layer error
        hidden_error = np.dot(output_delta, self.weights_hidden_output.T)
        hidden_delta = hidden_error * sigmoid_derivative(self.hidden_layer_input)

        # Update weights and biases
        self.weights_hidden_output += self.learning_rate * np.dot(self.hidden_layer_output.T, output_delta)
        self.bias_output += self.learning_rate * np.sum(output_delta, axis=0, keepdims=True)
        self.weights_input_hidden += self.learning_rate * np.dot(inputs.T, hidden_delta)
        self.bias_hidden += self.learning_rate * np.sum(hidden_delta, axis=0, keepdims=True)

    def train(self, training_inputs, targets, epochs):
        for epoch in range(epochs):
            for inputs, target in zip(training_inputs, targets):
                # Forward pass
                output = self.forward(inputs)

                # Backward pass and weight update
                self.backward(inputs, target, output)

            if epoch % 1000 == 0:
                predictions = np.array([self.forward(x) for x in training_inputs])
                loss = np.mean((targets - predictions) ** 2)
                accuracy = np.mean(((predictions > 0.5).astype(int) == targets.astype(int)))
                print(f"Epoch {epoch}, Loss: {loss:.4f}, Accuracy: {accuracy:.4f}")

    def predict(self, inputs):
        return self.forward(inputs)
```

4. Training for XOR Function

Python

```
# Training data for XOR
xor_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
xor_targets = np.array([[0], [1], [1], [0]])

# Create and train the multi-layer perceptron
mlp = MultiLayerPerceptron(num_inputs=2, num_hidden=2, num_outputs=1)
mlp.train(xor_inputs, xor_targets, epochs=20000)

# Test the trained network for XOR
```

```
print("\nXOR Gate Results:")
for inputs, target in zip(xor_inputs, xor_targets):
    prediction = mlp.predict(inputs)
    print(f"Input: {inputs}, Prediction: {prediction[0]:.4f}, Target: {target[0]}")
```

Explanation:

- **__init__:** Initializes the network with the specified number of input, hidden, and output neurons. It also initializes the weights and biases with small random values.
- **forward:** Performs the forward pass through the network. It calculates the weighted sum of inputs at each layer, adds the bias, and applies the sigmoid activation function.
- **backward:** Implements the backpropagation algorithm.
    - Calculates the error at the output layer.
    - Computes the delta (error multiplied by the derivative of the activation function) for the output layer.
    - Propagates the error backward to the hidden layer and calculates the hidden layer error and delta.
    - Updates the weights and biases based on the calculated deltas and the learning rate.
- **train:** Iterates through the training data for a specified number of epochs, performing the forward and backward passes for each training example. It also prints the loss and accuracy periodically to monitor training progress.
- **predict:** Performs a forward pass to get the network's prediction for a given input.