National University of Sciences and Technology (NUST)
School of Electrical Engineering and Computer Science

Department of Electrical Engineering

Faculty Member: Ma'am Neelma Naz **Date:** December 1, 2025

Semester: 7th **Group:** 02

CS471 Machine Learning

Lab 11: Neural Networks

		PLO4	PLO5	PLO5	PLO8	PLO9
		CLO4	CLO5	CLO5	CLO6	CLO7
Student Name	**Reg. No**	**Viva / Quiz / Demo** 5 Marks	**Analysis of Data in Report** 5 Marks	**Modern Tool Usage** 5 Marks	**Ethics** 5 Marks	**Individual and Teamwork** 5 Marks
Hanzla Sajjad	403214					
Irfa Farooq	412564					

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 vectors/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

Lab Conduct

- Respect faculty and peers through speech and actions
- The lab faculty will be available to assist the students. In case some aspect of the lab experiment is not understood, the students are advised to seek help from the faculty.
- In the tasks, there are commented lines such as #YOUR CODE STARTS HERE# where you have to provide the code. You must put the code/screenshot/plot between the #START and #END parts of these commented lines. Do NOT remove the commented lines.
- Use the tab key to provide the indentation in python.

- When you provide the code in the report, keep the font size at 12

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.

A brief summary of the relevant keywords and functions in python is provided below:

print()	output text on console
input()	get input from user on console
range()	create a sequence of numbers
len()	gives the number of characters in a string
if	contains code that executes depending on a logical condition
else	connects with **if** and **elif**, executes when conditions are not met
elif	equivalent to **else if**
while	loops code as long as a condition is true
for	loops code through a sequence of items in an iterable object
break	exit loop immediately
continue	jump to the next iteration of the loop
def	used to define a function

pd.read_csv import csv file as a dataframe
df.to_csv export dataframe as a csv file

In this lab, you will write a python program that implements a shallow neural network from scratch. The neural network must contain an input layer (3 features), two hidden layers and one output layer. You are ONLY allowed to use NumPy, Pandas and MatplotLib modules.

In the given sections, you will program a vectorized implementation of the neural network, i.e. the features, labels and weights are contained inside vectors or matrices and the training will make extensive use of matrix implementations. Such vectorized methods have the advantage of being many times faster than using traditional loop-based implementation. Speed is an important metric in training as neural networks can be of extremely large sizes. As a result, NumPy computations form a central aspect of machine learning.

The pseudocode for the overall program is given as follows:

```
function definitions...
matrix initializations...
for epoch in epochs:
     forward propagate through training examples
     store training cost
     back propagate using training cost
     update the weights
     forward propagate through test examples
     store test cost
plot the training and test costs
save the weights on disk
```

Lab Task 1 – Activation Functions _____

After an input is propagated through the network weights, the resulting output is "activated" by applying a nonlinear function. Some of these activation functions are given as follows:

$$\text{Sigmoid}(z) = 1 / (1 + e^{-z})$$

$$\text{ReLU}(z) = \max(0,z)$$

$$\text{dReLU}(z) = \begin{cases} 0, & z < 0 \\ 1, & z > 0 \end{cases}$$

Write the code for each of the given activation functions. Call each function with any value for z and take screenshots of your work showing your printed output. Provide the code and all relevant screenshots.

Code

```
# Task 1: Activation Functions
# Implementing multiple activation functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

def dReLU(x):
    return np.where(x > 0, 1, 0)

# Testing each funtion's output for multiple values of z
z = [0, -5, 1.45, 120, 457]

for value in z:
    print()
    print(f"Value: {value}")
    print(f"Sigmoid: {sigmoid(value)}")
```

```
    print(f"ReLU: {ReLU(value)}")
    print(f"dReLU: {dReLU(value)}")
    print()
```

Output Console

```
Value: 0
Sigmoid: 0.5
ReLU: 0
dReLU: 0


Value: -5
Sigmoid: 0.0066928509242848554
ReLU: 0
dReLU: 0
```

```
Value: 1.45
Sigmoid: 0.8099984339846871
ReLU: 1.45
dReLU: 1


Value: 120
Sigmoid: 1.0
ReLU: 120
dReLU: 1
```

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

Lab 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 X_{train} ($n \times m_{train}$), X_{test} ($n \times m_{test}$)
- Initialize all corresponding labels as array Y_{train} ($1 \times m_{train}$), Y_{test} ($1 \times m_{test}$)
- Initialize remaining layers Z_1, A_1, Z_2, A_2, Z_3, A_3 as zero arrays with appropriate sizes
- Initialize derivative layers dZ_1, dZ_2, dZ_3, dZ_{1cache}, dZ_{2cache}, dZ_{3cache} as zero arrays with appropriate sizes
- Initialize weight matrices W_1, B_1, W_2, B_2, W_3, B_3 with random numbers
- Initialize weight gradient matrices dW_1, dB_1, dW_2, dB_2, dW_3, dB_3 as zero arrays

Code

```
# Task 2: Matrix Initializations
from sklearn.model_selection import train_test_split

# Splitting the dataset
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2,
random_state=42)

# Rearranging X and Y matrix dimensions
X_train = X_train.T
Y_train = Y_train.reshape(1, Y_train.shape[0])
X_test = X_test.T
Y_test = Y_test.reshape(1, Y_test.shape[0])

# Extracting index values of train and test dataset
n = 3 # Number of features of input layer
```

```
m_train = np.array(X_train.shape[1])
m_test = np.array(X_test.shape[1])

# Hidden Layer 1 variables
n_hl1 = 2 # Neurons in 1st layer
W1 = np.random.randn(n_hl1, n) # Weight Matrix
B1 = np.random.randn(n_hl1, 1) # Broadcasting
dW1 = np.zeros((n_hl1, n))     # Derivative Layers
dB1 = np.zeros((n_hl1, 1))

# Hidden layer 2 variables
n_hl2 = 3  # Neurons in 2nd layer
W2 = np.random.randn(n_hl2, n_hl1) # Weight Matrix
B2 = np.random.randn(n_hl2, 1)     # Broadcasting
dW2 = np.zeros((n_hl2, n_hl1))     # Derivative Layers
dB2 = np.zeros((n_hl2, 1))

# Output layer variables
W3 = np.random.randn(1, n_hl2) # Weight Matrix
B3 = np.random.randn(1, 1)     # Broadcasting
dW3 = np.zeros((1, n_hl2))     # Derivative Layers
dB3 = np.zeros((1, 1))

# Remaining Derivative layers
dZ1 = np.zeros((n_hl1, m_train))
dZ1_cache = np.zeros((n_hl1, m_train))

dZ2 = np.zeros((n_hl2, m_train))
dZ2_cache = np.zeros((n_hl2, m_train))

dZ3 = np.zeros((1, m_train))
dZ3_cache = np.zeros((1, m_train))

# Newron Outputs
Z1 = np.zeros((n_hl1, m_train))
Z2 = np.zeros((n_hl2, m_train))
Z3 = np.zeros((1, m_train))
```

```
# Activation Layers
A1 = np.zeros((n_hl1, m_train))
A2 = np.zeros((n_hl2, m_train))
A3 = np.zeros((1, m_train))
```

Lab Task 3 – Forward Propagation _____

Write the code for implementing forward propagation given as follows:

$Z_1 = W_1X + B_1$ $A_1 = ReLU(Z_1)$
$Z_2 = W_2A_1 + B_2$ $A_2 = ReLU(Z_2)$
$Z_3 = W_3A_2 + B_3$ $A_3 = Sigmoid(Z_3)$

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.

Code

```
# Task 3: Forward Propogation

def Forward_Propogation(Xtrain, Xtest, Ytrain, Ytest, w1, b1, w2, b2, w3,
b3):
  # Computation on training dataset
  Z1 = w1 @ Xtrain + b1
  A1 = ReLU(Z1)
  Z2 = w2 @ A1 + b2
  A2 = ReLU(Z2)
  Z3 = w3 @ A2 + b3
  A3 = sigmoid(Z3)

  # Computation on test dataset
  Z1_test = w1 @ Xtest + b1
  A1_test = ReLU(Z1_test)
  Z2_test = w2 @ A1_test + b2
  A2_test = ReLU(Z2_test)
  Z3_test = w3 @ A2_test + b3
  A3_test = sigmoid(Z3_test)

  # Computing cost for training and test
  cost_train = -1 / m_train * np.sum(Ytrain * np.log(A3) + (1 - Ytrain) *
np.log(1 - A3))
```

```
  cost_test = -1 / m_test * np.sum(Ytest * np.log(A3_test) + (1 - Ytest) *
np.log(1 - A3_test))

  return cost_train, cost_test, Z1, A1, Z2, A2, Z3, A3, Z1_test, A1_test,
Z2_test, A2_test, Z3_test, A3_test

# calling function
cost_train, cost_test, Z1, A1, Z2, A2, Z3, A3, Z1_test, A1_test, Z2_test,
A2_test, Z3_test, A3_test = Forward_Propogation(X_train, X_test, Y_train,
Y_test, W1, B1, W2, B2, W3, B3)

# Printing the costs
print(f"Training Cost: {cost_train}")
print(f"Test Cost: {cost_test}")
```

Output Console

```
Training Cost: 2.3554249819674595
Test Cost: 2.4146548938224157
```

Lab Task 4 – Backward Propagation _____

Write the code for implementing backward propagation given as follows:

$dZ_{3cache} = dZ_3$
$dZ_3 = A_3 - Y_{train}$
$dW_3 = \frac{1}{m}(dZ_{3cache} \cdot A_2^T)$
dB_3 = sum dZ_{3cache} across training examples axis

$dZ_{2cache} = dZ_2$
$dZ_2 = (W_3 \cdot dZ_3)$.* $dReLU(Z_2)$
$dW_2 = \frac{1}{m}(dZ_{2cache} \cdot A_1^T)$
dB_2 = sum dZ_{2cache} across training examples axis

$dZ_{1cache} = dZ_1$
$dZ_1 = (W_2 \cdot dZ_2)$.* $dReLU(Z_1)$
$dW_1 = \frac{1}{m}(dZ_{1cache} \cdot X^T)$
dB_1 = sum dZ_{1cache} 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 dW_1, dB_1, dW_2, dB_2, dW_3, dB_3 are found, the weights can be updated using gradient descent:

$$W_L = W_L - \alpha \frac{\partial J}{\partial W_L} = W_L - \alpha\, dW_L$$

$$B_L = B_L - \alpha \frac{\partial J}{\partial B_L} = B_L - \alpha\, dB_L$$

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.

Code

```
def Backward_Propogation(Xtrain, Xtest, Ytrain, Ytest, w1, b1, w2, b2, w3,
b3, alpha):

    # Forward pass
    cost_train, cost_test, Z1, A1, Z2, A2, Z3, A3, Z1_test, A1_test,
Z2_test, A2_test, Z3_test, A3_test = Forward_Propogation(

                              Xtrain, Xtest,

                              Ytrain, Ytest,

                              w1, b1, w2, b2, w3, b3

                              )

    # Computing Backward Propagation
    dZ3 = A3 - Ytrain
    dZ3_cache = dZ3
    dW3 = (1/m_train) * (dZ3_cache @ A2.T)
    dB3 = (1/m_train) * np.sum(dZ3_cache, axis=1, keepdims=True)

    dZ2 = (w3.T @ dZ3) * dReLU(Z2)
    dZ2_cache = dZ2
    dW2 = (1/m_train) * (dZ2_cache @ A1.T)
    dB2 = (1/m_train) * np.sum(dZ2_cache, axis=1, keepdims=True)

    dZ1 = (w2.T @ dZ2) * dReLU(Z1)
    dZ1_cache = dZ1
```

```
    dW1 = (1/m_train) * (dZ1_cache @ Xtrain.T)
    dB1 = (1/m_train) * np.sum(dZ1_cache, axis=1, keepdims=True)

    # Updating weights
    w1 -= alpha * dW1
    b1 -= alpha * dB1
    w2 -= alpha * dW2
    b2 -= alpha * dB2
    w3 -= alpha * dW3
    b3 -= alpha * dB3

    return w1, b1, w2, b2, w3, b3, dW1, dB1, dW2, dB2, dW3, dB3, dZ1, dZ2,
dZ3, cost_train, cost_test

# Calling Backward Propagation
W1, B1, W2, B2, W3, B3, dW1, dB1, dW2, dB2, dW3, dB3, dZ1, dZ2, dZ3,
train_cost, test_cost = Backward_Propogation(

                X_train, X_test,

                Y_train, Y_test,

                W1, B1,

                W2, B2,

                W3, B3,

                alpha = 0.01

                )

# Printing results
print("\nDerivatives")
print("dZ1:", dZ1)
print("dW1:", dW1)
print("dB1:", dB1)
```

```
print("\ndZ2:", dZ2)
print("dW2:", dW2)
print("dB2:", dB2)

print("\ndZ3:", dZ3)
print("dW3:", dW3)
print("dB3:", dB3)

print("\nUpdated Weights")
print("W1:", W1)
print("B1:", B1)
print("W2:", W2)
print("B2:", B2)
print("W3:", W3)
print("B3:", B3)
```

Output Console

```
Derivatives
dZ1: [[-0.          0.          1.33530204 ... -0.09111272 -0.07733189
  -0.10027335]
 [ 0.         -0.00312647 -0.71785044 ...  0.04773923  0.04157316
   0.        ]]
dW1: [[ 1.37459448  3.53993174 -0.76289246]
 [-0.76891934 -1.85176057  0.42281789]]
dB1: [[ 0.55270781]
 [-0.31510037]]

dZ2: [[ 0.00000000e+00 -2.41591806e-03 -0.00000000e+00 ...  0.00000000e+00
   0.00000000e+00  0.00000000e+00]
 [-3.37173028e-01  0.00000000e+00  2.33081698e+00 ... -1.97664455e-01
  -1.34985551e-01 -2.17537979e-01]
 [-3.55065900e-02  4.85063305e-04  0.00000000e+00 ... -2.08153979e-02
  -0.00000000e+00 -2.29082138e-02]]
dW2: [[-0.02927352 -0.07591779]
 [ 5.22729605  4.74991903]
 [ 0.02859384  0.05146399]]
dB2: [[-0.02092796]
 [ 0.89714884]
 [ 0.00967851]]
```

```
dZ3: [[-0.14450902  0.00197417  0.99896508 ... -0.08471703 -0.05785347
  -0.09323462]]
dW3: [[0.01720032 0.9959093  0.04132167]]
dB3: [[0.38526246]]

Updated Weights
W1: [[ 0.67708983  0.60058668 -0.17120765]
 [ 1.3991838   0.1233063  -0.92269951]]
B1: [[-0.57607273]
 [-0.35883292]]
W2: [[-1.97363347  1.42159359]
 [ 0.52061717 -0.35548152]
 [-1.06331109  0.63064913]]
B2: [[-1.38710673]
 [ 0.87372703]
 [ 0.8333698 ]]
W3: [[-1.22393632  2.3232726   0.24529179]]
B3: [[-0.48984609]]
```

Lab 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.

Code

```
# Task 5: Training and Testing

# Creating main function
def main(Xtrain, Xtest, Ytrain, Ytest, w1, b1, w2, b2, w3, b3, epochs,
alpha):
  train_cost_history = []
  test_cost_history = []

  for epoch in range(epochs):
    # Calling Backward Propagation
    w1, b1, w2, b2, w3, b3, dw1, db1, dw2, db2, dw3, db3, dz1, dz2, dz3,
train_cost, test_cost= Backward_Propogation(

                      Xtrain, Xtest, Ytrain, Ytest,

                      w1, b1, w2, b2, w3, b3,

                      alpha

                      )
    train_cost_history.append(train_cost)
    test_cost_history.append(test_cost)

  # Plotting training and test cost vs epochs
  plt.plot(train_cost_history, label="Training Cost")
  plt.plot(test_cost_history, label="Test Cost")
  plt.xlabel("Epochs")
  plt.ylabel("Cost")
  plt.legend()
  plt.show()

# Calling main
main(X_train, X_test, Y_train, Y_test, W1, B1, W2, B2, W3, B3, epochs =
5000, alpha = 0.01)
```

Output Console