Experiment-4

AIM: AdaGrad optimization

Code:

import numpy as np

class AdaGradOptimizer:

def \_\_init\_\_(self, learning\_rate=0.01, epsilon=1e-8):

self.learning\_rate = learning\_rate

self.epsilon = epsilon

self.grad\_squares = {}

def update(self, param, grad):

if param not in self.grad\_squares:

self.grad\_squares[param] = np.zeros\_like(grad)

self.grad\_squares[param] += grad \*\* 2

adjusted\_grad = grad / (np.sqrt(self.grad\_squares[param]) + self.epsilon)

return self.learning\_rate \* adjusted\_grad

class DNN:

def \_\_init\_\_(self, layer\_sizes):

self.params = {}

self.layer\_sizes = layer\_sizes

self.initialize\_params()

def initialize\_params(self):

for i in range(1, len(self.layer\_sizes)):

self.params[f'W{i}'] = np.random.randn(self.layer\_sizes[i], self.layer\_sizes[i-1]) \* 0.01

self.params[f'b{i}'] = np.zeros((self.layer\_sizes[i], 1))

def forward(self, X):

cache = {'A0': X}

A = X

for i in range(1, len(self.layer\_sizes)):

W = self.params[f'W{i}']

b = self.params[f'b{i}']

Z = np.dot(W, A) + b

A = np.maximum(0, Z) # ReLU activation

cache[f'Z{i}'] = Z

cache[f'A{i}'] = A

return A, cache

def compute\_loss(self, A, Y):

m = Y.shape[1]

loss = -np.sum(Y \* np.log(A + 1e-8)) / m

return loss

def backward(self, cache, Y):

grads = {}

m = Y.shape[1]

A\_final = cache[f'A{len(self.layer\_sizes)-1}']

dA = - (np.divide(Y, A\_final + 1e-8) - np.divide(1 - Y, 1 - A\_final + 1e-8))

for i in reversed(range(1, len(self.layer\_sizes))):

dZ = dA \* (cache[f'Z{i}'] > 0)

dW = np.dot(dZ, cache[f'A{i-1}'].T) / m

db = np.sum(dZ, axis=1, keepdims=True) / m

dA = np.dot(self.params[f'W{i}'].T, dZ)

grads[f'dW{i}'] = dW

grads[f'db{i}'] = db

return grads

def update\_params(self, grads, optimizer):

for i in range(1, len(self.layer\_sizes)):

self.params[f'W{i}'] -= optimizer.update(f'W{i}', grads[f'dW{i}'])

self.params[f'b{i}'] -= optimizer.update(f'b{i}', grads[f'db{i}'])

def train(self, X, Y, epochs=1000, learning\_rate=0.01):

optimizer = AdaGradOptimizer(learning\_rate=learning\_rate)

for epoch in range(epochs):

A, cache = self.forward(X)

loss = self.compute\_loss(A, Y)

grads = self.backward(cache, Y)

self.update\_params(grads, optimizer)

if epoch % 100 == 0:

print(f'Epoch {epoch}, Loss: {loss:.4f}')

# Example usage:

np.random.seed(42)

X = np.random.randn(2, 500)

Y = (np.sum(X, axis=0, keepdims=True) > 0).astype(float)

dnn = DNN([2, 4, 1])

dnn.train(X, Y, epochs=1000, learning\_rate=0.1)