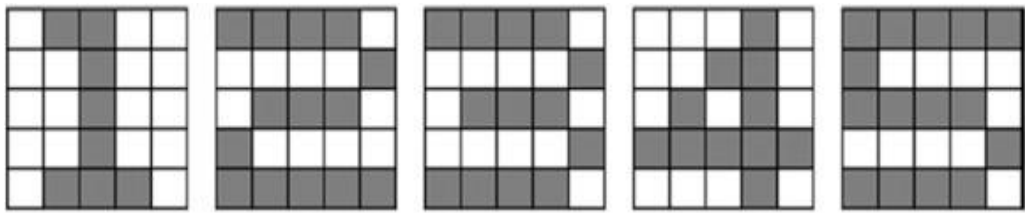
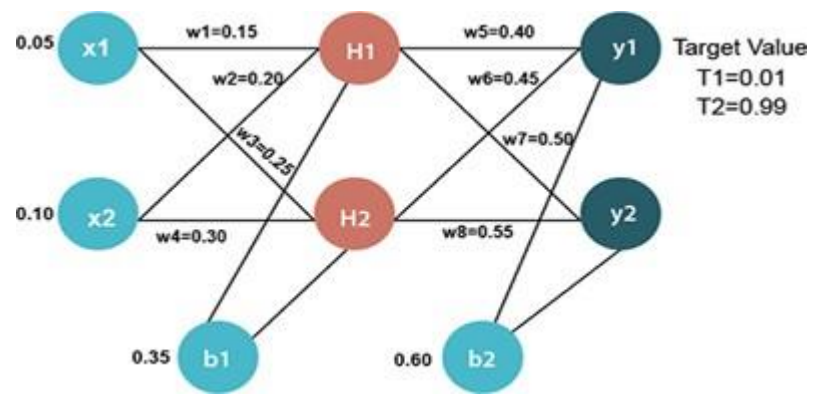




INDEX

Pb. No.	Name of the Problem
01	Write a MATLAB or Python program using perceptron net for AND function with bipolar inputs and targets. The convergence curves and the decision boundary lines are also shown.
02	Generate the XOR function using the McCulloch-Pitts neuron by writing an M-file or .py file. The convergence curves and the decision boundary lines are also shown.
03	Implement the SGD Method using Delta learning rule for following input-target sets. $X_{Input} = [0\ 0\ 1; 0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 1]$, $D_{Target} = [0; 0; 1; 1]$
04	Compare the performance of SGD and the Batch method using the delta learning rule.
05	<p>Write a MATLAB or Python program to recognize the image of digits. The input images are five by-five pixel squares, which display five numbers from 1 to 5, as shown in Figure 1.</p>  <p>Figure 1: Five-by-five pixel squares that display five numbers from 1 to 5</p>
06	Write a MATLAB or Python program to classify face/fruit/bird using Convolution Neural Network (CNN).
07	<p>Consider an artificial neural network (ANN) with three layers given below. Write a MATLAB or Python program to learn this network using Back Propagation Network.</p> 
08	Write a MATLAB or Python program to recognize the numbers 1 to 4 from speech signal using artificial neural network (ANN).
09	Write a MATLAB or Python program to Purchase Classification Prediction using SVM.
10	Write a MATLAB or Python program to reduce dimensions of a dataset into a new coordinate system using PCA algorithm.

Source Code:

```
import numpy as np
import matplotlib.pyplot as plt
# Bipolar activation function
def bipolar_activation(x):
    return 1 if x >= 0 else -1
# Perceptron training function
def perceptron_train(inputs, targets, learning_rate=0.1, max_epochs=100):
    num_inputs = inputs.shape[1]
    num_samples = inputs.shape[0]
    # Initialize weights and bias
    weights = np.random.randn(num_inputs)
    bias = np.random.randn()
    convergence_curve = []

    for epoch in range(max_epochs):
        misclassified = 0

        for i in range(num_samples):
            net_input = np.dot(inputs[i], weights) + bias
            predicted = bipolar_activation(net_input)

            if predicted != targets[i]:
                misclassified += 1
                update = learning_rate * (targets[i] - predicted)
                weights += update * inputs[i]
                bias += update

        accuracy = (num_samples - misclassified) / num_samples
        convergence_curve.append(accuracy)

        if misclassified == 0:
            print("Converged in { } epochs.".format(epoch + 1))
            break

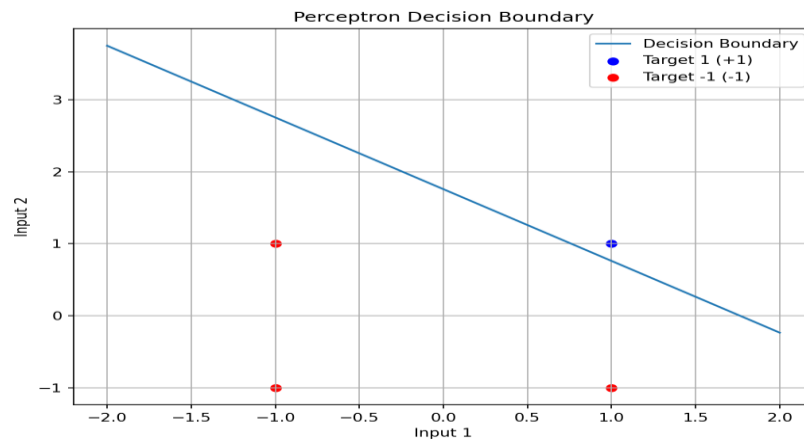
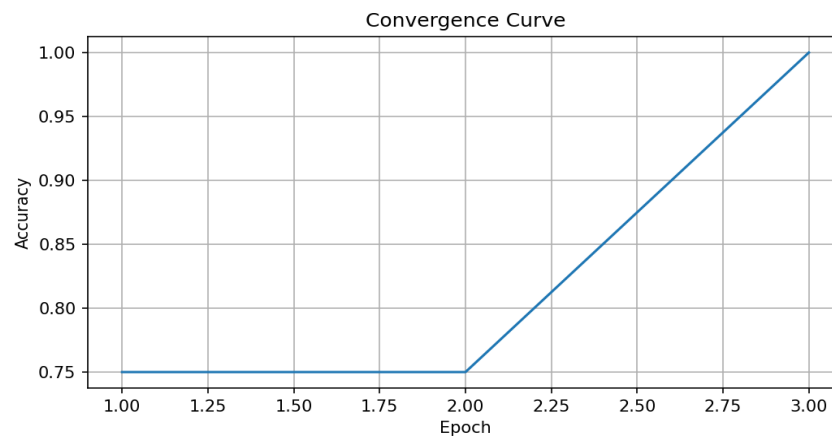
    return weights, bias, convergence_curve
# Main function
if __name__ == "__main__":
    # Input and target data (bipolar representation)
    inputs = np.array([[-1, -1], [-1, 1], [1, -1], [1, 1]])
    targets = np.array([-1, -1, -1, 1])
    # Training the perceptron
    weights, bias, convergence_curve = perceptron_train(inputs, targets)
    # Decision boundary line
    x = np.linspace(-2, 2, 100)

    y = (-weights[0] * x - bias) / weights[1]
    # print(y)
    # Plot convergence curve
    plt.figure(figsize=(8, 4))
    plt.plot(range(1, len(convergence_curve) + 1), convergence_curve)
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Convergence Curve')
    plt.grid()
    plt.show()
```

```
# Plot the decision boundary line and data points
plt.figure(figsize=(8, 6))
plt.plot(x, y, label='Decision Boundary')
plt.xlabel('Input 1')
plt.ylabel('Input 2')
plt.title('Perceptron Decision Boundary')
plt.legend()
plt.grid()
plt.show()
```

Output:

Converged in 3 epochs.



Source Code:

```
import numpy as np
import matplotlib.pyplot as plt
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)
inputs = np.array([[0, 0],
                   [0, 1],
                   [1, 0],
                   [1, 1]])
targets = np.array([0, 1, 1, 0])
# Neural network parameters
input_layer_size = 2
hidden_layer_size = 2
output_layer_size = 1
learning_rate = 0.1
max_epochs = 10000
# Initialize weights and biases with random values
np.random.seed(42)
weights_input_hidden = np.random.randn(input_layer_size, hidden_layer_size)
bias_hidden = np.random.randn(hidden_layer_size)
weights_hidden_output = np.random.randn(hidden_layer_size, output_layer_size)
bias_output = np.random.randn(output_layer_size)
convergence_curve = []
# Training the neural network
for epoch in range(max_epochs):
    misclassified = 0
    for i in range(len(inputs)):
        # Update weights and biases
        weights_hidden_output += hidden_layer_output[:, np.newaxis] * output_delta * learning_rate
        bias_output += output_delta * learning_rate

        weights_input_hidden += inputs[i][:, np.newaxis] * hidden_delta * learning_rate
        bias_hidden += hidden_delta * learning_rate
    err = 1 - (len(inputs) - misclassified) / len(inputs)
    convergence_curve.append(err)
    if misclassified == 0:
        print("Converged in {} epochs.".format(epoch + 1))
        break
# Plot convergence curve
plt.figure(figsize=(8, 4))
plt.plot(range(1, len(convergence_curve) + 1), convergence_curve)
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.title('Convergence Curve')
plt.grid()
plt.show()
```

```
# Create a grid of points to plot decision boundaries
```

```
x1 = np.linspace(-0.5, 1.5, 200)
```

```
x2 = np.linspace(-0.5, 1.5, 200)
```

```
X1, X2 = np.meshgrid(x1, x2)
```

```
Z = predict(X1, X2)
```

```
# Plot decision boundaries
```

```
plt.figure(figsize=(8, 6))
```

```
plt.xlabel('Input 1')
```

```
plt.ylabel('Input 2')
```

```
plt.title('XOR Function Decision Boundary')
```

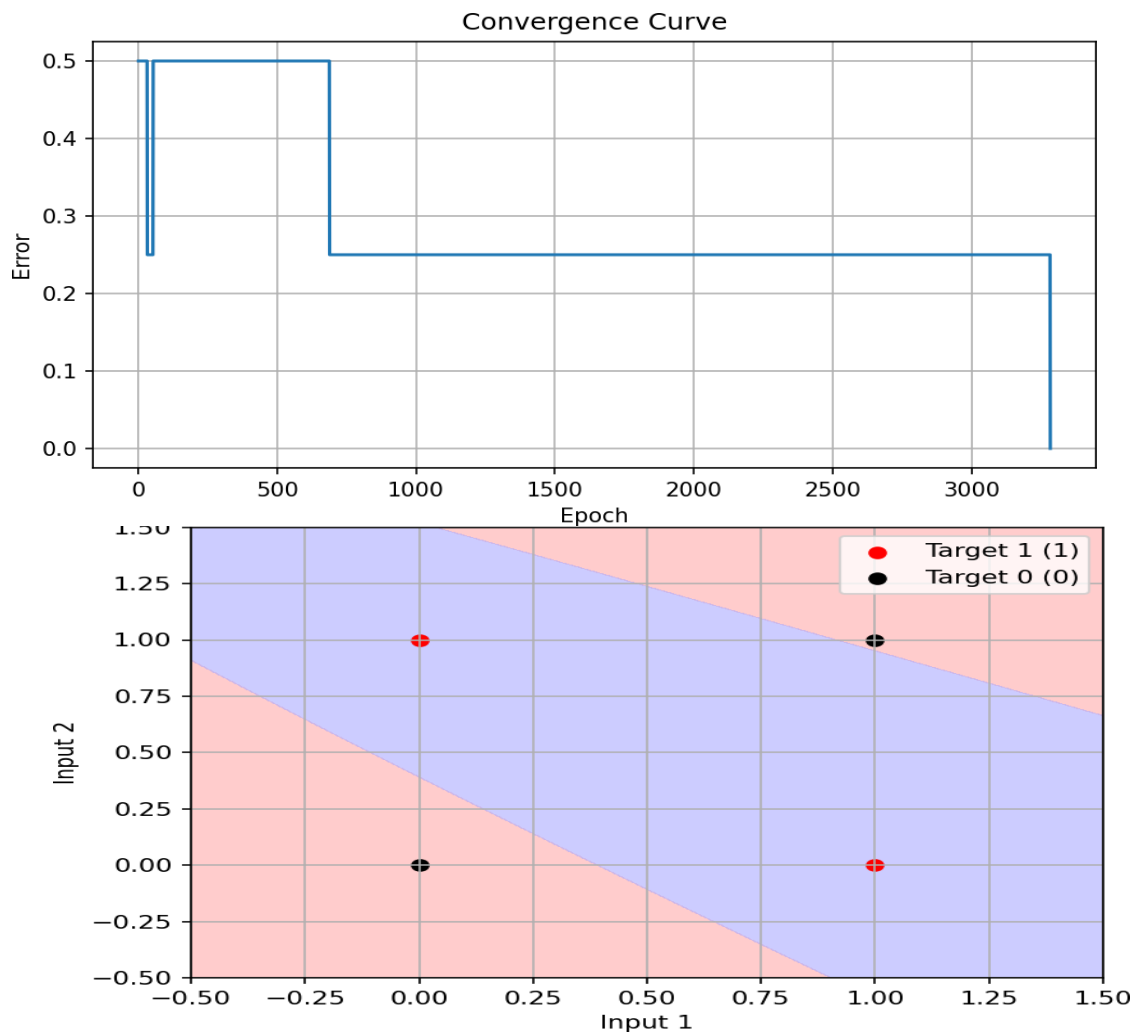
```
plt.legend()
```

```
plt.grid()
```

```
plt.show()
```

Output:

Generated XOR Result is: [0 1 1 0]



Source Code:

```
import numpy as np
import matplotlib.pyplot as plt
# Input and target values
Xinput = np.array([[0, 0, 1], # Bias included in the third column
                  [0, 1, 1],
                  [1, 0, 1],
                  [1, 1, 1]])
Dtarget = np.array([0, 0, 1, 1]) # Target output
# Initialize weights randomly
weights = np.random.randn(3)
learning_rate = 0.1
epochs = 100
# Activation function (Step function)
def step_function(x):
    return np.where(x >= 0, 1, 0)
# Training using SGD and Delta learning rule
convergence_curve = []
converged = False
for epoch in range(epochs):
    total_error = 0

    # Shuffle the data for each epoch (stochastic gradient descent)
    indices = np.random.permutation(len(Xinput))
    Xinput_shuffled = Xinput[indices]
    Dtarget_shuffled = Dtarget[indices]
    for i in range(len(Xinput)):
        x = Xinput_shuffled[i]
        d = Dtarget_shuffled[i]
        # Forward pass (calculate output)
        y = step_function(np.dot(x, weights))

        # Calculate error
        error = d - y
        total_error += abs(error)

        # Delta rule: weight update
        weights += learning_rate * error * x

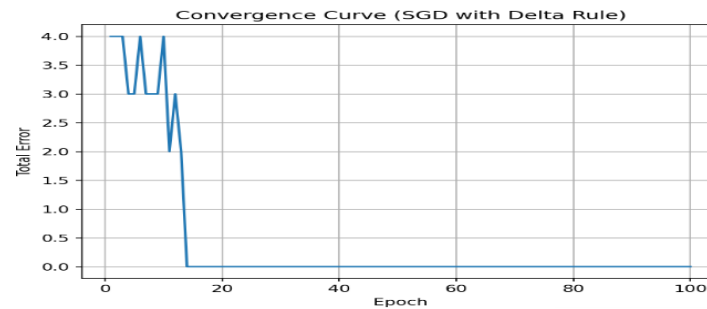
    convergence_curve.append(total_error)
# Stop early if there is no error
if total_error == 0 and converged == False:
    print(f"Converged in {epoch + 1} epochs.")
    converged = True
```

```
# Print final weights
print("Final weights:", weights)
# Plot convergence curve
plt.plot(range(1, len(convergence_curve) + 1), convergence_curve)
plt.xlabel('Epoch')
plt.ylabel('Total Error')
plt.title('Convergence Curve (SGD with Delta Rule)')
plt.grid()
plt.show()
```

Output:

Converged in 14 epochs.

Final weights: [0.1209001 -0.04639043 -0.01045075]



Source Code:

```
import numpy as np
import time
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_derivative(x):
    return x * (1 - x)
# XOR function dataset with binary inputs and outputs
X_input = np.array([[0, 0, 1],[0, 1, 1],[1, 0, 1],[1, 1, 1]])
D_target = np.array([[0],[0],[1],[1]])
# Neural network parameters
input_layer_size = 3
output_layer_size = 1
learning_rate = 0.1
max_epochs = 10000
# Initialize weights with random values
np.random.seed(42)
weights_sgd = np.random.randn(input_layer_size, output_layer_size)
weights_batch = np.random.randn(input_layer_size, output_layer_size)
# Training the neural network with SGD
start_time_sgd = time.time()
for epoch in range(max_epochs):
    error_sum = 0
    # Check for convergence
    if error_sum < 0.01:
        break
end_time_sgd = time.time()

# Training the neural network with the batch method
start_time_batch = time.time()
for epoch in range(max_epochs):
    # Forward pass
    net_input = np.dot(X_input, weights_batch)
    predicted_output = sigmoid(net_input)

    # Calculate error
    error = D_target - predicted_output
    error_sum = np.sum(np.abs(error))
    # Update weights using the delta learning rule
    weight_update = learning_rate * np.dot(X_input.T, error * sigmoid_derivative(predicted_output))
    weights_batch += weight_update

    # Check for convergence
    if error_sum < 0.01:
        break
end_time_batch = time.time()
```

```
print("SGD Results:")
print("Time taken: {:.6f} seconds".format(end_time_sgd - start_time_sgd))
print("Trained weights:")
print("\nBatch Method Results:")
print("Time taken: {:.6f} seconds".format(end_time_batch - start_time_batch))
print("Trained weights:")
print(weights_batch)
print("Predicted binary outputs:")
print(test_model(weights_batch))
```

Output:

SGD Results:

Time taken: 0.912471 seconds

Trained weights:

[[7.25950187]

[-0.22431325]

[-3.41036643]]

Predicted binary outputs:

[[0.]

[0.]

[1.]

[1.]]

Batch Method Results:

Time taken: 0.418971 seconds

Trained weights:

[[7.26775966]

[-0.22304058]

[-3.41538639]]

Predicted binary outputs:

[[0.]

[0.]

[1.]

[1.]]

Source Code:

```
import numpy as np
def softmax(x):
    ex = np.exp(x)
    return ex / np.sum(ex)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def multi_class(W1, W2, X, D):
    alpha = 0.9
    N = 5
    for k in range(N):
        x = X[:, :, k].reshape(25, 1)
        d = D[k, :].reshape(-1, 1)
        v1 = np.dot(W1, x)
        y1 = sigmoid(v1)
        v = np.dot(W2, y1)
        y = softmax(v)
        e = d - y
        delta = e
        e1 = np.dot(W2.T, delta)
        delta1 = y1 * (1 - y1) * e1
        dW1 = alpha * np.dot(delta1, x.T)
        W1 = W1 + dW1
        dW2 = alpha * np.dot(delta, y1.T)
        W2 = W2 + dW2

    return W1, W2
def main():
    np.random.seed(3)
    X = np.zeros((5, 5, 5))
    X[:, :, 0] = np.array([[0, 1, 1, 0, 0],
                           [0, 0, 1, 0, 0],
                           [0, 0, 1, 0, 0],
                           [0, 0, 1, 0, 0],
                           [0, 1, 1, 1, 0]])
    X[:, :, 1] = np.array([[1, 1, 1, 1, 0],
                           [0, 0, 0, 0, 1],
                           [0, 1, 1, 1, 0],
                           [1, 0, 0, 0, 0],
                           [1, 1, 1, 1, 1]])
    X[:, :, 2] = np.array([[1, 1, 1, 1, 0],
                           [0, 0, 0, 0, 1],
                           [0, 1, 1, 1, 0],
                           [0, 0, 0, 0, 1],
                           [1, 1, 1, 1, 0]])
    X[:, :, 3] = np.array([[0, 0, 0, 1, 0],
```

```

        [0, 0, 1, 1, 0],
        [0, 1, 0, 1, 0],
        [1, 1, 1, 1, 1],
        [0, 0, 0, 1, 0]])
X[:, :, 4] = np.array([[1, 1, 1, 1, 1],
        [1, 0, 0, 0, 0],
        [1, 1, 1, 1, 0],
        [0, 0, 0, 0, 1],
        [1, 1, 1, 1, 0]])

D = np.eye(5)

W1 = 2 * np.random.rand(50, 25) - 1
W2 = 2 * np.random.rand(5, 50) - 1

for epoch in range(10000):
    W1, W2 = multi_class(W1, W2, X, D)

N = 5
for k in range(N):
    x = X[:, :, k].reshape(25, 1)
    v1 = np.dot(W1, x)
    y1 = sigmoid(v1)
    v = np.dot(W2, y1)
    y = softmax(v)
    print(f"\n\n Output for X[:, :, {k}]:\n\n")
    print(f"{y} \n\n This matrix from see that {k+1} position accuracy is higher that is : {max(y)} So this number
is correctly identified")
if __name__ == "__main__":
    main()

```

Output:

Output for X[:, :, 0]:

[[9.99990560e-01]

[3.73975045e-06]

[7.29323123e-07]

[4.95516529e-06]

[1.56459758e-08]]

This matrix from see that 1 position accuracy is higher that is : [0.99999056] So this number is correctly identified

Output for X[:, :, 1]:

[[3.81399150e-06]

[9.99984069e-01]

[1.07138749e-05]

[7.38201374e-07]

[6.65377695e-07]]

This matrix from see that 2 position accuracy is higher that is : [0.99998407] So this number is correctly identified

Output for X[:,2]:

[[2.10669179e-06]

[9.17015598e-06]

[9.99972467e-01]

[2.22084036e-06]

[1.40352894e-05]]

This matrix from see that 3 position accuracy is higher that is : [0.99997247] So this number is correctly identified

Output for X[:,3]:

[[4.72578106e-06]

[8.98916172e-07]

[9.07090140e-07]

[9.99990801e-01]

[2.66714208e-06]]

This matrix from see that 4 position accuracy is higher that is : [0.9999908] So this number is correctly identified

Output for X[:,4]:

[[6.12205780e-07]

[2.29663674e-06]

[1.16748707e-05]

[1.01696314e-06]

[9.99984399e-01]]

This matrix from see that 5 position accuracy is higher that is : [0.9999844] So this number is correctly identified

Source Code:

```
#Training portion of the code
import os
import cv2
import numpy as np
from sklearn.model_selection import train_test_split
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense
def load_data(folder):
    images = []
    labels = []
    for filename in os.listdir(folder):
        label = folder.split('/')[-1]
        img = cv2.imread(os.path.join(folder, filename))
        img = cv2.resize(img, (150, 150)) # Resize the image to a consistent size
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB) # Convert to RGB format
        images.append(img)
        labels.append(label)
    return images, labels
banana_folder = "E:/BOOK/ICE-4-2/ICE-4206_Neural Networks Sessional/Neural Network
Sessional/lab_11/dataset/banana"
cucumber_folder = "E:/BOOK/ICE-4-2/ICE-4206_Neural Networks Sessional/Neural Network
Sessional/lab_11/dataset/cucumber"
# Encode labels to numerical values
label_dict = {'banana': 0, 'cucumber': 1}
encoded_labels = np.array([label_dict[label] for label in labels])
print(encoded_labels)
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(images, encoded_labels, test_size=0.15, random_state=42)
# Normalize the pixel values between 0 and 1
X_train = X_train.astype('float32') / 255
X_test = X_test.astype('float32') / 255
import matplotlib.pyplot as plt
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(150, 150, 3)))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(128, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.add(Conv2D(512, (3, 3), activation='relu'))
model.add(MaxPooling2D((2, 2)))
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
history = model.fit(X_train, y_train, epochs=100, batch_size=32)
# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
# Plotting loss
plt.plot(history.history['loss'], label='Training Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

```

# Plotting accuracy
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
print('Test accuracy:', accuracy*100)
#Testing portion of the code
#.....
from tensorflow.keras.preprocessing import image
import numpy as np
# Path to the test image
test_image_path = 'E:/BOOK/ICE-4-2/ICE-4206_Neural Networks Sessional/Neural Network
Sessional/lab_11/pic1.jpg' # Replace with the actual path of your test image
# Load and preprocess the test image
test_image = image.load_img(test_image_path, target_size=(150, 150))
test_image = image.img_to_array(test_image)
test_image = np.expand_dims(test_image, axis=0)
test_image = test_image / 255.0 # Normalize the image
# Predict the class of the test image
prediction = model.predict(test_image)
print('prediction',prediction)
if prediction < 0.5:
    print('This is Banana')
elif prediction >= 0.5:
    print('This is Cucumber')

```

Output:

Prediction accuracy is: [[0.9634724]]

This is Cucumber

Source Code:

```
import torch
import torch.nn as nn
import torch.optim as optim
class SimpleANN(nn.Module):
    def __init__(self):
        super(SimpleANN, self).__init__()
        # Input to Hidden layer (2 inputs to 2 hidden nodes)
        self.hidden = nn.Linear(2, 2) # 2 input neurons, 2 hidden neurons
        # Hidden to Output layer (2 hidden nodes to 2 output nodes)
        self.output = nn.Linear(2, 2) # 2 hidden neurons, 2 output neurons
        # Sigmoid activation function
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # Forward pass through the network
        h = self.sigmoid(self.hidden(x)) # Hidden layer activation
        y = self.sigmoid(self.output(h)) # Output layer activation
        return y
# Create the
network model =
SimpleANN()
# Set the weights and biases based on the
diagramwith torch.no_grad():
    model.hidden.weight = torch.nn.Parameter(torch.tensor([[0.15, 0.20], [0.25, 0.30]])) # w1, w2, w3, w4
    model.hidden.bias = torch.nn.Parameter(torch.tensor([0.35, 0.35])) # Bias b1 for both hidden neurons
    model.output.weight = torch.nn.Parameter(torch.tensor([[0.40, 0.45], [0.50, 0.55]])) # w5, w6, w7, w8
    model.output.bias = torch.nn.Parameter(torch.tensor([0.60, 0.60])) # Bias b2 for both output neurons
# Define input (x1, x2) and target values (T1, T2)
inputs = torch.tensor([[0.05, 0.10]]) # Single input
pair
targets = torch.tensor([[0.01, 0.99]]) # Target values for y1 and
y2# Define the loss function (Mean Squared Error Loss)
criterion = nn.MSELoss()
# Define the optimizer (Stochastic Gradient
Descent)optimizer =
optim.SGD(model.parameters(), lr=0.5) #
Number of epochs (iterations)
epochs = 15000
# Training
loop
for epoch in
    range(epochs):
        output =
        model(inputs)
        # Compute the loss (Mean Squared
        Error)loss = criterion(output, targets)
        # Print the loss every 1000
        epochsif epoch % 1000 == 0:
            print(f"Epoch {epoch}, Loss: {loss.item()}")
        # Print final weights and biases after training
        print("\nFinal weights and biases:")
        for name, param in
            model.named_parameters():
                print(f"{name}: {param.data}")
```



```
# Final output after
training final_output =
model(inputs)
print(f'\nFinal output (y1, y2): {final_output[0][0]:.2f}, {final_output[0][1]:.2f}')
```

Output:

Epoch 0, Loss: 0.2983711063861847

Epoch 1000, Loss: 0.0002707050589378923

Epoch 2000, Loss: 9.042368037626147e-05

Epoch 3000, Loss: 4.388535307953134e-05

Epoch 4000, Loss: 2.489008147676941e-05

Epoch 5000, Loss: 1.5388504834845662e-05

Epoch 6000, Loss: 1.0049888260255102e-05

Epoch 7000, Loss: 6.816366294515319e-06

Epoch 8000, Loss: 4.752399945573416e-06

Epoch 9000, Loss: 3.3828823688963894e-06

Epoch 10000, Loss: 2.4476007638440933e-06

Epoch 11000, Loss: 1.7939109966391698e-06

Epoch 12000, Loss: 1.3286518196764519e-06

Epoch 13000, Loss: 9.925176982505945e-07

Epoch 14000, Loss: 7.467624527635053e-07

Final weights and biases:

hidden.weight: tensor([[0.1833, 0.2667], [0.2826, 0.3652]])

hidden.bias: tensor([1.0170, 1.0025])

output.weight: tensor([[-1.4728, -1.4261], [1.5441, 1.5950]])

output.bias: tensor([-2.3714, 2.1961])

Final output (y1, y2): 0.01, 0.99

Source Code:

```
import numpy as np
import librosa
import soundfile as sf
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
# Load audio data and extract features
def extract_features(audio_path):
    y, sr = librosa.load(audio_path, sr=None)
    mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=13)
    mfccs_mean = np.mean(mfccs, axis=1)
    return mfccs_mean
# Load data
def load_data(file_paths, labels):
    features = []
    for file_path in file_paths:
        features.append(extract_features(file_path))
    return np.array(features), np.array(labels)

# Define file paths and labels (paths should be updated to actual audio files)
file_paths = [
    'data/1.wav',
    'data/2.wav',
    'data/3.wav',
    'data/4.wav',
    # Add more paths to your dataset
]
labels = [
    1, 2, 3, 4,
    # Corresponding labels
]
# Preprocess data
X, y = load_data(file_paths, labels)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Train a neural network
clf = MLPClassifier(hidden_layer_sizes=(10,), max_iter=1000, random_state=42)
clf.fit(X_train, y_train)

# Evaluate the model
accuracy = clf.score(X_test, y_test)
print(f'Accuracy: {accuracy:.2f}')
# Predict on new data
def predict_number(audio_path):
    features = extract_features(audio_path).reshape(1, -1)
    prediction = clf.predict(features)
    return prediction[0]
# Example prediction
new_audio_path = 'data/test.wav' # Replace with your test audio file
predicted_number = predict_number(new_audio_path)
print(f'Predicted number: {predicted_number}')
```

Output:

- 1.wav: An audio file with the spoken number "one."
- 2.wav: An audio file with the spoken number "two."
- 3.wav: An audio file with the spoken number "three."
- 4.wav: An audio file with the spoken number "four."
- test.wav: A test audio file where you say "two."

Source Code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.svm import SVR
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.datasets import fetch_openml

#Load the Boston housing dataset
boston = fetch_openml(name='boston', version=1, as_frame=True)
X = boston.data
y = boston.target
# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Create an SVM regressor
svr = SVR(kernel='linear') # You can also try 'rbf' or 'poly'
# Fit the model on the training data
svr.fit(X_train, y_train)
# Make predictions on the test set
y_pred = svr.predict(X_test)
# Evaluate the model
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f'Mean Squared Error: {mse:.2f}')
print(f'R^2 Score: {r2:.2f}')
print("\nPredicted values for the test set:")
for actual, predicted in zip(y_test, y_pred):
    print(f"Actual: {actual:.2f}, Predicted: {predicted:.2f}")
# Plotting the predicted vs actual values
plt.scatter(y_test, y_pred)
plt.xlabel('Actual Prices')
plt.ylabel('Predicted Prices')
plt.title('Actual vs Predicted House Prices')
plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], color='red', linestyle='--')
plt.show()
# Note: Ensure that the custom data has the same feature structure as the training data
custom_test_data = np.array([[0.00632, 18.0, 2.31, 0.0, 0.538, 6.575, 65.2, 4.09, 2.0, 240.0, 17.8, 396.9, 9.14],
                             [0.02731, 0.0, 7.07, 0.0, 0.469, 6.421, 78.9, 4.9671, 2.0, 240.0, 19.58, 396.9, 4.03]])
# Predicting house prices for custom test data
custom_predictions = svr.predict(custom_test_data)
print("\nPredicted values for custom test data:")
for i, prediction in enumerate(custom_predictions):
    print(f"Custom Test Data {i + 1}: Predicted Price: {prediction:.2f}")
```

Output:

Mean Squared Error: 29.44

R² Score: 0.60



Predicted values for custom test data:

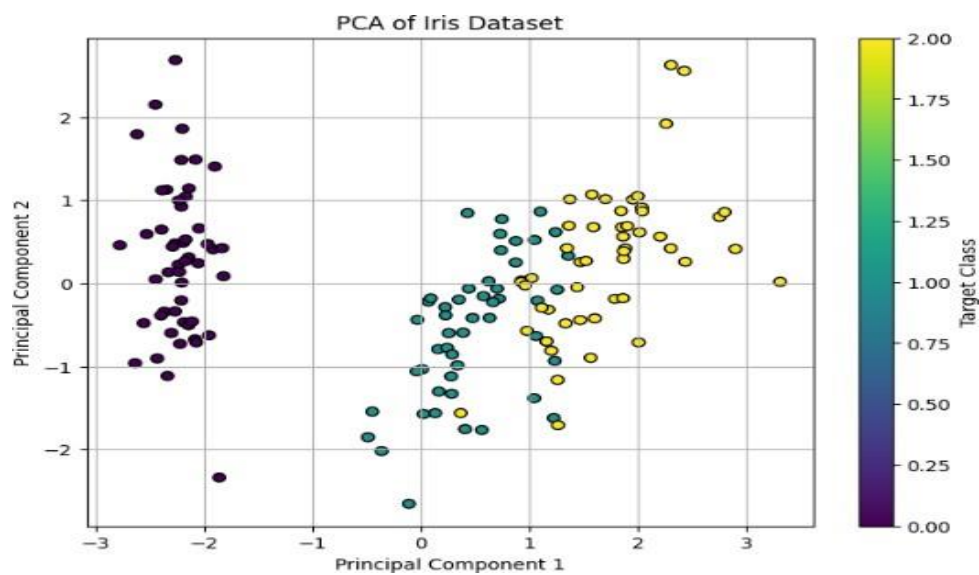
Custom Test Data 1: Predicted Price: 26.34

Custom Test Data 2: Predicted Price: 24.27

Source Code:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler# Load the Iris dataset
iris = load_iris()
X = iris.data
y = iris.target
# Standardize the data
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)# Apply PCA
pca = PCA(n_components=2) # Reduce to 2 dimensions
X_pca = pca.fit_transform(X_scaled)
# Create a DataFrame for the PCA results
pca_df = pd.DataFrame(data=X_pca, columns=['Principal Component 1', 'Principal Component 2'])
pca_df['Target'] = y
# Plot the PCA results
plt.figure(figsize=(8, 6))
scatter = plt.scatter(pca_df['Principal Component 1'], pca_df['Principal Component 2'], c=pca_df['Target'],
cmap='viridis', edgecolor='k')
plt.title('PCA of Iris Dataset')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.colorbar(scatter, label='Target Class')plt.grid()
plt.show()
# Explained variance
explained_variance = pca.explained_variance_ratio_
print(f'Explained variance by each component: {explained_variance}')print(f'Total explained variance: {sum(explained_variance)}')
```

Output:



Explained variance by each component: [0.72962445 0.22850762]

Total explained variance: 0.9581320720000165