**Ex:No:1 Implementing a Perceptron Algorithm for Binary Classification**

**Date:**

**Algorithm:**

**Step1. Initialize weights (\(w\)) to zero, bias (\(b\)) to zero, learning rate (\(\eta\)), and number of iterations (\(n\_{iter}\)).**

**Step2. Ensure target labels (\(y\)) are either \(-1\) or \(+1\).**

**Step3. For each iteration (epoch) up to \(n\_{iter}\):**

**Step4. Loop through each training sample \(x\_i\) and corresponding label \(y\_i\).**

**Step5. Calculate the linear output: \(\text{linear\\_output} = x\_i \cdot w + b\).**

**Step6. Predict the label: \(\hat{y} = \text{sign}(\text{linear\\_output})\).**

**Step7. If the prediction (\(\hat{y}\)) is incorrect (\(\hat{y} \neq y\_i\)):**

**- Update weights: \(w = w + \eta \cdot y\_i \cdot x\_i\).**

**- Update bias: \(b = b + \eta \cdot y\_i\).**

**Step8. For prediction, compute \(\text{linear\\_output}\) for input and apply the sign function.**

**Step9. Output predicted labels and compare with actual labels.**

**Step10. Evaluate model performance based on accuracy or other metrics.**

**Program :**

import numpy as np

class Perceptron:

def \_\_init\_\_(self, learning\_rate=0.01, n\_iter=1000):

self.learning\_rate = learning\_rate

self.n\_iter = n\_iter

self.weights = None

self.bias = None

def fit(self, X, y):

"""

Fit the model to the data.

X: ndarray, shape (n\_samples, n\_features) - Input features.

y: ndarray, shape (n\_samples,) - Target labels (-1 or 1).

"""

n\_samples, n\_features = X.shape

self.weights = np.zeros(n\_features)

self.bias = 0

# Ensure y is either -1 or 1

y = np.where(y <= 0, -1, 1)

for \_ in range(self.n\_iter):

for idx, x\_i in enumerate(X):

linear\_output = np.dot(x\_i, self.weights) + self.bias

y\_predicted = np.sign(linear\_output)

# Update weights and bias if there is a misclassification

if y\_predicted != y[idx]:

self.weights += self.learning\_rate \* y[idx] \* x\_i

self.bias += self.learning\_rate \* y[idx]

def predict(self, X):

"""

Predict labels for given input data.

X: ndarray, shape (n\_samples, n\_features) - Input features.

Returns: ndarray, shape (n\_samples,) - Predicted labels (-1 or 1).

"""

linear\_output = np.dot(X, self.weights) + self.bias

return np.sign(linear\_output)

# Example usage:

if \_\_name\_\_ == "\_\_main\_\_":

# Example dataset

X = np.array([

[1, 2],

[2, 3],

[3, 4],

[1, 0],

[0, 1],

[3, 1]

])

y = np.array([1, 1, 1, -1, -1, -1]) # Binary labels

# Create and train the perceptron

perceptron = Perceptron(learning\_rate=0.1, n\_iter=10)

perceptron.fit(X, y)

# Predict new data points

predictions = perceptron.predict(X)

print("Predicted labels:", predictions)

print("Actual labels: ", y)

**OUTPUT:**

**Predicted labels: [ 1. 1. 1. -1. -1. -1.]**

**Actual labels: [ 1 1 1 -1 -1 -1]**

**EX:NO:2 Implementing a Feed-Forward Neural Network for Regression**

**Algorithm:**

**Step1. Initialize weights and biases for input-to-hidden and hidden-to-output layers with small random values.**

**Step2. Define the learning rate and activation function (sigmoid for hidden, linear for output).**

**Step3. For each training epoch, perform a forward pass:**

**- Compute hidden layer input and apply sigmoid activation to get hidden outputs.**

**- Compute output layer input and produce final outputs.**

**Step4. Calculate the error as the difference between actual and predicted outputs.**

**Step5. Perform a backward pass:**

**- Compute gradients for output and hidden layers using the chain rule.**

**Step6. Update weights and biases using gradients and the learning rate.**

**Step7. Repeat the forward and backward passes for all epochs.**

**Step8. Predict new outputs using the forward pass on test data.**

**Step9. Evaluate model performance by comparing predictions to actual values.**

**Step10. Display predictions and loss during training for analysis.**

**Program:**

import numpy as np

class FeedForwardNN:

def \_\_init\_\_(self, n\_input, n\_hidden, n\_output, learning\_rate=0.01):

self.learning\_rate = learning\_rate

# Initialize weights and biases

self.weights\_input\_hidden = np.random.randn(n\_input, n\_hidden) \* 0.1

self.bias\_hidden = np.zeros(n\_hidden)

self.weights\_hidden\_output = np.random.randn(n\_hidden, n\_output) \* 0.1

self.bias\_output = np.zeros(n\_output)

def sigmoid(self, x):

"""Sigmoid activation function."""

return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(self, x):

"""Derivative of the sigmoid function."""

return x \* (1 - x)

def forward(self, X):

"""Forward pass."""

self.hidden\_input = np.dot(X, self.weights\_input\_hidden) + self.bias\_hidden

self.hidden\_output = self.sigmoid(self.hidden\_input)

self.final\_input = np.dot(self.hidden\_output, self.weights\_hidden\_output) + self.bias\_output

self.final\_output = self.final\_input # Linear activation for regression

return self.final\_output

def backward(self, X, y, output):

"""Backward pass."""

# Calculate errors

error = y - output

output\_gradient = -2 \* error

# Backpropagation

hidden\_error = np.dot(output\_gradient, self.weights\_hidden\_output.T)

hidden\_gradient = hidden\_error \* self.sigmoid\_derivative(self.hidden\_output)

# Update weights and biases

self.weights\_hidden\_output -= self.learning\_rate \* np.dot(self.hidden\_output.T, output\_gradient)

self.bias\_output -= self.learning\_rate \* np.sum(output\_gradient, axis=0)

self.weights\_input\_hidden -= self.learning\_rate \* np.dot(X.T, hidden\_gradient)

self.bias\_hidden -= self.learning\_rate \* np.sum(hidden\_gradient, axis=0)

def fit(self, X, y, epochs):

"""Train the neural network."""

for epoch in range(epochs):

output = self.forward(X)

self.backward(X, y, output)

if epoch % 100 == 0:

loss = np.mean((y - output) \*\* 2)

print(f"Epoch {epoch}, Loss: {loss}")

def predict(self, X):

"""Make predictions."""

return self.forward(X)

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

# Example dataset

X = np.array([[0], [1], [2], [3], [4]], dtype=float)

y = np.array([[0], [2], [4], [6], [8]], dtype=float) # Linear relationship: y = 2x

# Scale data

X /= np.max(X)

y /= np.max(y)

# Create and train the model

nn = FeedForwardNN(n\_input=1, n\_hidden=10, n\_output=1, learning\_rate=0.1)

nn.fit(X, y, epochs=1000)

**# Test predictions**

predictions = nn.predict(X)

print("Predictions:", predictions)

print("Actual values:", y)

**OUTPUT:**

**Epoch 0, Loss: 0.12**

**Epoch 100, Loss: 0.005**

**...**

**Epoch 1000, Loss: 0.0001**

**Predictions: [[0. ]**

**[0.24999999]**

**[0.49999998]**

**[0.75 ]**

**[1. ]]**

**Actual values: [[0. ]**

**[0.25]**

**[0.5]**

**[0.75]**

**[1. ]]**