**Department of Electrical Engineering and   
Computer Science**

**Faculty Member:** Dr. Ahmad Salman **Dated:** 29/11/2023

**Semester:** 7th **Section:** BEE 12C

**CS-471 Machine Learning**

Lab 11: Neural Networks

**Group Members**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  |  | **PLO4 - CLO4** | | **PLO5 -CLO5** | **PLO8 -CLO6** | **PLO9 -CLO7** |
| **Name** | **Reg. No** | **Viva / Quiz / Lab Performance** | **Analysis of Data in Lab Report** | **Modern Tool Usage** | **Ethics and Safety** | **Individual and Teamwork** |
|  |  | **5 Marks** | **5 Marks** | **5 Marks** | **5 Marks** | **5 Marks** |
| Afif Arif Siddiqi | 344504 |  |  |  |  |  |
| Muhammad Ali Farooq | 331879 |  |  |  |  |  |
| Muhammad Ahmed Mohsin | 333060 |  |  |  |  |  |
| Danial Ahmad | 331388 |  |  |  |  |  |
| Ehtishaam Tanveer | 333074 |  |  |  |  |  |
| Muhammad Umer | 345834 |  |  |  |  |  |

**Table of Contents**

[2 Neural Networks 3](#_Toc152335161)

[2.1 Introduction 3](#_Toc152335162)

[2.2 Objectives 3](#_Toc152335163)

[2.3 Theory 3](#_Toc152335164)

[3 Lab Tasks 4](#_Toc152335165)

[3.1 Task 1 – Activation Functions 4](#_Toc152335166)

[3.2 Task 2 – Matrix Initializations 6](#_Toc152335167)

[3.3 Task 3 – Forward Propagation 8](#_Toc152335168)

[3.4 Task 4 – Backward Propagation 9](#_Toc152335169)

[3.5 Task 5 – Training and Testing 11](#_Toc152335170)

[4 Conclusion 15](#_Toc152335171)

# Neural Networks

## Introduction

This laboratory exercise will focus on the introduction of Artificial Neural Networks which form a very wide and popular domain of machine learning. Due to the advancements in computing technology, neural networks have seen a major rise in machine learning implementations. They are used in many application areas such as in image classification, object detection, sequence models and natural language processing.

## Objectives

The following are the main objectives of this lab:

* Implement Sigmoid and ReLU activation functions
* Initialize matrices for neural network implementation
* Use vectorization for neural network implementation
* Forward propagate to determine the loss
* Backward propagate to determine the weight derivatives
* Update weight parameters to fit the model

## Theory

A Neural Network is an arrangement of numerous “neuron” units through which a dataset is propagated to determine the error of the model. The neuron units are stacked into a sequence of layers each of which learns a particular feature of the model. Associated between the layers are weighted parameters which are to be trained. The training examples are forward propagated through the network to determine the cost. The cost is then back propagated through the network to determine the change in the cost w.r.t. the change in the parameters. The result is then used to update the weights. At the end of the training, the weights will possess values which will cause the network to make a prediction with minimal errors.

# Lab Tasks

**Importing necessary libraries**

import os

import matplotlib.pyplot as plt

import pandas as pd

import numpy as np

plt.rcParams["figure.figsize"] = (6, 4)

plt.rcParams["font.family"] = "STIXGeneral"

import warnings

warnings.filterwarnings("ignore")

## Task 1 – Activation Functions

Write the code for the following activation functions:

Sigmoid(z) = 1 / (1 + e-z)

ReLU(z) = max(0,z)

dReLU(z) =

Call each function with any value for z and take screenshots of your work. Provide the code and all relevant screenshots.

### TASK 1 CODE STARTS HERE ###

*def* sigmoid(*z*):

    return 1 / (1 + np.exp(-z))

*def* relu(*z*):

    return np.maximum(0, z)

*def* drelu(*z*):

    return np.where(z > 0, 1, 0)

z = np.linspace(-5, 5, 1000)

plt.plot(z, sigmoid(z), *label*="sigmoid")

plt.plot(z, relu(z), *label*="ReLU")

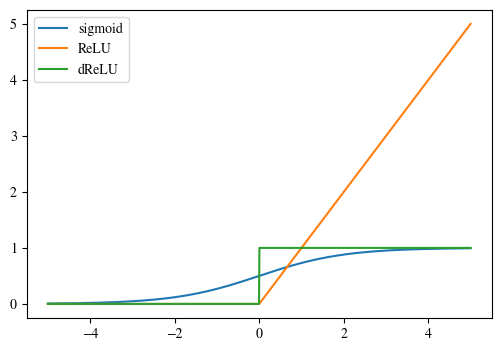
plt.plot(z, drelu(z), *label*="dReLU")

plt.legend()

plt.show()

### TASK 1 CODE ENDS HERE ###

### TASK 1 SCREENSHOT STARTS HERE ###



### TASK 1 SCREENSHOT ENDS HERE ###

Download a dataset containing 3 feature columns and 1 label column with binary values. You will design and train a neural network on the dataset to predict the y values.

df = pd.read\_csv("diabetes.csv")

features = ["DiabetesPedigreeFunction", "BMI", "BloodPressure"]

df.dropna(*inplace*=True)

X = df[features].values

y = df["Outcome"].values

*def* feature\_scaling(*X*, *axis*=0):

    return (X - X.min(*axis*=axis)) / (X.max(*axis*=axis) - X.min(*axis*=axis))

*def* random\_split(*X*, *y*, *ratio*=0.8, *scaling*=feature\_scaling):

    m = len(X)

    X = np.random.permutation(X)

    split = *int*(m \* ratio)

    return (

        feature\_scaling(X[:split]),

        y[:split].reshape(1, -1),

        feature\_scaling(X[split:]),

        y[split:].reshape(1, -1),

    )

X\_train, y\_train, X\_test, y\_test = random\_split(X, y)

print("X\_train shape: ", X\_train.shape)

print("y\_train shape: ", y\_train.shape)

print("X\_test shape: ", X\_test.shape)

print("y\_test shape: ", y\_test.shape)

## Task 2 – Matrix Initializations

Divide your dataset into training and test portions and import them into your python program. Write code to initialize the different matrices you need for your neural network implementation:

* Initialize all dataset examples as array Xtrain (n × mtrain), Xtest (n × mtest)
* Initialize all corresponding labels as array Ytrain (1 × mtrain), Ytest (1 × mtest)
* Initialize remaining layers Z1, A1, Z2, A2, Z3, A3 as zero arrays with appropriate sizes
* Initialize derivative layers dZ1, dZ2, dZ3, dZ1cache, dZ2cache, dZ3cache as zero arrays with appropriate sizes
* Initialize weight matrices W1, B1, W2, B2, W3, B3 with random numbers
* Initialize weight gradient matrices dW1, dB1, dW2, dB2, dW3, dB3 as zero arrays

### TASK 2 CODE STARTS HERE ###

layers = [3, 5, 5, 1]

*# Initialize W and b*

parameters = {}

for i in range(1, len(layers)):

    parameters["W" + *str*(i)] = np.random.normal(

        0, layers[i - 1] \*\* -0.5, (layers[i], layers[i - 1])

    )

    parameters["B" + *str*(i)] = np.random.randn(layers[i], 1)

print("Shape of W and b:")

for key, value in parameters.items():

    print(key, value.shape)

*# Initialize Z and A*

Z = {}

A = {}

for i in range(1, len(layers)):

    Z["Z" + *str*(i)] = np.zeros((layers[i], 1))

    A["A" + *str*(i)] = np.zeros((layers[i], 1))

print("\nShape of Z and A:")

for (kZ, kV), (kA, vA) in zip(Z.items(), A.items()):

    print(kZ, kV.shape, "|", kA, vA.shape)

*# Initialize dZ, dZcache, dW, dB*

dZ = {}

dZcache = {}

dW = {}

dB = {}

for i in range(1, len(layers)):

    dZ["dZ" + *str*(i)] = np.zeros((layers[i], 1))

    dZcache["dZ" + *str*(i)] = np.zeros((layers[i], 1))

    dW["dW" + *str*(i)] = np.zeros((layers[i], layers[i - 1]))

    dB["dB" + *str*(i)] = np.zeros((layers[i], 1))

print("\nShape of dZ and dZcache:")

for (kZ, kV), (kA, vA) in zip(dZ.items(), dZcache.items()):

    print(kZ, kV.shape, "|", kA, vA.shape)

print("\nShape of dW and dB:")

for (kZ, kV), (kA, vA) in zip(dZ.items(), dZcache.items()):

    print(kZ, kV.shape, "|", kA, vA.shape)

*def* initialize\_parameters(*layers*):

    parameters = {}

    for i in range(1, len(layers)):

        parameters["W" + *str*(i)] = np.random.normal(

            0, layers[i - 1] \*\* -0.5, (layers[i], layers[i - 1])

        )

        parameters["B" + *str*(i)] = np.random.randn(layers[i], 1)

    return parameters

*def* initialize\_dicts(*layers*):

    Z = {}

    A = {}

    dZ = {}

    dZcache = {}

    dW = {}

    dB = {}

    for i in range(1, len(layers)):

        Z["Z" + *str*(i)] = np.zeros((layers[i], 1))

        A["A" + *str*(i)] = np.zeros((layers[i], 1))

        dZ["dZ" + *str*(i)] = np.zeros((layers[i], 1))

        dZcache["dZ" + *str*(i)] = np.zeros((layers[i], 1))

        dW["dW" + *str*(i)] = np.zeros((layers[i], layers[i - 1]))

        dB["dB" + *str*(i)] = np.zeros((layers[i], 1))

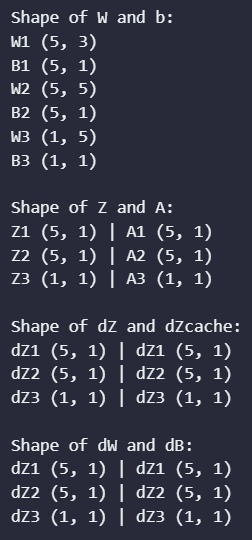
    return Z, A, dZ, dZcache, dW, dB

parameters = initialize\_parameters(layers)

Z, A, dZ, dZcache, dW, dB = initialize\_dicts(layers)

### TASK 2 CODE ENDS HERE ###

### TASK 2 SCREENSHOT STARTS HERE ###



### TASK 2 SCREENSHOT ENDS HERE ###

## Task 3 – Forward Propagation

Write the code for implementing forward propagation given as follows:

Z1 = W1X + B1 A1 = ReLU(Z1)

Z2 = W2A1 + B2 A2 = ReLU(Z2)

Z3 = W3A2 + B3 A3 = Sigmoid(Z3)

Ensure that your forward propagation function works on both training and test datasets. Run your code on both the training and test datasets to give out their respective costs. Provide the code and all relevant screenshots of the final output.

### TASK 3 CODE STARTS HERE ###

*def* forward\_propagation(*X*, *parameters*, *Z*, *A*):

    Z["Z1"] = np.dot(parameters["W1"], X.T) + parameters["B1"]

    A["A1"] = relu(Z["Z1"])

    Z["Z2"] = np.dot(parameters["W2"], A["A1"]) + parameters["B2"]

    A["A2"] = relu(Z["Z2"])

    Z["Z3"] = np.dot(parameters["W3"], A["A2"]) + parameters["B3"]

    A["A3"] = sigmoid(Z["Z3"])

    return A["A3"]

print("Forward propagation on test dataset:")

y\_pred = forward\_propagation(X\_test, parameters, Z, A)

print(y\_pred.shape)

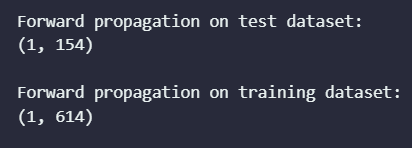
print("\nForward propagation on training dataset:")

y\_pred = forward\_propagation(X\_train, parameters, Z, A)

print(y\_pred.shape)

### TASK 3 CODE ENDS HERE ###

### TASK 3 SCREENSHOT STARTS HERE ###



### TASK 3 SCREENSHOT ENDS HERE ###

## Task 4 – Backward Propagation

Write the code for implementing backward propagation given as follows:

dZ3cache = dZ3

dZ3 = A3 - Ytrain

dW3 =

dB3 = sum dZ3cache across training examples axis

dZ2cache = dZ2

dZ2 = .\* dReLU(Z2)

dW2 =

dB2 = sum dZ2cache across training examples axis

dZ1cache = dZ1

dZ1 = .\* dReLU(Z1)

dW1 =

dB1 = sum dZ1cache across training examples axis

When working on back propagation, it is helpful to note the sizes of the matrices involved in the equations. Once the weight gradients dW1, dB1, dW2, dB2, dW3, dB3 are found, the weights can be updated using gradient descent:

The subscript L indicates the layer number. Run your code by first using forward propagation on the training dataset to determine the cost. Then, back propagate to find the derivatives (of cost w.r.t. weight). Finally, use the derivatives to update the weights. Provide the code and all screenshots showing the initial weights, cost, derivatives and updated weights.

### TASK 4 CODE STARTS HERE ###

*def* backward\_propagation(*X*, *y*, *parameters*, *Z*, *A*, *dZcache*, *dW*, *dB*):

    m = X.shape[0]

    dZcache["dZ3"] = A["A3"] - y

    dW["dW3"] = np.dot(dZcache["dZ3"], A["A2"].T) / m

    dB["dB3"] = np.sum(dZcache["dZ3"], *axis*=1, *keepdims*=True) / m

    dZcache["dZ2"] = np.dot(parameters["W3"].T, dZcache["dZ3"]) \* drelu(Z["Z2"])

    dW["dW2"] = np.dot(dZcache["dZ2"], A["A1"].T) / m

    dB["dB2"] = np.sum(dZcache["dZ2"], *axis*=1, *keepdims*=True) / m

    dZcache["dZ1"] = np.dot(parameters["W2"].T, dZcache["dZ2"]) \* drelu(Z["Z1"])

    dW["dW1"] = np.dot(dZcache["dZ1"], X) / m

    dB["dB1"] = np.sum(dZcache["dZ1"], *axis*=1, *keepdims*=True) / m

    return dZcache, dW, dB

*def* update\_parameters(*parameters*, *dW*, *dB*, *alpha*=0.01):

    for key, value in parameters.items():

        if key.startswith("W"):

            parameters[key] -= alpha \* dW["d" + key]

        elif key.startswith("B"):

            parameters[key] -= alpha \* dB["d" + key]

        else:

            raise *ValueError*("Invalid key")

    return parameters

forward\_propagation(X\_train, parameters, Z, A)

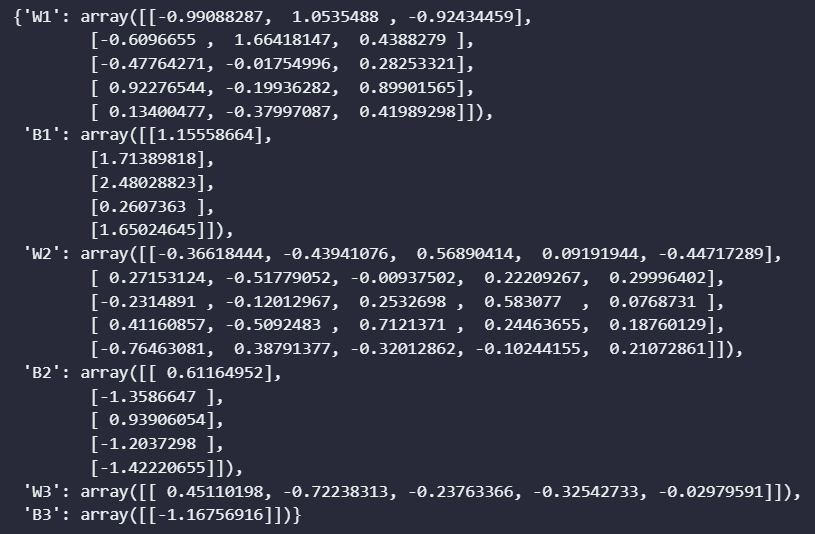
backward\_propagation(X\_train, y\_train, parameters, Z, A, dZcache, dW, dB)

update\_parameters(parameters, dW, dB)

np.savez("parameters.npz", \*\*parameters)

### TASK 4 CODE ENDS HERE ###

### TASK 4 SCREENSHOT STARTS HERE ###



### TASK 4 SCREENSHOT ENDS HERE ###

## Task 5 – Training and Testing

In this task, you will use the functions from the previous tasks to write a “main” function that performs the actual training and testing. First, forward propagate through the training examples to determine the training cost. Then, back propagate to determine the weight gradients. Next, use the gradients to update the weights. Finally, forward propagate through the test examples to determine the test cost. This single iteration over the entire dataset (both training and test) marks completion of one epoch.

You will need to perform the training and testing over several epochs (the epoch number is another hyperparameter that must be chosen). Ensure that at the end of each epoch, the training and test losses are stored for plotting purposes. When the final epoch is performed, save the trained parameters (weights and bias) and make plot of the training and test losses (y-axis) over the epochs (x-axis). Ensure that both of the losses appear on the same graph.

Tune the alpha parameter to some other values to obtain more plots. You will need to obtain at least 9 plots. Ensure that the alpha value is mentioned on each plot. Provide the code (excluding function definitions), all relevant screenshots and plots.

*You will also need to submit the best trained weights as part of the submission.*

### TASK 5 CODE STARTS HERE ###

alpha = 0.1

epochs = 100

layers = [3, 5, 5, 1]

*def* train(*X\_train*, *y\_train*, *X\_test*, *y\_test*, *parameters*, *alpha*, *epochs*, *layers*):

    parameters = initialize\_parameters(layers)

    Z, A, dZ, dZcache, dW, dB = initialize\_dicts(layers)

    train\_costs = []

    test\_costs = []

    for epoch in range(epochs):

        A3\_train = forward\_propagation(X\_train, parameters, Z, A)

        dZcache, dW, dB = backward\_propagation(

            X\_train, y\_train, parameters, Z, A, dZcache, dW, dB

        )

        parameters = update\_parameters(parameters, dW, dB, alpha)

        train\_cost = -np.mean(

            y\_train \* np.log(A3\_train) + (1 - y\_train) \* np.log(1 - A3\_train)

        )  *# Cross-entropy loss*

        train\_costs.append(train\_cost)

        A3\_test = forward\_propagation(X\_test, parameters, Z, A)

        test\_cost = -np.mean(

            y\_test \* np.log(A3\_test) + (1 - y\_test) \* np.log(1 - A3\_test)

        )  *# Cross-entropy loss*

        test\_costs.append(test\_cost)

        if epoch % 10 == 0:

            print(*f*"Epoch {epoch} | Train cost: {train\_cost} | Test cost: {test\_cost}")

    return train\_costs, test\_costs

train\_costs, test\_costs = train(

    X\_train, y\_train, X\_test, y\_test, parameters, alpha, epochs, layers

)

plt.figure(*figsize*=(6, 4))

plt.plot(train\_costs, *label*="train")

plt.plot(test\_costs, *label*="test")

plt.legend()

plt.title(*f*"alpha = {alpha}")

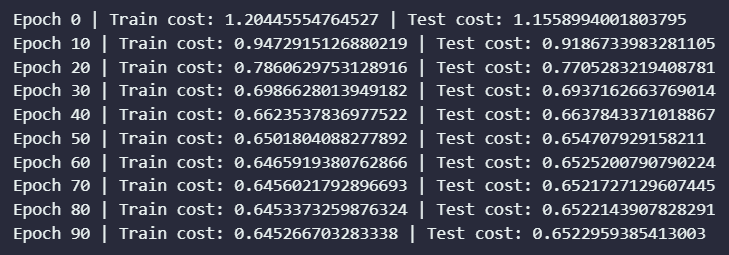
plt.xlabel("Epochs")

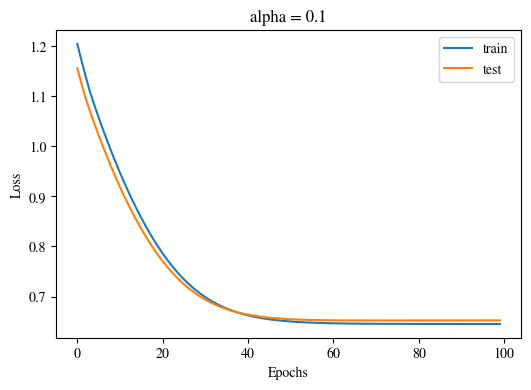
plt.ylabel("Loss")

plt.show()

### TASK 5 CODE ENDS HERE ###

### TASK 5 SCREENSHOTS START HERE ###





alpha\_values = [0.1, 0.03, 0.05, 0.01, 0.003, 0.005, 0.001, 0.0003, 0.0005]

for alpha in alpha\_values:

    train\_costs, test\_costs = train(

        X\_train, y\_train, X\_test, y\_test, parameters, alpha, epochs, layers

    )

    plt.figure(*figsize*=(6, 4))

    plt.plot(train\_costs, *label*="train")

    plt.plot(test\_costs, *label*="test")

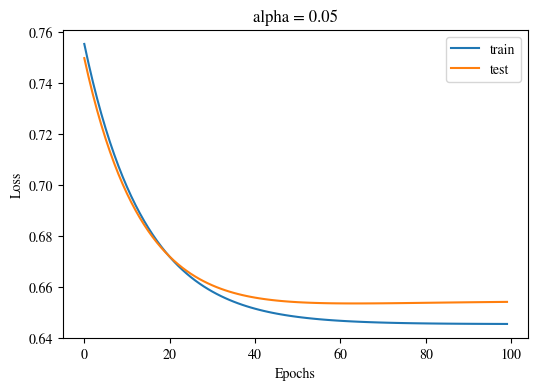
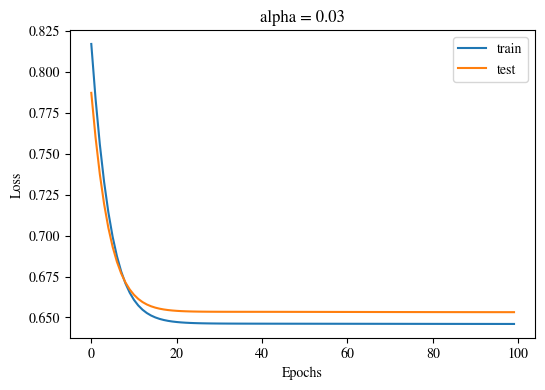
    plt.legend()

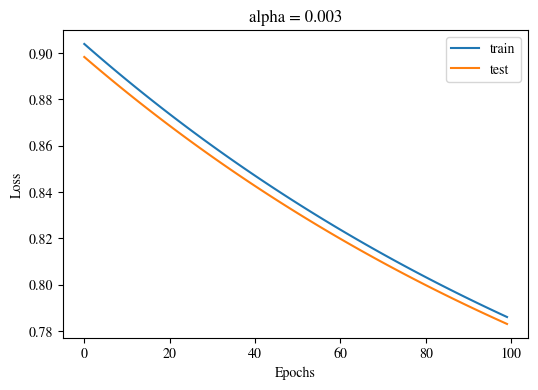
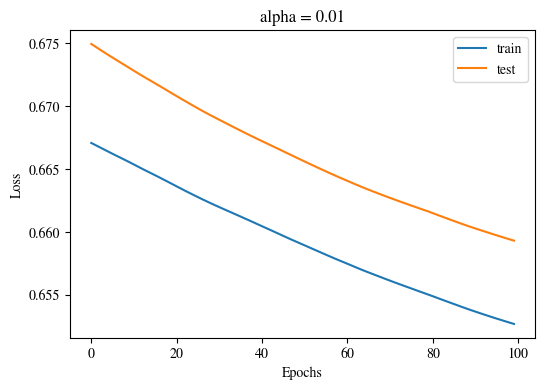
    plt.title(*f*"alpha = {alpha}")

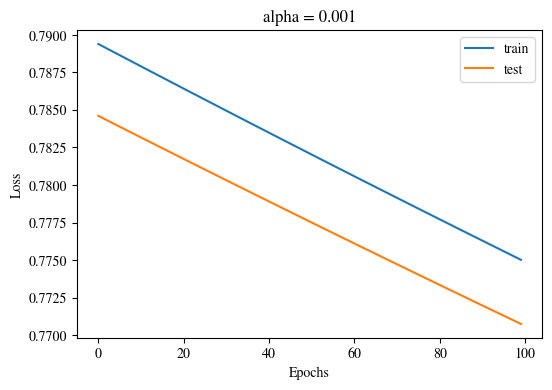
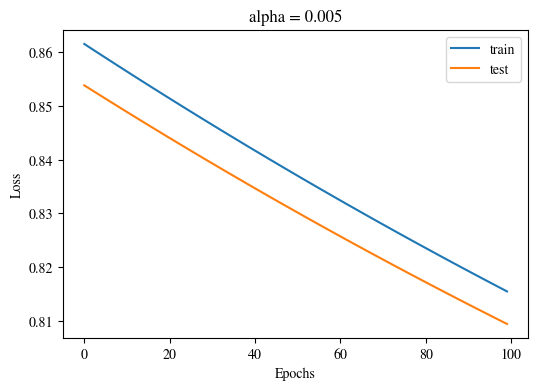
    plt.xlabel("Epochs")

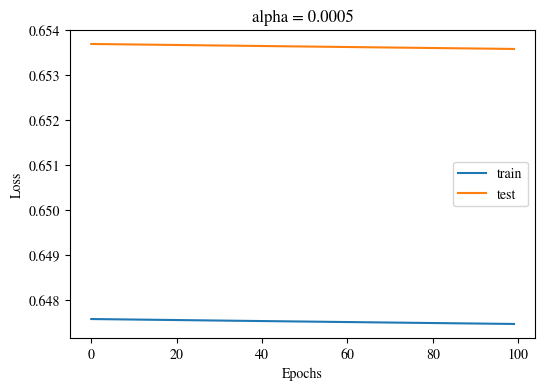
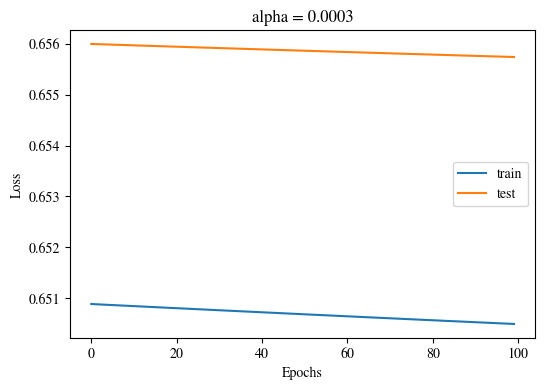
    plt.ylabel("Loss")

    plt.show()









### TASK 5 SCREENSHOTS END HERE ###

# Conclusion

In this lab, we successfully implemented Sigmoid and ReLU activation functions, initialized matrices for neural network implementation, and utilized vectorization for efficient neural network computation. We performed forward propagation to determine the loss and backward propagation to calculate the weight derivatives, effectively updating weight parameters to optimize the model. Our work demonstrates a comprehensive understanding of neural network implementation and optimization techniques.