Exp 6

Use the concept of data augmentation to increase the data size from a single image. Use any random image of your choice.

import numpy as np

from sklearn.datasets import load\_iris

from sklearn.preprocessing import StandardScaler, OneHotEncoder

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

import matplotlib.pyplot as plt

# Load and preprocess the Iris dataset

data = load\_iris()

X = data.data

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

# One-hot encoding for classification

encoder = OneHotEncoder(sparse\_output=False)

y\_encoded = encoder.fit\_transform(y)

# Normalize features

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

# Split into train and test

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_scaled, y\_encoded, test\_size=0.3, random\_state=42)

# Activation functions and their derivatives

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

def sigmoid\_deriv(x): return sigmoid(x) \* (1 - sigmoid(x))

def tanh(x): return np.tanh(x)

def tanh\_deriv(x): return 1 - np.tanh(x) \*\* 2

def relu(x): return np.maximum(0, x)

def relu\_deriv(x): return (x > 0).astype(float)

# Neural Network class

class NeuralNetwork:

    def \_\_init\_\_(self, input\_size, hidden\_size, output\_size, activation='sigmoid'):

        self.W1 = np.random.randn(input\_size, hidden\_size)

        self.b1 = np.zeros((1, hidden\_size))

        self.W2 = np.random.randn(hidden\_size, output\_size)

        self.b2 = np.zeros((1, output\_size))

        # Select activation

        if activation == 'sigmoid':

            self.act = sigmoid

            self.act\_deriv = sigmoid\_deriv

        elif activation == 'tanh':

            self.act = tanh

            self.act\_deriv = tanh\_deriv

        elif activation == 'relu':

            self.act = relu

            self.act\_deriv = relu\_deriv

        else:

            raise ValueError("Unsupported activation function")

        self.activation\_name = activation

    def forward(self, X):

        self.z1 = X @ self.W1 + self.b1

        self.a1 = self.act(self.z1)

        self.z2 = self.a1 @ self.W2 + self.b2

        self.a2 = sigmoid(self.z2)  # Output layer uses sigmoid for multi-class

        return self.a2

    def backward(self, X, y, output, lr=0.1):

        output\_error = output - y

        dW2 = self.a1.T @ (output\_error \* sigmoid\_deriv(self.z2))

        db2 = np.sum(output\_error \* sigmoid\_deriv(self.z2), axis=0, keepdims=True)

        hidden\_error = (output\_error @ self.W2.T) \* self.act\_deriv(self.z1)

        dW1 = X.T @ hidden\_error

        db1 = np.sum(hidden\_error, axis=0, keepdims=True)

        # Update weights

        self.W1 -= lr \* dW1

        self.b1 -= lr \* db1

        self.W2 -= lr \* dW2

        self.b2 -= lr \* db2

    def train(self, X, y, epochs=500, lr=0.1):

        loss\_history = []

        for epoch in range(epochs):

            output = self.forward(X)

            self.backward(X, y, output, lr)

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

            loss\_history.append(loss)

        return loss\_history

    def predict(self, X):

        output = self.forward(X)

        return np.argmax(output, axis=1)

# Train and compare models with different activation functions

activations = ['sigmoid', 'tanh', 'relu']

results = {}

for act in activations:

    print(f"\nTraining with activation: {act}")

    nn = NeuralNetwork(4, 8, 3, activation=act)

    loss\_hist = nn.train(X\_train, y\_train, epochs=500, lr=0.1)

    preds = nn.predict(X\_test)

    accuracy = np.mean(preds == np.argmax(y\_test, axis=1)) \* 100

    results[act] = (loss\_hist, accuracy)

    print(f"Test Accuracy: {accuracy:.2f}%")

# Plot training losses

plt.figure(figsize=(10, 5))

for act in activations:

    plt.plot(results[act][0], label=f"{act} (Acc: {results[act][1]:.2f}%)")

plt.title("Training Loss Comparison")

plt.xlabel("Epochs")

plt.ylabel("Loss")

plt.legend()

plt.grid()

plt.show()

