Exp 7

Build a Deep Feed Forward ANN by implementing the Backpropagation algorithm and test the same using appropriate data sets. Use the number of hidden layers >=4

import numpy as np

import matplotlib.pyplot as plt

from sklearn.datasets import load\_iris

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler, OneHotEncoder

from sklearn.metrics import confusion\_matrix, ConfusionMatrixDisplay

# ------------------ Data Preprocessing ------------------

# Load the dataset

iris = load\_iris()

X = iris.data

y = iris.target.reshape(-1, 1)

# Standardize features

scaler = StandardScaler()

X = scaler.fit\_transform(X)

# One-hot encode labels

encoder = OneHotEncoder(sparse\_output=False)

y = encoder.fit\_transform(y)

# Train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# ------------------ Activation Functions ------------------

def relu(x):

    return np.maximum(0, x)

def relu\_derivative(x):

    return (x > 0).astype(float)

def softmax(x):

    e\_x = np.exp(x - np.max(x, axis=1, keepdims=True))

    return e\_x / e\_x.sum(axis=1, keepdims=True)

# ------------------ Loss Functions ------------------

def cross\_entropy\_loss(y\_true, y\_pred):

    return -np.mean(np.sum(y\_true \* np.log(y\_pred + 1e-8), axis=1))

def cross\_entropy\_derivative(y\_true, y\_pred):

    return y\_pred - y\_true

# ------------------ Deep ANN Implementation ------------------

class DeepANN:

    def \_\_init\_\_(self, layer\_sizes, learning\_rate=0.01):

        self.layer\_sizes = layer\_sizes

        self.lr = learning\_rate

        self.weights = []

        self.biases = []

        self.loss\_history = []

        # Initialize weights and biases

        for i in range(len(layer\_sizes) - 1):

            weight = np.random.randn(layer\_sizes[i], layer\_sizes[i + 1]) \* np.sqrt(2. / layer\_sizes[i])

            bias = np.zeros((1, layer\_sizes[i + 1]))

            self.weights.append(weight)

            self.biases.append(bias)

    def forward(self, X):

        activations = [X]

        zs = []

        for i in range(len(self.weights) - 1):

            z = np.dot(activations[-1], self.weights[i]) + self.biases[i]

            zs.append(z)

            a = relu(z)

            activations.append(a)

        # Output layer with softmax

        z = np.dot(activations[-1], self.weights[-1]) + self.biases[-1]

        zs.append(z)

        a = softmax(z)

        activations.append(a)

        return activations, zs

    def backward(self, activations, zs, y\_true):

        grads\_w = [None] \* len(self.weights)

        grads\_b = [None] \* len(self.biases)

        # Output layer

        delta = cross\_entropy\_derivative(y\_true, activations[-1])

        grads\_w[-1] = np.dot(activations[-2].T, delta)

        grads\_b[-1] = np.sum(delta, axis=0, keepdims=True)

        # Hidden layers

        for i in reversed(range(len(grads\_w) - 1)):

            delta = np.dot(delta, self.weights[i + 1].T) \* relu\_derivative(zs[i])

            grads\_w[i] = np.dot(activations[i].T, delta)

            grads\_b[i] = np.sum(delta, axis=0, keepdims=True)

        # Update weights and biases

        for i in range(len(self.weights)):

            self.weights[i] -= self.lr \* grads\_w[i]

            self.biases[i] -= self.lr \* grads\_b[i]

    def train(self, X, y, epochs=1000):

        for epoch in range(epochs):

            activations, zs = self.forward(X)

            loss = cross\_entropy\_loss(y, activations[-1])

            self.loss\_history.append(loss)

            self.backward(activations, zs, y)

            if epoch % 100 == 0:

                print(f"Epoch {epoch} | Loss: {loss:.4f}")

    def predict(self, X):

        activations, \_ = self.forward(X)

        return np.argmax(activations[-1], axis=1)

    def accuracy(self, X, y\_true):

        preds = self.predict(X)

        labels = np.argmax(y\_true, axis=1)

        return np.mean(preds == labels)

# ------------------ Model Setup ------------------

# 4 hidden layers + input(4) and output(3)

layer\_sizes = [4, 16, 32, 16, 8, 3]

model = DeepANN(layer\_sizes, learning\_rate=0.01)

# Train the model

model.train(X\_train, y\_train, epochs=1000)

# Test accuracy

acc = model.accuracy(X\_test, y\_test)

print(f"\nTest Accuracy: {acc \* 100:.2f}%")

# ------------------ Plot Training Loss ------------------

plt.figure(figsize=(8, 4))

plt.plot(model.loss\_history, label='Training Loss')

plt.xlabel("Epoch")

plt.ylabel("Loss")

plt.title("Training Loss over Epochs")

plt.legend()

plt.grid(True)

plt.show()

# ------------------ Confusion Matrix ------------------

y\_pred = model.predict(X\_test)

y\_true = np.argmax(y\_test, axis=1)

cm = confusion\_matrix(y\_true, y\_pred)

disp = ConfusionMatrixDisplay(confusion\_matrix=cm, display\_labels=iris.target\_names)

disp.plot(cmap=plt.cm.Blues)

plt.title("Confusion Matrix")

plt.grid(False)

plt.show()





