vjy3u9nei

January 25, 2025

1 Assignment 1

Question 1: [3 Marks]

Explain the functionality of a perceptron with its mathematical representation. Provide:

The formula for the perceptron model.

A detailed explanation of each term in the formula.

Question 2: [4 Marks]

Build and train a neural network to solve the XOR problem using any deep learning library (e.g., TensorFlow or PyTorch):

Define the 3-bit XOR dataset and preprocess it for model training. [1 Mark]

Design a neural network with one hidden layer. Clearly specify the architecture: [input, hidden, output]. [2 Marks] Train the network on the dataset and evaluate its performance, reporting the accuracy. [1 Mark]

Question 3: [5 Marks]

Implement a neural network for the Pima Indians Diabetes Dataset:

Design a neural network with the following architecture: [Input, hidden1(8), hidden2(4), hidden3(4), output]. [2 Marks]

Train the model on the diabetes dataset and record its performance. [2 Marks]

Provide a schematic representation of the neural network with its layers and activation functions. [1 Mark]

1.1 Question 1: [3 Marks]

Explain the functionality of a perceptron with its mathematical representation. Prov de

1 The formula for the perceptr mode

2. . A detailed explanation of each term in the formula.

1.1.1 1- The formula for the perceptron model.

A perceptron is a simple artificial neuron designed for binary classification. It maps inputs to an output based on a weighted sum and a threshold. Below is the explanation of its functionality, formula, and the detailed explanation of each term. The perceptron is a fundamental building block of neural networks used for binary classification. It processes inputs and computes an output based on a linear combination of weights, inputs, and a bias term, followed by an activation function.

The perceptron works on the following principle:

–> Takes weighted sums of input features.

–> Passes the result through an activation function (typically a step function).

–> Outputs a binary decision (0 or 1).

Mathematical Representation of a Perceptron The perceptron model can be described using the following formula:

y=f(i=1 n wi xi +b)

Where:

–> y: The output of the perceptron, which is either 0 or 1.

–> f(z): The activation function, which applies a step function

–> wi=[w1 ,w 2 ,...,w n]: The weights of the model assosiated with i-th input feature.

–> xi=[x1,x 2,...,x n]: i-th input features.

–> b: The bias term, which shifts the decision boundary.

–> i=1 n w i x i : Weighted sum of inputs.

1.1.2 2- A detailed explanation of each term in the formula.

1. Input Features ( ):

   These are the feature values of the input data. For example, if the input is a vector like [1, 2, 3] each represents a specific attribute or feature.

2. Weights ( ):

   Weights represent the importance or contribution of each input feature to the output. The weights are adjusted during training to minimize classification errors.

3. Weighted Sum (i=1 n wixi):

   This is the linear combination of inputs and their corresponding weights. It determines the overall influence of the inputs before applying the activation function.

4. Bias ( ):

   The bias term shifts the decision boundary away from the origin, allowing more flexibility in classification. Without bias, the decision boundary always passes through the origin.

5. Activation Function ( ):

   The activation function determines the output of the perceptron. In a simple perceptron, the step function is used:

   ( )= {1,if 0

```
0,if  <0
```

Here, = i=1 n wixi+b)

b is the input to the activation function.

Summary:-

The perceptron essentially calculates a weighted sum of inputs, adjusts it using a bias, and applies an activation function to make a classification decision. Its functionality can be visualized geometrically as finding a hyperplane (decision boundary) that separates the data into two classes in a feature space.

1.2 Question 2: [4 Marks]

Build and train a neural network to solve the XOR problem using any deep learning library (e.g., TensorFlow or PyTorch):

1. Define the 3-bit XOR dataset and preprocess it for model training. [1 Mark]
2. Design a neural network with one hidden layer. Clearly specify the architecture: [input, hidden, output]. [2 Marks]
3. Train the network on the dataset and evaluate its performance, reporting the accuracy. [1 Mark]

1.2.1 1. Define the 3-bit XOR dataset and preprocess it for model training. [1 Mark]

[3]:
```python
# Importing required packages
import numpy as np
import tensorflow as tnf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
# Defining 3-bit XOR dataset
x = np.array([
    [0, 0, 0],
    [0, 0, 1],
    [0, 1, 0],
    [0, 1, 1],
    [1, 0, 0],
    [1, 0, 1],
    [1, 1, 0],
    [1, 1, 1]
])

# XOR output (1 if odd number of 1s, otherwise 0)
y = np.array([[0], [1], [1], [0], [1], [0], [0], [1]])

# Normalize the inputs (not strictly necessary here since inputs are binary)
x = x.astype("float32")
```

```
y = y.astype("float32")
```

[4]:
```
x
```

[4]:
```
array([[0., 0., 0.],
       [0., 0., 1.],
       [0., 1., 0.],
       [0., 1., 1.],
       [1., 0., 0.],
       [1., 0., 1.],
       [1., 1., 0.],
       [1., 1., 1.]], dtype=float32)
```

[5]:
```
y
```

[5]:
```
array([[0.],
       [1.],
       [1.],
       [0.],
       [1.],
       [0.],
       [0.],
       [1.]], dtype=float32)
```

1.2.2 2. Design a neural network with one hidden layer. Clearly specify the architecture: [input, hidden, output]. [2 Marks]

Designing a neural network with one hidden layer:

–> Input layer: 3 neurons (corresponding to the 3 input bits).

–> Hidden layer: 4 neurons with ReLU activation.

–> Output layer: 1 neuron with sigmoid activation (for binary classification).

[6]:
```
# Designing  the neural network
model = Sequential([
    Dense(4, activation="relu", input_shape=(3,)),  # Hidden layer with 4␣
  ↪neurons
    Dense(1, activation="sigmoid")                  # Output layer with 1 neuron
])

# Compile the model
model.compile(optimizer="adam", loss="binary_crossentropy",␣
  ↪metrics=["accuracy"])
```

```
C:\Users\ASUS\miniconda3\Lib\site-packages\keras\src\layers\core\dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
```

```
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

1.2.3 3. Train the network on the dataset and evaluate its performance, reporting the accuracy. [1 Mark]

[9]:
```
# Training the model
model.fit(x, y, epochs=100, verbose=0, batch_size=8)

# Evaluate the model
loss, accuracy = model.evaluate(x, y, verbose=0)

print(f"Final Model Accuracy: {accuracy * 100:.2f}%")
```

```
Final Model Accuracy: 75.00%
```

1.3 Question 3: [5 Marks]

Implement a neural network for the Pima Indians Diabetes Dataset: 1. Design a neural network with the following architecture: [Input, hidden1(8), hidden2(4), hidden3(4), output]. [2 Marks] 2. Train the model on the diabetes dataset and record its performance. [2 Marks] 3. Provide a schematic representation of the neural network with its layers and activation functions. [1 Mark]

1.3.1 1. Design a neural network with the following architecture: [Input, hidden1(8), hidden2(4), hidden3(4), output]. [2 Marks]

[10]:
```
# Importing required packages
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import warnings as war
war.filterwarnings("ignore")
```

[11]:
```
# Step 1:- Load the Diabetes Dataset

dataSetPath="C:\\Users\\ASUS\\jupyterworkspace\\Assignment & Mini␣
 ↪Project\Module_06_Deep␣
 ↪Learning\\Deep-Learning-Assignment01-neuralnetworkdesignon_diabetesdataSet\\diabetes.
 ↪csv"
dataSetRead=pd.read_csv(dataSetPath)
```

[12]:
```
# Displaying first 5 records to confirming data loading
print("*****************************************************Displaying below␣
 ↪first 5 records*********************************************************")
dataSetRead.head()
```

```
*****************************************************Displaying below first 5
records*********************************************************
```

```
[12]:    Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
      0            6      148             72             35        0  33.6
      1            1       85             66             29        0  26.6
      2            8      183             64              0        0  23.3
      3            1       89             66             23       94  28.1
      4            0      137             40             35      168  43.1

         DiabetesPedigreeFunction  Age  Outcome
      0                     0.627   50        1
      1                     0.351   31        0
      2                     0.672   32        1
      3                     0.167   21        0
      4                     2.288   33        1
```

```python
[13]: # Displaying last 5 records to confirming data loading
      print("*****************************************************Displaying below␣
       ↪last 5 records**********************************************************")
      dataSetRead.tail()
```

```
*****************************************************Displaying below last 5
records**********************************************************

[13]:      Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
      763           10      101             76             48      180  32.9
      764            2      122             70             27        0  36.8
      765            5      121             72             23      112  26.2
      766            1      126             60              0        0  30.1
      767            1       93             70             31        0  30.4

           DiabetesPedigreeFunction  Age  Outcome
      763                     0.171   63        0
      764                     0.340   27        0
      765                     0.245   30        0
      766                     0.349   47        1
      767                     0.315   23        0
```

```python
[14]: # Displaying all records to confirming data loading
      print("*****************************************************Displaying below␣
       ↪all records**********************************************************")
      dataSetRead
```

```
*****************************************************Displaying below all
records**********************************************************

[14]:      Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
      0              6      148             72             35        0  33.6
      1              1       85             66             29        0  26.6
```

```
2            8      183           64            0        0  23.3
3            1       89           66           23       94  28.1
4            0      137           40           35      168  43.1
..         ...      ...          ...          ...      ...   ...
763         10      101           76           48      180  32.9
764          2      122           70           27        0  36.8
765          5      121           72           23      112  26.2
766          1      126           60            0        0  30.1
767          1       93           70           31        0  30.4

     DiabetesPedigreeFunction  Age  Outcome
0                       0.627   50        1
1                       0.351   31        0
2                       0.672   32        1
3                       0.167   21        0
4                       2.288   33        1
..                        ...  ...      ...
763                     0.171   63        0
764                     0.340   27        0
765                     0.245   30        0
766                     0.349   47        1
767                     0.315   23        0

[768 rows x 9 columns]
```

[15]:

```python
# Spliting data into features (X) and target (y)
X = dataSetRead.iloc[:, :-1].values  # All columns except the last one
y = dataSetRead.iloc[:, -1].values   # Last column as the target variable

# Spliting the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Standardizing the features (important for neural networks)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

1.3.2 Design the Neural Network Architecture

The neural network will have the following structure:

–> Input Layer: Matches the number of features in the dataset (8 features).

–> Hidden Layer 1: 8 neurons, ReLU activation.

–> Hidden Layer 2: 4 neurons, ReLU activation.

–> Hidden Layer 3: 4 neurons, ReLU activation.

–> Output Layer: 1 neuron, sigmoid activation (for binary classification).

[16]:
```python
# Defining the model
model = Sequential([
    Dense(8, activation='relu', input_shape=(X.shape[1],)),  # Hidden layer 1
    Dense(4, activation='relu'),                             # Hidden layer 2
    Dense(4, activation='relu'),                             # Hidden layer 3
    Dense(1, activation='sigmoid')                           # Output layer
])

# Compiling the model
model.compile(optimizer='adam', loss='binary_crossentropy',␣
↪metrics=['accuracy'])
```

1.3.3 2.Train the model on the diabetes dataset and record its performance. [2 Marks]

[19]:
```python
# Training the model
history = model.fit(X_train, y_train, epochs=100, batch_size=16,␣
↪validation_split=0.2, verbose=1)
```

```
Epoch 1/100
31/31            0s 14ms/step -
accuracy: 0.8178 - loss: 0.4344 - val_accuracy: 0.7154 - val_loss: 0.5904
Epoch 2/100
31/31            0s 11ms/step -
accuracy: 0.8114 - loss: 0.4465 - val_accuracy: 0.7073 - val_loss: 0.5918
Epoch 3/100
31/31            0s 11ms/step -
accuracy: 0.8166 - loss: 0.4365 - val_accuracy: 0.7154 - val_loss: 0.5891
Epoch 4/100
31/31            0s 12ms/step -
accuracy: 0.7823 - loss: 0.4601 - val_accuracy: 0.7073 - val_loss: 0.5904
Epoch 5/100
31/31            0s 12ms/step -
accuracy: 0.7979 - loss: 0.4568 - val_accuracy: 0.7154 - val_loss: 0.5917
Epoch 6/100
31/31            0s 12ms/step -
accuracy: 0.8032 - loss: 0.4651 - val_accuracy: 0.7073 - val_loss: 0.5930
Epoch 7/100
31/31            0s 11ms/step -
accuracy: 0.7863 - loss: 0.4556 - val_accuracy: 0.7073 - val_loss: 0.5923
Epoch 8/100
31/31            0s 11ms/step -
accuracy: 0.8275 - loss: 0.4117 - val_accuracy: 0.7073 - val_loss: 0.5919
Epoch 9/100
31/31            0s 11ms/step -
accuracy: 0.8075 - loss: 0.4422 - val_accuracy: 0.7073 - val_loss: 0.5928
```

```
Epoch 10/100
31/31            0s 11ms/step -
accuracy: 0.7837 - loss: 0.4869 - val_accuracy: 0.7073 - val_loss: 0.5936
Epoch 11/100
31/31            0s 11ms/step -
accuracy: 0.8096 - loss: 0.4419 - val_accuracy: 0.7073 - val_loss: 0.5919
Epoch 12/100
31/31            0s 11ms/step -
accuracy: 0.7951 - loss: 0.4658 - val_accuracy: 0.7154 - val_loss: 0.5928
Epoch 13/100
31/31            0s 11ms/step -
accuracy: 0.7642 - loss: 0.4896 - val_accuracy: 0.7154 - val_loss: 0.5943
Epoch 14/100
31/31            0s 12ms/step -
accuracy: 0.7934 - loss: 0.4803 - val_accuracy: 0.7236 - val_loss: 0.5955
Epoch 15/100
31/31            0s 11ms/step -
accuracy: 0.8023 - loss: 0.4292 - val_accuracy: 0.7154 - val_loss: 0.5950
Epoch 16/100
31/31            0s 11ms/step -
accuracy: 0.8325 - loss: 0.4128 - val_accuracy: 0.7154 - val_loss: 0.5973
Epoch 17/100
31/31            0s 12ms/step -
accuracy: 0.8117 - loss: 0.4287 - val_accuracy: 0.7154 - val_loss: 0.5936
Epoch 18/100
31/31            0s 13ms/step -
accuracy: 0.8203 - loss: 0.4257 - val_accuracy: 0.7154 - val_loss: 0.5957
Epoch 19/100
31/31            0s 11ms/step -
accuracy: 0.7857 - loss: 0.4610 - val_accuracy: 0.7154 - val_loss: 0.5965
Epoch 20/100
31/31            0s 11ms/step -
accuracy: 0.8051 - loss: 0.4488 - val_accuracy: 0.7073 - val_loss: 0.5946
Epoch 21/100
31/31            0s 11ms/step -
accuracy: 0.8060 - loss: 0.4448 - val_accuracy: 0.7154 - val_loss: 0.5967
Epoch 22/100
31/31            0s 11ms/step -
accuracy: 0.8347 - loss: 0.4212 - val_accuracy: 0.7073 - val_loss: 0.5959
Epoch 23/100
31/31            0s 13ms/step -
accuracy: 0.8089 - loss: 0.4315 - val_accuracy: 0.7154 - val_loss: 0.5994
Epoch 24/100
31/31            0s 11ms/step -
accuracy: 0.8092 - loss: 0.4520 - val_accuracy: 0.7154 - val_loss: 0.5973
Epoch 25/100
31/31            1s 15ms/step -
accuracy: 0.8107 - loss: 0.4414 - val_accuracy: 0.7154 - val_loss: 0.5996
```

```
Epoch 26/100
31/31            0s 13ms/step -
accuracy: 0.7956 - loss: 0.4661 - val_accuracy: 0.7154 - val_loss: 0.6011
Epoch 27/100
31/31            0s 11ms/step -
accuracy: 0.8024 - loss: 0.4437 - val_accuracy: 0.7154 - val_loss: 0.5984
Epoch 28/100
31/31            0s 12ms/step -
accuracy: 0.7834 - loss: 0.4834 - val_accuracy: 0.7154 - val_loss: 0.6000
Epoch 29/100
31/31            0s 12ms/step -
accuracy: 0.8120 - loss: 0.4428 - val_accuracy: 0.7154 - val_loss: 0.5989
Epoch 30/100
31/31            0s 13ms/step -
accuracy: 0.8250 - loss: 0.4379 - val_accuracy: 0.7154 - val_loss: 0.6008
Epoch 31/100
31/31            0s 12ms/step -
accuracy: 0.8148 - loss: 0.4397 - val_accuracy: 0.7154 - val_loss: 0.5998
Epoch 32/100
31/31            0s 11ms/step -
accuracy: 0.8158 - loss: 0.4360 - val_accuracy: 0.7154 - val_loss: 0.6011
Epoch 33/100
31/31            0s 11ms/step -
accuracy: 0.8303 - loss: 0.4261 - val_accuracy: 0.7154 - val_loss: 0.6027
Epoch 34/100
31/31            0s 11ms/step -
accuracy: 0.7862 - loss: 0.4672 - val_accuracy: 0.7154 - val_loss: 0.6022
Epoch 35/100
31/31            0s 11ms/step -
accuracy: 0.8494 - loss: 0.4047 - val_accuracy: 0.7154 - val_loss: 0.6024
Epoch 36/100
31/31            0s 12ms/step -
accuracy: 0.7958 - loss: 0.4469 - val_accuracy: 0.7154 - val_loss: 0.6038
Epoch 37/100
31/31            0s 11ms/step -
accuracy: 0.8039 - loss: 0.4458 - val_accuracy: 0.7154 - val_loss: 0.6031
Epoch 38/100
31/31            0s 12ms/step -
accuracy: 0.8217 - loss: 0.4128 - val_accuracy: 0.7154 - val_loss: 0.6026
Epoch 39/100
31/31            0s 11ms/step -
accuracy: 0.8128 - loss: 0.4187 - val_accuracy: 0.7154 - val_loss: 0.6052
Epoch 40/100
31/31            0s 11ms/step -
accuracy: 0.8401 - loss: 0.4218 - val_accuracy: 0.7154 - val_loss: 0.6033
Epoch 41/100
31/31            0s 11ms/step -
accuracy: 0.8207 - loss: 0.4191 - val_accuracy: 0.7236 - val_loss: 0.6068
```

```
Epoch 42/100
31/31            0s 12ms/step -
accuracy: 0.8217 - loss: 0.4147 - val_accuracy: 0.7154 - val_loss: 0.6015
Epoch 43/100
31/31            1s 15ms/step -
accuracy: 0.8174 - loss: 0.4234 - val_accuracy: 0.7154 - val_loss: 0.6034
Epoch 44/100
31/31            0s 12ms/step -
accuracy: 0.8289 - loss: 0.4352 - val_accuracy: 0.7236 - val_loss: 0.6070
Epoch 45/100
31/31            0s 11ms/step -
accuracy: 0.8087 - loss: 0.4399 - val_accuracy: 0.7154 - val_loss: 0.6049
Epoch 46/100
31/31            0s 11ms/step -
accuracy: 0.7805 - loss: 0.4829 - val_accuracy: 0.7154 - val_loss: 0.6039
Epoch 47/100
31/31            0s 10ms/step -
accuracy: 0.8155 - loss: 0.4306 - val_accuracy: 0.7154 - val_loss: 0.6088
Epoch 48/100
31/31            0s 11ms/step -
accuracy: 0.7850 - loss: 0.4766 - val_accuracy: 0.7154 - val_loss: 0.6063
Epoch 49/100
31/31            1s 12ms/step -
accuracy: 0.8020 - loss: 0.4443 - val_accuracy: 0.7236 - val_loss: 0.6055
Epoch 50/100
31/31            0s 11ms/step -
accuracy: 0.8080 - loss: 0.4374 - val_accuracy: 0.7154 - val_loss: 0.6088
Epoch 51/100
31/31            0s 11ms/step -
accuracy: 0.8057 - loss: 0.4460 - val_accuracy: 0.7154 - val_loss: 0.6090
Epoch 52/100
31/31            0s 12ms/step -
accuracy: 0.8132 - loss: 0.4368 - val_accuracy: 0.7236 - val_loss: 0.6076
Epoch 53/100
31/31            0s 11ms/step -
accuracy: 0.7809 - loss: 0.4558 - val_accuracy: 0.7154 - val_loss: 0.6098
Epoch 54/100
31/31            0s 13ms/step -
accuracy: 0.8152 - loss: 0.4323 - val_accuracy: 0.7236 - val_loss: 0.6092
Epoch 55/100
31/31            0s 11ms/step -
accuracy: 0.8138 - loss: 0.4459 - val_accuracy: 0.7154 - val_loss: 0.6137
Epoch 56/100
31/31            0s 11ms/step -
accuracy: 0.7851 - loss: 0.4471 - val_accuracy: 0.7236 - val_loss: 0.6126
Epoch 57/100
31/31            0s 11ms/step -
accuracy: 0.8389 - loss: 0.4017 - val_accuracy: 0.7154 - val_loss: 0.6160
```

```
Epoch 58/100
31/31            0s 11ms/step -
accuracy: 0.8213 - loss: 0.4298 - val_accuracy: 0.7073 - val_loss: 0.6192
Epoch 59/100
31/31            0s 12ms/step -
accuracy: 0.7991 - loss: 0.4356 - val_accuracy: 0.7154 - val_loss: 0.6167
Epoch 60/100
31/31            0s 14ms/step -
accuracy: 0.7764 - loss: 0.4654 - val_accuracy: 0.7236 - val_loss: 0.6188
Epoch 61/100
31/31            0s 11ms/step -
accuracy: 0.8109 - loss: 0.4222 - val_accuracy: 0.7236 - val_loss: 0.6191
Epoch 62/100
31/31            0s 11ms/step -
accuracy: 0.8115 - loss: 0.4322 - val_accuracy: 0.7236 - val_loss: 0.6234
Epoch 63/100
31/31            0s 11ms/step -
accuracy: 0.8212 - loss: 0.4080 - val_accuracy: 0.7073 - val_loss: 0.6259
Epoch 64/100
31/31            0s 12ms/step -
accuracy: 0.8328 - loss: 0.4136 - val_accuracy: 0.7236 - val_loss: 0.6225
Epoch 65/100
31/31            0s 11ms/step -
accuracy: 0.8306 - loss: 0.4078 - val_accuracy: 0.6992 - val_loss: 0.6270
Epoch 66/100
31/31            0s 11ms/step -
accuracy: 0.8211 - loss: 0.4183 - val_accuracy: 0.7236 - val_loss: 0.6230
Epoch 67/100
31/31            0s 11ms/step -
accuracy: 0.8231 - loss: 0.4017 - val_accuracy: 0.7236 - val_loss: 0.6253
Epoch 68/100
31/31            0s 11ms/step -
accuracy: 0.7960 - loss: 0.4539 - val_accuracy: 0.7236 - val_loss: 0.6241
Epoch 69/100
31/31            0s 11ms/step -
accuracy: 0.8266 - loss: 0.4142 - val_accuracy: 0.6992 - val_loss: 0.6299
Epoch 70/100
31/31            0s 11ms/step -
accuracy: 0.8028 - loss: 0.4208 - val_accuracy: 0.7236 - val_loss: 0.6302
Epoch 71/100
31/31            0s 11ms/step -
accuracy: 0.7935 - loss: 0.4547 - val_accuracy: 0.7236 - val_loss: 0.6310
Epoch 72/100
31/31            0s 11ms/step -
accuracy: 0.8172 - loss: 0.4122 - val_accuracy: 0.7154 - val_loss: 0.6309
Epoch 73/100
31/31            0s 11ms/step -
accuracy: 0.8267 - loss: 0.3965 - val_accuracy: 0.7154 - val_loss: 0.6319
```

```
Epoch 74/100
31/31            0s 11ms/step -
accuracy: 0.8045 - loss: 0.4540 - val_accuracy: 0.7073 - val_loss: 0.6334
Epoch 75/100
31/31            0s 11ms/step -
accuracy: 0.8018 - loss: 0.4327 - val_accuracy: 0.7154 - val_loss: 0.6319
Epoch 76/100
31/31            0s 12ms/step -
accuracy: 0.8055 - loss: 0.4294 - val_accuracy: 0.7073 - val_loss: 0.6315
Epoch 77/100
31/31            0s 14ms/step -
accuracy: 0.8105 - loss: 0.4329 - val_accuracy: 0.7073 - val_loss: 0.6358
Epoch 78/100
31/31            0s 13ms/step -
accuracy: 0.7996 - loss: 0.4484 - val_accuracy: 0.7154 - val_loss: 0.6336
Epoch 79/100
31/31            0s 12ms/step -
accuracy: 0.7919 - loss: 0.4350 - val_accuracy: 0.7154 - val_loss: 0.6392
Epoch 80/100
31/31            0s 11ms/step -
accuracy: 0.8316 - loss: 0.4087 - val_accuracy: 0.7236 - val_loss: 0.6379
Epoch 81/100
31/31            0s 11ms/step -
accuracy: 0.8195 - loss: 0.4149 - val_accuracy: 0.7073 - val_loss: 0.6406
Epoch 82/100
31/31            0s 11ms/step -
accuracy: 0.8159 - loss: 0.4249 - val_accuracy: 0.7073 - val_loss: 0.6380
Epoch 83/100
31/31            0s 12ms/step -
accuracy: 0.8179 - loss: 0.4314 - val_accuracy: 0.7154 - val_loss: 0.6381
Epoch 84/100
31/31            0s 11ms/step -
accuracy: 0.7992 - loss: 0.4497 - val_accuracy: 0.7236 - val_loss: 0.6399
Epoch 85/100
31/31            0s 11ms/step -
accuracy: 0.8168 - loss: 0.4189 - val_accuracy: 0.7073 - val_loss: 0.6409
Epoch 86/100
31/31            0s 12ms/step -
accuracy: 0.8217 - loss: 0.4256 - val_accuracy: 0.7236 - val_loss: 0.6420
Epoch 87/100
31/31            0s 11ms/step -
accuracy: 0.8184 - loss: 0.4322 - val_accuracy: 0.7154 - val_loss: 0.6443
Epoch 88/100
31/31            0s 12ms/step -
accuracy: 0.8323 - loss: 0.4240 - val_accuracy: 0.7073 - val_loss: 0.6448
Epoch 89/100
31/31            0s 11ms/step -
accuracy: 0.8012 - loss: 0.4487 - val_accuracy: 0.7236 - val_loss: 0.6424
```

```
Epoch 90/100
31/31            0s 11ms/step -
accuracy: 0.8314 - loss: 0.4042 - val_accuracy: 0.7154 - val_loss: 0.6420
Epoch 91/100
31/31            0s 11ms/step -
accuracy: 0.8482 - loss: 0.3900 - val_accuracy: 0.7073 - val_loss: 0.6437
Epoch 92/100
31/31            0s 11ms/step -
accuracy: 0.8512 - loss: 0.3928 - val_accuracy: 0.6992 - val_loss: 0.6441
Epoch 93/100
31/31            0s 11ms/step -
accuracy: 0.8342 - loss: 0.4150 - val_accuracy: 0.6992 - val_loss: 0.6478
Epoch 94/100
31/31            0s 12ms/step -
accuracy: 0.8241 - loss: 0.4259 - val_accuracy: 0.6911 - val_loss: 0.6473
Epoch 95/100
31/31            0s 11ms/step -
accuracy: 0.8175 - loss: 0.4272 - val_accuracy: 0.6992 - val_loss: 0.6486
Epoch 96/100
31/31            0s 13ms/step -
accuracy: 0.8077 - loss: 0.4512 - val_accuracy: 0.7073 - val_loss: 0.6455
Epoch 97/100
31/31            0s 11ms/step -
accuracy: 0.8320 - loss: 0.4116 - val_accuracy: 0.6992 - val_loss: 0.6483
Epoch 98/100
31/31            0s 11ms/step -
accuracy: 0.8142 - loss: 0.4541 - val_accuracy: 0.6992 - val_loss: 0.6526
Epoch 99/100
31/31            0s 12ms/step -
accuracy: 0.7873 - loss: 0.4529 - val_accuracy: 0.7154 - val_loss: 0.6491
Epoch 100/100
31/31            0s 11ms/step -
accuracy: 0.8501 - loss: 0.3974 - val_accuracy: 0.6911 - val_loss: 0.6543
```

[20]:
```python
# Evaluating the model
loss, accuracy = model.evaluate(X_test, y_test, verbose=0)

print(f"Test Loss: {loss:.4f}")
print(f"Model Accuracy: {accuracy * 100:.2f}%")
```

```
Test Loss: 0.7250
Model Accuracy: 73.38%
```

1.3.4 3. Provide a schematic representation of the neural network with its layers and activation functions. [1 Mark]

Below is the schematic representation of the network:

–> Input Layer (8 neurons)

Receives 8 input features.

–> Hidden Layer 1 (8 neurons, ReLU activation)

Applies the ReLU function to learn complex patterns.

–> Hidden Layer 2 (4 neurons, ReLU activation)

Further reduces dimensionality while preserving essential features.

–> Hidden Layer 3 (4 neurons, ReLU activation)

Extracts high-level patterns.

–> Output Layer (1 neuron, Sigmoid activation)

Outputs a probability value for binary classification.

[21]:
```
# Importing required packages
import matplotlib.pyplot as plt
import matplotlib.patches as patches

def draw_neural_network(ax, left, right, bottom, top, layer_sizes, activations):
    # Calculate vertical and horizontal spacing
    v_spacing = (top - bottom) / float(max(layer_sizes))
    h_spacing = (right - left) / float(len(layer_sizes) - 1)

    # Draw nodes and activation functions
    for n, layer_size in enumerate(layer_sizes):
        layer_top = v_spacing * (layer_size - 1) / 2. + (top + bottom) / 2.

        for m in range(layer_size):
            # Create circles for nodes
            circle = plt.Circle((n * h_spacing + left, layer_top - m *␣
↪v_spacing), v_spacing / 4.,
                                color='w', ec='k', zorder=4)
            ax.add_artist(circle)

            # Annotate activation functions
            if n < len(layer_sizes) - 1 and m == layer_size - 1:
                ax.text(n * h_spacing + left, layer_top - m * v_spacing -␣
↪v_spacing / 2, activations[n], fontsize=12, ha='center')

    # Draw edges between layers
    for n, (layer_size_a, layer_size_b) in enumerate(zip(layer_sizes[:-1],␣
↪layer_sizes[1:])):
        layer_top_a = v_spacing * (layer_size_a - 1) / 2. + (top + bottom) / 2.
        layer_top_b = v_spacing * (layer_size_b - 1) / 2. + (top + bottom) / 2.

        for m in range(layer_size_a):
            for o in range(layer_size_b):
```

```
                # Draw lines for edges
                line = plt.Line2D([n * h_spacing + left, (n + 1) * h_spacing +␣
↪left],
                                  [layer_top_a - m * v_spacing, layer_top_b - o␣
↪* v_spacing], c='k')
                ax.add_artist(line)

# Neural network architecture
layer_sizes = [8, 8, 4, 4, 1]
activations = ['ReLU', 'ReLU', 'ReLU', 'Sigmoid']

# Create figure and axis
fig, ax = plt.subplots(figsize=(12, 8))
ax.axis('on')

# Draw the neural network
draw_neural_network(ax, 0.1, 0.9, 0.1, 0.9, layer_sizes, activations)

# Annotate layer names
for i, layer_size in enumerate(layer_sizes):
    ax.text(i * (0.8 / (len(layer_sizes) - 1)) + 0.1, 0.95, f'Layer {i + 1}',␣
↪fontsize=16, ha='center')

plt.show()
```