Single Layer Perceptron

Aim:

To implement single layer perceptron on linearly and non linearly separable data

Single-Layer Perceptron (SLP):

A Single-Layer Perceptron (SLP) is a basic neural network used for binary classification. It processes input data by computing a weighted sum of inputs, adding a bias, and applying an activation function to produce an output. The model learns by adjusting the weights to improve prediction accuracy.

The training process involves the following steps:

- Initialize weights and bias.
- Compute output using the activation function.
- Update weights and bias based on the difference between actual and predicted outputs.
- Repeat until the model converges or a predefined number of iterations is reached.

SLPs are suitable for simple tasks but struggle with non-linearly separable data and more complex problems.

Algorithm:

1. **Initialize** weights, bias, learning rate, and number of iterations.
2. **Define activation function** to classify output as 0 or 1.
3. **Train the model** by computing the output using weights and bias, applying the activation function, calculating the error, and updating the weights and bias.
4. **Predict output** for new inputs using the trained model.
5. **Apply to AND/XOR gates**, train the perceptron, and print results.
6. **Note**: Perceptron works for linearly separable data but fails for non-linearly separable cases like XOR.

```
import numpy as np

class Perceptron(object):

    def __init__(self):
```

```python
        self.lr = 0.1  # Learning Rate
        self.n_iterations = 5  # Iterations
        self.weights = None
        self.bias = None

    def activation(self, x):
        return np.where(x >= 0, 1, 0)  # Step Activation function for
binary classification

    def fit(self, X, y):
        '''
        Shapes: X:(n_samples, n_features) y:(n_samples,)
        '''
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)  # Initialize the weights
to 0
        self.bias = 0  # Initialize the bias to 0

        for i in range(self.n_iterations):
            Z = np.dot(X, self.weights) + self.bias  # Linear Output
            y_pred = self.activation(Z)  # Activation

            error = y - y_pred  # Calculate error by subtracting
predicted value from true value
            print(f"Iter {i}: Error: {error.sum()}")  # Display error
at each iteration

            updates = self.lr * error  # Update factor

            # Update weights and bias
            self.weights += np.dot(X.T, updates)  # Update weights
            self.bias += np.sum(updates)  # Update bias

    def predict(self, X):
        Z = np.dot(X, self.weights) + self.bias  # Linear Output
        y_pred = self.activation(Z)  # Activation
        return y_pred

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

percp = Perceptron()
percp.fit(X, y)
predictions = percp.predict(X)
```

```python
print(f"Actual Value: {y}")
print(f"Predicted Value: {predictions}")
```

```
Iter 0: Error: -3
Iter 1: Error: 1
Iter 2: Error: 1
Iter 3: Error: 0
Iter 4: Error: 0
Actual Value: [0 0 0 1]
Predicted Value: [0 0 0 1]
```

```python
class Perceptron(object):

    def __init__(self):
        self.lr = 0.1  # Learning Rate
        self.n_iterations = 5  # Iterations
        self.weights = None
        self.bias = None

    def activation(self, x):
        return np.where(x >= 0, 1, 0)  # Step Activation function for
binary classification

    def fit(self, X, y):
        '''
        Shapes: X:(n_samples, n_features) y:(n_samples,)
        '''
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)  # Initialize the weights
to 0
        self.bias = 0  # Initialize the bias to 0

        for i in range(self.n_iterations):
            Z = np.dot(X, self.weights) + self.bias  # Linear Output
            y_pred = self.activation(Z)  # Activation

            error = y - y_pred  # Calculate error by subtracting
predicted value from true value
            print(f"Iter {i}: Error: {error.sum()}")  # Display error
at each iteration

            updates = self.lr * error  # Update factor

            # Update weights and bias
            self.weights += np.dot(X.T, updates)  # Update weights
            self.bias += np.sum(updates)  # Update bias

    def predict(self, X):
        Z = np.dot(X, self.weights) + self.bias  # Linear Output
        y_pred = self.activation(Z)  # Activation
```

```python
        return y_pred

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

percp = Perceptron()
percp.fit(X, y)
predictions = percp.predict(X)

print(f"Actual Value: {y}")
print(f"Predicted Value: {predictions}")
```

```
Iter 0: Error: -2
Iter 1: Error: 2
Iter 2: Error: -2
Iter 3: Error: 2
Iter 4: Error: -2
Actual Value: [0 1 1 0]
Predicted Value: [0 0 0 0]
```