**EXPERIMENT – 1**

**PYTHON PROGRAM THAT DEMONSTRATES THE USE OF THE ReLU ACTIVATION FUNCTION IN A BASIC NEURAL NETWORK**

**Aim: -** The aim of this program is to use the ReLU activation function in a basic neural network.

**Procedure: -**

import tensorflow as tf

from tensorflow.keras import layers, models

# Build a simple neural network with ReLU activation

model = models.Sequential([

layers.Dense(32, input\_shape=(10,), activation='relu'),

layers.Dense(1, activation='sigmoid')

])

# Display the model summary

model.summary()

**Output: -**

Model: "sequential"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense (Dense) (None, 32) 352

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 1) 33

=================================================================

Total params: 385

Trainable params: 385

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Result: -** Program executed successfully.

**EXPERIMENT – 2**

**SIMPLE PYTHON PROGRAM THAT CREATES A BASIC NEURON AND PERFORMS A FORWARD PASS**

**Aim: -** To create a neuron, performs a forward pass with random weights and bias, and displays the relevant information.

**Procedure: -**

import numpy as np

class Neuron:

def \_\_init\_\_(self, input\_size):

# Initialize weights and bias randomly

self.weights = np.random.rand(input\_size)

self.bias = np.random.rand(1)

def sigmoid(self, x):

# Sigmoid activation function

return 1 / (1 + np.exp(-x))

def forward(self, inputs):

# Perform a forward pass through the neuron

weighted\_sum = np.dot(inputs, self.weights) + self.bias

output = self.sigmoid(weighted\_sum)

return output

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

# Create a neuron with 3 input features

neuron = Neuron(input\_size=3)

# Input data for a single example

input\_data = np.array([0.5, 0.3, 0.2])

# Perform a forward pass through the neuron

output = neuron.forward(input\_data)

# Display the results

print("Input Data:", input\_data)

print("Weights:", neuron.weights)

print("Bias:", neuron.bias)

print("Weighted Sum:", np.dot(input\_data, neuron.weights) + neuron.bias)

print("Output after Sigmoid Activation:", output)

**Output:-**

Input Data: [0.5 0.3 0.2]

Weights: [0.46353624 0.80029125 0.23403905]

Bias: [0.06518426]

Weighted Sum: [0.58384757]

Output after Sigmoid Activation: [0.64195225]

**Result: -** Program executed successfully.

**EXPERIMENT – 3**

**WRITE A PROGRAM USING TENSORFLOW AND KERAS TO CREATE A NEURAL NETWORK WITH AN OPTIMIZER**

**Aim: -** To sets up a basic neural network with the SGD optimizer for the MNIST digit classification task.

**Procedure: -**

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

# Create a simple dataset for demonstration

X = [[0, 0], [0, 1], [1, 0], [1, 1]]

y = [0, 1, 1, 0]

# Define a fully connected neural network with SGD optimizer

model = Sequential()

# Input layer with 2 neurons

model.add(Dense(units=2, input\_dim=2, activation='relu', name='input\_layer'))

# Output layer with 1 neuron (binary classification)

model.add(Dense(units=1, activation='sigmoid', name='output\_layer'))

# Compile the model with Stochastic Gradient Descent (SGD) optimizer

sgd\_optimizer = tf.keras.optimizers.SGD(learning\_rate=0.01)

model.compile(optimizer=sgd\_optimizer, loss='binary\_crossentropy', metrics=['accuracy'])

# Display the model summary

model.summary()

**Output: -**

Model: "sequential\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_layer (Dense) (None, 2) 6

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

output\_layer (Dense) (None, 1) 3

=================================================================

Total params: 9

Trainable params: 9

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Result: -** Program executed successfully.

**EXPERIMENT – 4**

**SIMPLE PROGRAM THAT BUILDS A GENERATOR FOR A GENERATIVE ADVERSARIAL NETWORK (GAN)**

**Aim: -** To generates images that resemble handwritten digits using a simple generator architecture.

**Procedure:-**

import tensorflow as tf

from tensorflow.keras import layers, models

import numpy as np

import matplotlib.pyplot as plt

# Define the generator model

def build\_generator(latent\_dim):

model = models.Sequential()

model.add(layers.Dense(128, input\_dim=latent\_dim, activation='relu'))

model.add(layers.Dense(784, activation='sigmoid'))

model.add(layers.Reshape((28, 28, 1)))

return model

# Build the generator

latent\_dim = 100

generator = build\_generator(latent\_dim)

# Display the generator summary

generator.summary()

**Output:-**

Model: "sequential\_2"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense\_2 (Dense) (None, 128) 12928

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_3 (Dense) (None, 784) 101136

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

reshape (Reshape) (None, 28, 28, 1) 0

=================================================================

Total params: 114,064

Trainable params: 114,064

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Result: -** Program executed successfully.

**EXPERIMENT – 5**

**WRITE A PROGRAM THAT BUILDS A DISCRIMINATOR FOR A GENERATIVE ADVERSARIAL NETWORK (GAN)**

**Aim: -** To discriminate between real and generated images, particularly focusing on recognizing handwritten digits.

**Procedure: -**

import tensorflow as tf

from tensorflow.keras import layers, models

# Define the discriminator model

def build\_discriminator(input\_shape=(28, 28, 1)):

model = models.Sequential()

model.add(layers.Flatten(input\_shape=input\_shape))

model.add(layers.Dense(128, activation='relu'))

model.add(layers.Dense(1, activation='sigmoid'))

return model

# Build the discriminator

discriminator = build\_discriminator()

# Display the discriminator summary

discriminator.summary()

**Output:-**

Model: "sequential\_3"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

flatten (Flatten) (None, 784) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_4 (Dense) (None, 128) 100480

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_5 (Dense) (None, 1) 129

=================================================================

Total params: 100,609

Trainable params: 100,609

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Result: -** Program executed successfully.

**EXPERIMENT – 6**

**CREATE A PYTHON PROGRAM USING TENSORFLOW AND KERAS TO DEFINE A NEURAL NETWORK WITH A SPECIFIC ACTIVATION FUNCTION**

**Aim: -** By creating function create\_neural\_network() that takes the input shape as an argument and returns a simple neural network model.

**Procedure: -**

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Dense

import numpy as np

# Create a simple dataset for demonstration

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

y = np.array([[0], [1], [1], [0]])

# Define a simple neural network with one hidden layer and a specific activation function

model = Sequential()

# Input layer (2 input nodes)

model.add(Dense(units=2, input\_dim=2, activation='relu', name='input\_layer'))

# Hidden layer with a specific activation function (e.g., sigmoid)

model.add(Dense(units=1, activation='sigmoid', name='output\_layer'))

# Compile the model

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

# Display the model summary

model.summary()

# Train the model on the dataset

model.fit(X, y, epochs=1000, verbose=0)

# Evaluate the trained model

loss, accuracy = model.evaluate(X, y)

print(f'\nEvaluation - Loss: {loss}, Accuracy: {accuracy}')

# Display the activations of the hidden layer for a sample input

sample\_input = np.array([[0, 1]])

hidden\_layer\_activation **=** Model(inputs**=**model**.**input,

outputs**=**model**.**get\_layer('input\_layer')**.**output)**.**predict(sample\_input)

output\_activation **=** Model(inputs**=**model**.**input,

outputs**=**model**.**get\_layer('output\_layer')**.**output)**.**predict(sample\_input)

print("\nHidden Layer Activation:")

print(hidden\_layer\_activation)

print("\nOutput Layer Activation:")

print(output\_activation)

**Output: -**

Model: "sequential\_17"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_layer (Dense) (None, 2) 6

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

output\_layer (Dense) (None, 1) 3

=================================================================

Total params: 9

Trainable params: 9

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1/1 [==============================] - 0s 425ms/step - loss: 0.5255 - accuracy: 0.7500

Evaluation - Loss: 0.5255492329597473, Accuracy: 0.75

Hidden Layer Activation:

[[0. 1.1805922]]

Output Layer Activation:

[[0.84069824]]

**Result: -** Program executed successfully.

**EXPERIMENT – 7**

**CREATE A SIMPLE PYTHON PROGRAM THAT CREATES AN ENCODER FOR AN AUTOENCODER**

**Aim: -** The aim of this program is to create a simple python program that creates an encoder for an autoencoder.

**Procedure: -**

import numpy as np

import tensorflow as tf

from tensorflow.keras import layers, models

import matplotlib.pyplot as plt

# Load the MNIST dataset

(x\_train, \_), (x\_test, \_) = tf.keras.datasets.mnist.load\_data()

# Normalize pixel values to the range [0, 1]

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

# Flatten the images

x\_train = x\_train.reshape((len(x\_train), np.prod(x\_train.shape[1:])))

x\_test = x\_test.reshape((len(x\_test), np.prod(x\_test.shape[1:])))

# Build the autoencoder model with an encoder and decoder

encoding\_dim = 32 # Number of neurons in the bottleneck layer

# Input layer

input\_img = tf.keras.Input(shape=(784,))

# Encoder

encoded = layers.Dense(encoding\_dim, activation='relu')(input\_img)

# Decoder (not used in this example, but included for completeness)

decoded = layers.Dense(784, activation='sigmoid')(encoded)

# Autoencoder model

autoencoder = models.Model(input\_img, decoded)

# Encoder model (only includes the encoder part)

encoder = models.Model(input\_img, encoded)

# Compile the autoencoder (not used in this example, but included for completeness)

autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

# Display the encoder model summary

encoder.summary()

**Output: -**

Model: "model\_1"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_1 (InputLayer) [(None, 784)] 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_6 (Dense) (None, 32) 25120

=================================================================

Total params: 25,120

Trainable params: 25,120

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Result: -** Program executed successfully.

**EXPERIMENT – 8**

**WRITE A PYTHON PROGRAM THAT CREATES A DECODER FOR AN AUTOENCODER**

**Aim: -** The aim of this program is to create a python program that creates a decoder for an autoencoder.

**Procedure: -**

import numpy as np

import tensorflow as tf

from tensorflow.keras import layers, models

import matplotlib.pyplot as plt

# Load the MNIST dataset

(x\_train, \_), (x\_test, \_) = tf.keras.datasets.mnist.load\_data()

# Normalize pixel values to the range [0, 1]

x\_train = x\_train.astype('float32') / 255.0

x\_test = x\_test.astype('float32') / 255.0

# Flatten the images

x\_train = x\_train.reshape((len(x\_train), np.prod(x\_train.shape[1:])))

x\_test = x\_test.reshape((len(x\_test), np.prod(x\_test.shape[1:])))

# Build the autoencoder model with an encoder and decoder

encoding\_dim = 32 # Number of neurons in the bottleneck layer

# Input layer

input\_img = tf.keras.Input(shape=(784,))

# Encoder

encoded = layers.Dense(encoding\_dim, activation='relu')(input\_img)

# Decoder

decoded = layers.Dense(784, activation='sigmoid')(encoded)

# Autoencoder model

autoencoder = models.Model(input\_img, decoded)

# Decoder model (only includes the decoder part)

decoder\_input = tf.keras.Input(shape=(encoding\_dim,))

decoder\_layer = autoencoder.layers[-1] # Use the last layer of the autoencoder

decoder = models.Model(decoder\_input, decoder\_layer(decoder\_input))

# Compile the autoencoder (not used in this example, but included for completeness)

autoencoder.compile(optimizer='adam', loss='binary\_crossentropy')

# Display the decoder model summary

decoder.summary()

**Output: -**

Model: "model\_3"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_3 (InputLayer) [(None, 32)] 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_9 (Dense) (None, 784) 25872

=================================================================

Total params: 25,872

Trainable params: 25,872

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Result: -** Program executed successfully.

**EXPERIMENT – 9**

**CREATE A PROGRAM OF AN ENCODER USING A BASIC NEURAL NETWORK WITH KERAS IN PYTHON**

**Aim: -** To use the trained encoder to encode the original data, and we display the original and encoded data for the first example.

**Procedure: -**

# Import necessary libraries

from keras.layers import Input, Dense

from keras.models import Model

import numpy as np

# Create a simple dataset for demonstration

# Each data point is a vector of length 10

# You can replace this with your own dataset

data = np.random.random((1000, 10))

# Define the architecture of the encoder

input\_data = Input(shape=(10,))

encoded = Dense(5, activation='relu')(input\_data) # Encoder layer with 5 neurons

# Create the encoder model

encoder\_model = Model(input\_data, encoded)

# Compile the encoder model (not necessary for an encoder, but included for completeness)

encoder\_model.compile(optimizer='adam', loss='mse') # Use mean squared error as a dummy loss

# Display the architecture of the encoder

encoder\_model.summary()

# Encode the data using the trained encoder

encoded\_data = encoder\_model.predict(data)

# Display the original and encoded data for the first example

print("Original Data:")

print(data[0])

print("Encoded Data:")

print(encoded\_data[0])

**Output: -**

Model: "model\_4"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_4 (InputLayer) [(None, 10)] 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_10 (Dense) (None, 5) 55

=================================================================

Total params: 55

Trainable params: 55

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Original Data:

[0.84292158 0.52975721 0.79121258 0.28516277 0.08029 0.43078745

0.43391456 0.47488537 0.5187685 0.48892646]

Encoded Data:

[0.5135934 0.598009 0. 0.9055654 0. ]

**Result: -** Program executed successfully.

**EXPERIMENT – 10**

**PROGRAM TO DEMONSTRATE THE FLOW OF DATA THROUGH A NEURAL NETWORK DURING FORWARD PROPAGATION**

**Aim: -** Create a program demonstrates the flow of data through a neural network during forward propagation.

**Procedure: -**

import numpy as np

def sigmoid(x):

return 1 / (1 + np.exp(-x))

# Define a simple neural network with random weights and biases

def initialize\_parameters(input\_size, hidden\_size, output\_size):

np.random.seed(42)

W1 = np.random.randn(hidden\_size, input\_size) # Weights for the first layer

b1 = np.zeros((hidden\_size, 1)) # Biases for the first layer

W2 = np.random.randn(output\_size, hidden\_size) # Weights for the second layer

b2 = np.zeros((output\_size, 1)) # Biases for the second layer

parameters = {"W1": W1, "b1": b1, "W2": W2, "b2": b2}

return parameters

def forward\_propagation(X, parameters):

# Retrieve parameters

W1, b1, W2, b2 = parameters["W1"], parameters["b1"], parameters["W2"], parameters["b2"]

# Forward pass through the first layer

Z1 = np.dot(W1, X) + b1

A1 = sigmoid(Z1)

# Forward pass through the second layer

Z2 = np.dot(W2, A1) + b2

A2 = sigmoid(Z2)

# Output of the neural network

output = A2

return output

# Example usage

input\_size = 3

hidden\_size = 4

output\_size = 1

# Initialize random parameters

parameters = initialize\_parameters(input\_size, hidden\_size, output\_size)

# Generate random input data

X = np.random.randn(input\_size, 1)

# Perform forward propagation

output = forward\_propagation(X, parameters)

# Display the input data and the output of the neural network

print("Input Data:")

print(X)

print("\nOutput of the Neural Network:")

print(output)

**Output: -**

Input Data:

[[-1.01283112]

[ 0.31424733]

[-0.90802408]]

Output of the Neural Network:

[[0.25942616]]

**Result: -** Program executed successfully.

**EXPERIMENT – 11**

**PROGRAM TO DEFINE A BASIC NEURAL NETWORK LAYER WITH BIAS AND PERFORMS A FORWARD PASS WITH A GIVEN INPUT**

**Aim: -** Create a program defines a basic neural network layer with bias and performs a forward pass with a given input and generates output predictions for a given input.

**Procedure: -**

import numpy as np

def linear\_activation(inputs, weights, bias):

"""

Perform a linear activation (weighted sum + bias) for a neural network layer.

Args:

- inputs: Input data (numpy array)

- weights: Weights for the layer (numpy array)

- bias: Bias for the layer (scalar)

Returns:

- output: Output of the layer (numpy array)

"""

weighted\_sum = np.dot(inputs, weights) + bias

return weighted\_sum

# Define a simple neural network layer with bias

input\_size = 3

output\_size = 1

# Randomly initialize weights and bias

weights = np.random.randn(input\_size)

bias = np.random.randn()

# Create input data

input\_data = np.array([0.5, 0.3, 0.2])

# Perform a forward pass through the layer

output = linear\_activation(input\_data, weights, bias)

# Display the input data, weights, bias, and output

print("Input Data:", input\_data)

print("Weights:", weights)

print("Bias:", bias)

print("Output:", output)

**Output: -**

Input Data: [0.5 0.3 0.2]

Weights: [-1.4123037 1.46564877 -0.2257763 ]

Bias: 0.06752820468792384

Output: -0.2440842754005629

**Result: -** Program executed successfully.

**EXPERIMENT – 12**

**WRITE A SIMPLE PYTHON PROGRAM THAT DEMONSTRATES GRADIENT DESCENT**

**Aim: -** To create a simple program that demonstrates gradient descent.

**Procedure: -**

import numpy as np

import matplotlib.pyplot as plt

# Function to compute the gradient of a quadratic function

def compute\_gradient(x):

return 2 \* x

# Gradient Descent function

def gradient\_descent(initial\_x, learning\_rate, num\_iterations):

"""

Perform gradient descent optimization on a quadratic function.

Args:

- initial\_x: Initial guess for the minimum (scalar)

- learning\_rate: Step size for each iteration (scalar)

- num\_iterations: Number of iterations to perform (integer)

Returns:

- x\_values: List of x values during the optimization (list)

- y\_values: List of y values (quadratic function) during the optimization (list)

"""

x\_values = []

y\_values = []

x = initial\_x

for \_ in range(num\_iterations):

# Compute the gradient

gradient = compute\_gradient(x)

# Update the value of x using gradient descent

x = x - learning\_rate \* gradient

# Calculate the corresponding y value (quadratic function)

y = x\*\*2

# Store the values for visualization

x\_values.append(x)

y\_values.append(y)

return x\_values, y\_values

# Initial parameters

initial\_guess = 4.0

learning\_rate = 0.1

num\_iterations = 20

# Perform gradient descent

x\_values, y\_values = gradient\_descent(initial\_guess, learning\_rate, num\_iterations)

# Display the results

print("Optimal x value:", x\_values[-1])

print("Optimal y value (minimized):", y\_values[-1])

# Plot the optimization process

plt.plot(range(num\_iterations), y\_values, marker='o')

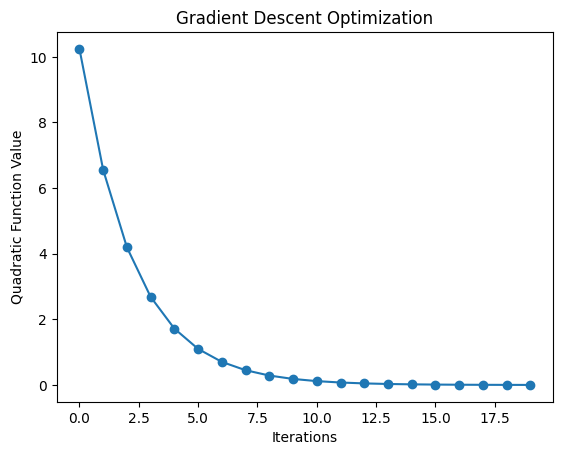
plt.xlabel('Iterations')

plt.ylabel('Quadratic Function Value')

plt.title('Gradient Descent Optimization')

plt.show()

**Output: -**



**Result: -** Program executed successfully.

**EXPERIMENT – 13**

**CREATE A PYTHON PROGRAM USING TENSORFLOW AND KERAS TO DEFINE A NEURAL NETWORK WITH WEIGHTS**

**Aim: -** The aim of this program is tocreate simple Python program using TensorFlow and Keras to define a neural network with weights.

**Procedure: -**

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

import numpy as np

# Create a simple dataset for demonstration

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

y = np.array([[0], [1], [1], [0]])

# Define a simple neural network with one hidden layer

model = Sequential()

# Input layer (2 input nodes)

model.add(Dense(units=2, input\_dim=2, activation='relu', name='input\_layer'))

# Hidden layer with weights to be defined

model.add(Dense(units=1, activation='sigmoid', name='output\_layer'))

# Compile the model

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

# Display the model summary

model.summary()

# Train the model on the dataset

model.fit(X, y, epochs=1000, verbose=0)

# Evaluate the trained model

loss, accuracy = model.evaluate(X, y)

print(f'\nEvaluation - Loss: {loss}, Accuracy: {accuracy}')

# Display the learned weights

print("\nLearned Weights:")

for layer in model.layers:

if 'Dense' in layer.name:

weights, biases = layer.get\_weights()

print(f"{layer.name} Weights:\n{weights}\n")

**Output: -**

Model: "sequential\_7"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_layer (Dense) (None, 2) 6

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

output\_layer (Dense) (None, 1) 3

=================================================================

Total params: 9

Trainable params: 9

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1/1 [==============================] - 0s 434ms/step - loss: 0.5028 - accuracy: 0.7500

Evaluation - Loss: 0.5028179883956909, Accuracy: 0.75

Learned Weights:

output\_layer Weights:

[[ 1.913241 ]

[-0.6327765]]

**Result: -** Program executed successfully.

**EXPERIMENT – 14**

**WRITE A PYTHON PROGRAM USING TENSORFLOW AND KERAS TO DEFINE A NEURAL NETWORK WITH BIASES**

**Aim: -** The aim of this program is to create a Python program using TensorFlow and Keras to define a neural network with biases.

**Procedure: -**

import tensorflow as tf

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

import numpy as np

# Create a simple dataset for demonstration

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

y = np.array([[0], [1], [1], [0]])

# Define a simple neural network with one hidden layer

model = Sequential()

# Input layer (2 input nodes)

model.add(Dense(units=2, input\_dim=2, activation='relu', use\_bias=True, name='input\_layer'))

# Hidden layer with biases to be defined

model.add(Dense(units=1, activation='sigmoid', use\_bias=True, name='output\_layer'))

# Compile the model

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

# Display the model summary

model.summary()

# Train the model on the dataset

model.fit(X, y, epochs=1000, verbose=0)

# Evaluate the trained model

loss, accuracy = model.evaluate(X, y)

print(f'\nEvaluation - Loss: {loss}, Accuracy: {accuracy}')

# Display the learned biases

print("\nLearned Biases:")

for layer in model.layers:

if 'Dense' in layer.name:

weights, biases = layer.get\_weights()

print(f"{layer.name} Biases:\n{biases}\n")

**Output: -**

Model: "sequential\_9"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_layer (Dense) (None, 2) 6

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

output\_layer (Dense) (None, 1) 3

=================================================================

Total params: 9

Trainable params: 9

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1/1 [==============================] - 1s 585ms/step - loss: 0.5683 - accuracy: 0.7500

Evaluation - Loss: 0.5682690739631653, Accuracy: 0.75

Learned Biases:

output\_layer Biases:[-0.41911563]

**Result: -** Program executed successfully.

**EXPERIMENT – 15**

**DEMONSTRATE A PYTHON PROGRAM THAT CREATES AN ARTIFICIAL NEURAL NETWORK (ANN)**

**Aim: -** Creating a Python program that creates an Artificial Neural Network (ANN) using TensorFlow and Keras for a binary classification task with uses a synthetic dataset for illustration purposes. The ANN consists of an input layer, a hidden layer with ReLU activation, and an output layer with a sigmoid activation function.

**Procedure: -**

import numpy as np

import tensorflow as tf

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

from sklearn.preprocessing import StandardScaler

from tensorflow.keras import models, layers

# Generate synthetic data for a binary classification task

np.random.seed(42)

X = np.random.randn(1000, 10) # 1000 samples with 10 features

y = np.random.randint(2, size=(1000, 1)) # Binary labels (0 or 1)

# Split 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)

# Standardize the features using StandardScaler

scaler = StandardScaler()

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

X\_test = scaler.transform(X\_test)

# Build the ANN model

model = models.Sequential()

model.add(layers.Dense(32, activation='relu', input\_shape=(X\_train.shape[1],)))

model.add(layers.Dense(1, activation='sigmoid'))

# Compile the model

model.compile(optimizer='adam', loss='binary\_crossentropy', metrics=['accuracy'])

# Display the model summary

model.summary()

# Train the model

model.fit(X\_train, y\_train, epochs=10, batch\_size=32, validation\_data=(X\_test, y\_test))

# Evaluate the model on the test set

test\_loss, test\_acc = model.evaluate(X\_test, y\_test)

print(f'Test accuracy: {test\_acc}')

**Output: -**

Model: "sequential\_10"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense (Dense) (None, 32) 352

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 1) 33

=================================================================

Total params: 385

Trainable params: 385

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Epoch 1/10

25/25 [==============================] - 3s 38ms/step - loss: 0.7165 - accuracy: 0.5125 - val\_loss: 0.7165 - val\_accuracy: 0.5050

Epoch 2/10

25/25 [==============================] - 0s 15ms/step - loss: 0.7058 - accuracy: 0.5138 - val\_loss: 0.7099 - val\_accuracy: 0.5050

Epoch 3/10

25/25 [==============================] - 0s 15ms/step - loss: 0.6992 - accuracy: 0.5188 - val\_loss: 0.7063 - val\_accuracy: 0.5150

Epoch 4/10

25/25 [==============================] - 0s 17ms/step - loss: 0.6945 - accuracy: 0.5275 - val\_loss: 0.7039 - val\_accuracy: 0.5100

Epoch 5/10

25/25 [==============================] - 0s 15ms/step - loss: 0.6907 - accuracy: 0.5312 - val\_loss: 0.7027 - val\_accuracy: 0.5250

Epoch 6/10

25/25 [==============================] - 0s 16ms/step - loss: 0.6880 - accuracy: 0.5362 - val\_loss: 0.7018 - val\_accuracy: 0.5000

Epoch 7/10

25/25 [==============================] - 1s 23ms/step - loss: 0.6853 - accuracy: 0.5425 - val\_loss: 0.7020 - val\_accuracy: 0.4800

Epoch 8/10

25/25 [==============================] - 0s 20ms/step - loss: 0.6831 - accuracy: 0.5575 - val\_loss: 0.7018 - val\_accuracy: 0.4700

Epoch 9/10

25/25 [==============================] - 1s 21ms/step - loss: 0.6812 - accuracy: 0.5638 - val\_loss: 0.7014 - val\_accuracy: 0.4750

Epoch 10/10

25/25 [==============================] - 1s 27ms/step - loss: 0.6799 - accuracy: 0.5612 - val\_loss: 0.7017 - val\_accuracy: 0.4800

7/7 [==============================] - 0s 10ms/step - loss: 0.7017 - accuracy: 0.4800

Test accuracy: 0.47999998927116394

**Result: -** Program executed successfully.