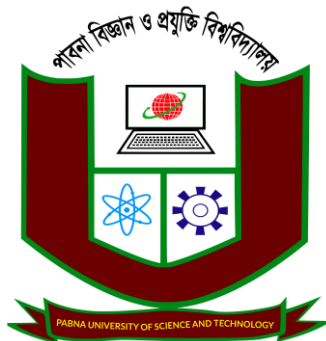


PABNA UNIVERSITY OF SCIENCE & TECHNOLOGY



DEPARTMENT OF INFORMATION AND COMMUNICATION ENGINEERING FACULTY OF ENGINEERING AND TECHNOLOGY

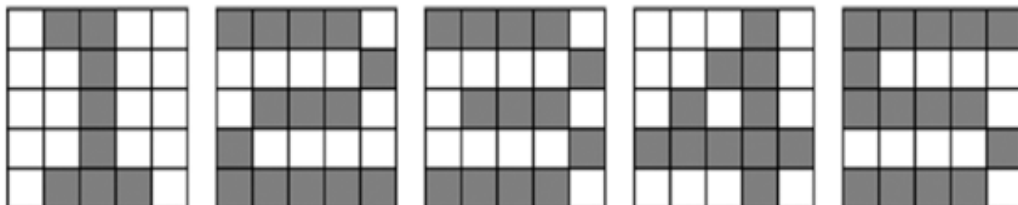
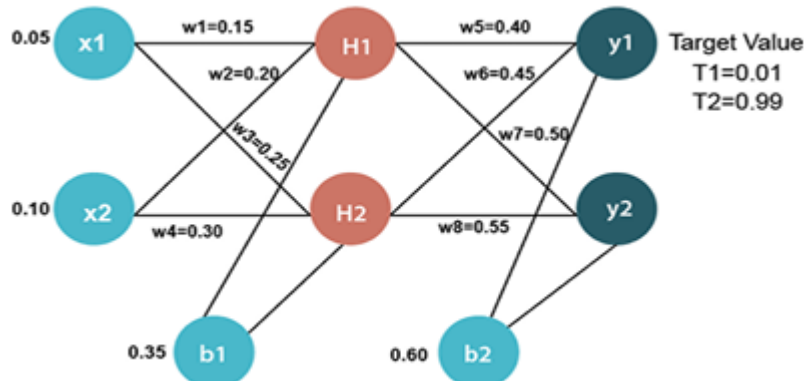
Lab Report

Course Title: Neural Networks Sessional

Course Code: ICE-4206

| | |
|---|--|
| <u>Submitted By:</u> Name: Shahanur Rahman Roll: 190617 Session: 2018-2019 4th Year 2nd Semester, Department of ICE, Pabna University of Science and Technology. Submission Date: 15-09-2024 | <u>Submitted To:</u> Dr. Md. Imran Hossain Associate Professor Department of ICE, Pabna University of Science and Technology. <u>Teacher's Signature:</u> |
|---|--|

INDEX

| Pb. No. | Name of the Problem | Page No. |
|---------|--|----------|
| 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. | 1-4 |
| 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. | 4-8 |
| 03 | Implement the SGD Method using Delta learning rule for following input-target sets. $X_{Input} = [0\ 0\ 1; 0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 1]$, $D_{Target} = [0; 0; 1; 1]$ | 8-11 |
| 04 | Compare the performance of SGD and the Batch method using the delta learning rule. | 11-14 |
| 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> | 14-18 |
| 06 | Write a MATLAB or Python program to classify face/fruit/bird using Convolution Neural Network (CNN). | 19-21 |
| 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>  | 22-25 |
| 08 | Write a MATLAB or Python program to recognize the numbers 1 to 4 from speech signal using artificial neural network (ANN). | 25-27 |
| 09 | Write a MATLAB or Python program to Purchase Classification Prediction using SVM. | 28-30 |
| 10 | Write a MATLAB or Python program to reduce dimensions of a dataset into a new coordinate system using PCA algorithm. | 30-32 |

Problem No: 01

Problem Name: 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.

Theory:

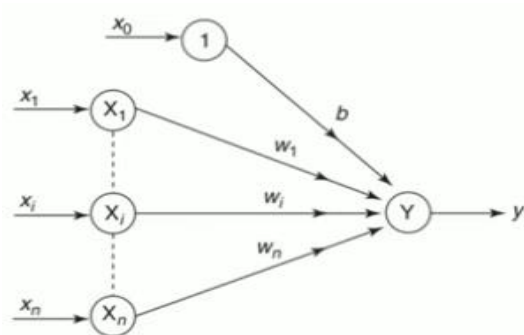
Perceptron for AND Function: A **perceptron** is the simplest form of a neural network model. It consists of a single layer of output nodes connected to inputs. Each output node represents a single binary output.

Key Concepts:

Bipolar Inputs and Targets: Bipolar values mean that inputs and outputs are either -1 or +1 instead of the standard 0 and 1 used in binary representation. For the AND function with two inputs, the input combinations and their corresponding outputs are:

| Input x_1 | Input x_2 | Target t |
|-------------|-------------|------------|
| +1 | +1 | +1 |
| +1 | -1 | -1 |
| -1 | +1 | -1 |
| -1 | -1 | -1 |

Perceptron Learning Rule:



Calculated input y_{in} is:

$$y_{in} = b + \sum_{i=1}^n x_i w_i$$

Activation function:

$$y = f(y_{in}) = \begin{cases} 1 & \text{if } y_{in} > \theta \\ 0 & \text{if } -\theta \leq y_{in} \leq \theta \\ -1 & \text{if } y_{in} < -\theta \end{cases}$$

The perceptron updates its weights w to minimize the error between the predicted output and the actual output. The weight update rule is given by:

$$w(t + 1) = w(t) + \eta \cdot (t - y) \cdot x$$

where:

$\omega(t)$ is the weight vector at time t

η is the learning rate.

t is the target output.

y is the predicted output.

x is the input vector.

Convergence Curve: The convergence curve in the context of machine learning, particularly in training a perceptron, represents how the error (or loss) of the model changes over time (i.e., over the training epochs). It visually shows the process of learning and how quickly the model is converging to the correct solution.

Decision Boundary: A decision boundary is a concept in machine learning that represents the region of a problem space where the output label of a classifier changes. It is a line, curve, or surface (in higher dimensions) that separates different classes in a classification problem. The decision boundary for the perceptron is a line in the input space that separates classes where the output is +1 or -1.

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

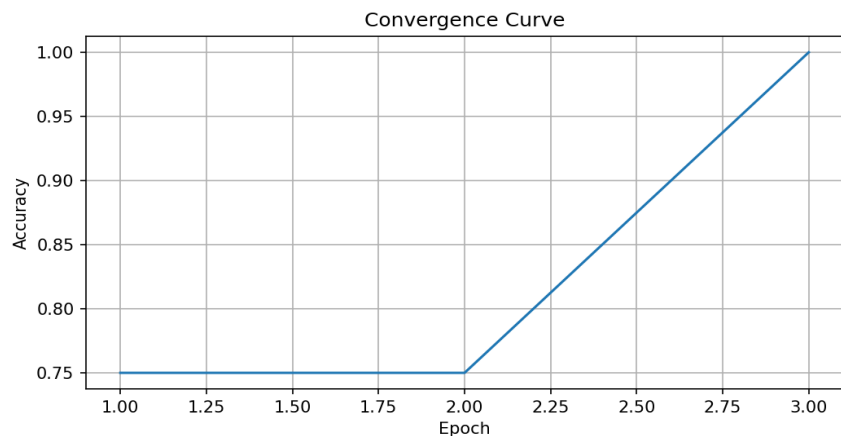
    return weights, bias, convergence_curve
# Main function
if __name__ == "__main__":
    # Input and target data (bipolar representation)
    inputs = np.array([[-1, -1], [-1, 1], [1, -1], [1, 1]])
    targets = np.array([-1, -1, -1, 1])
    # Training the perceptron
    weights, bias, convergence_curve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

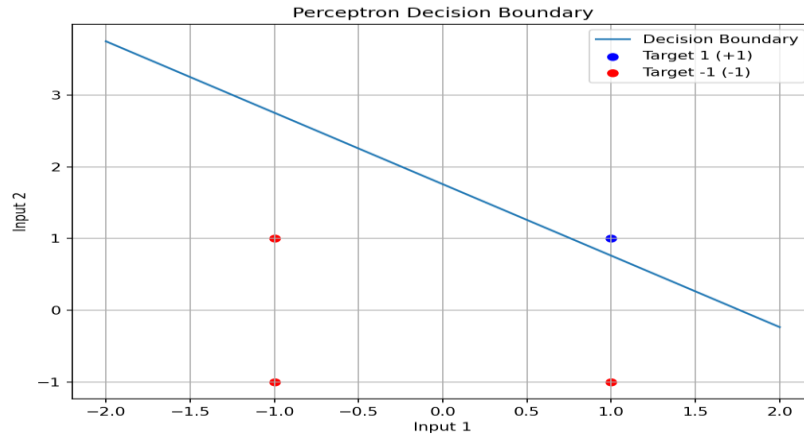
    y = (-weights[0] * x - bias) / weights[1]
    # print(y)
    # Plot convergence curve
    plt.figure(figsize=(8, 4))
    plt.plot(range(1, len(convergence_curve) + 1), convergence_curve)
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Convergence Curve')
    plt.grid()
    plt.show()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





Problem No: 02

Problem Name: 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.

Theory:

McCulloch-Pitts Neuron Model: The McCulloch-Pitts neuron, proposed in 1943 by Warren McCulloch and Walter Pitts, is a simple model of a biological neuron. It is a binary threshold unit that outputs either 0 or 1 based on the weighted sum of its inputs and a bias term. The model is defined as follows:

- **Inputs:** $x_1, x_2, x_3, \dots, x_n$ (binary values, either 0 or 1)
- **Weights:** $w_1, w_2, w_3, \dots, w_n$ (real-valued weights associated with each input)
- **Bias:** θ (threshold value)

The neuron computes the weighted sum of its inputs:

$$y = \sum_{i=1}^n w_i x_i - \theta$$

The output Y of the neuron is determined by applying a step function (also called a threshold or activation function):

$$Y = f(y) = \begin{cases} 1 & \text{if } y \geq 1 \\ 0 & \text{if } y < 0 \end{cases}$$

Limitations of McCulloch-Pitts Neurons for XOR: The McCulloch-Pitts neuron is a **linear classifier**. It can represent logical operations like AND, OR, and NOT, which are linearly separable. However, the XOR (exclusive OR) function is **not linearly separable**. This means that a single-layer network of McCulloch-Pitts neurons cannot solve the XOR problem.

XOR Truth Table:

| Input x_1 | Input x_2 | Target t |
|-------------|-------------|------------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Solving XOR Using a Multi-Layer Neural Network: To solve the XOR problem, we need a **multi-layer neural network** that introduces non-linearity. In a multi-layer network, the hidden layer can transform the input space into a new space where the XOR function becomes linearly separable.

Architecture for XOR using McCulloch-Pitts Neurons:

The simplest neural network architecture to solve the XOR problem consists of:

- **Input Layer:** Two neurons, corresponding to the two inputs x_1 and x_2
- **Hidden Layer:** Two neurons to learn intermediate representations.
- **Output Layer:** One neuron to produce the final output.

Step-by-Step Logic:

1. Hidden Layer Neurons:

- One neuron computes the AND function of inputs.
- Another neuron computes the OR function of inputs.

2. Output Layer Neuron:

- The output neuron computes a logical function that combines the outputs of the hidden layer neurons. In the XOR case, this can be achieved by combining the outputs using a logical AND with negation or OR functions.

The hidden layer adds non-linearity to the model, enabling the XOR function to be learned.

Network Equations:

1. Hidden Layer Outputs:

$$h_1 = \text{AND}(x_1, x_2) = x_1 \wedge x_2$$

$$h_2 = \text{OR}(x_1, x_2) = x_1 \vee x_2$$

2. Output Layer Output:

$$o = \text{AND}(\text{NOT}(h_1), h_2) = \neg(x_1 \wedge x_2) \wedge (x_1 \vee x_2)$$

This logic creates a multi-layer neural network capable of computing the XOR function.

Convergence Curves and Decision Boundary:

- **Convergence Curves:** These show how the predicted output values approach the true XOR output values during training. For the McCulloch-Pitts model, weights are typically fixed, so convergence curves aren't as relevant unless you consider a dynamic learning scenario.
- **Decision Boundary:** The decision boundary separates the input space into regions where the output is either 0 or 1. For the XOR function, the decision boundary is non-linear. Visualizing it shows two distinct regions in the input space where the function outputs 1, separated by a region where it outputs 0.

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

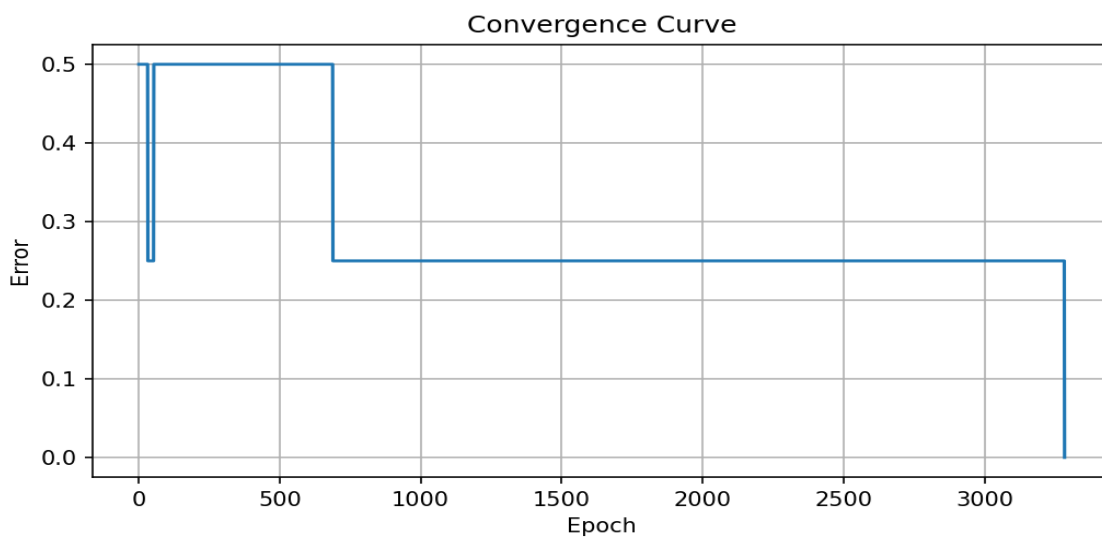
# Plot convergence curve
plt.figure(figsize=(8, 4))
plt.plot(range(1, len(convergence_curve) + 1), convergence_curve)
plt.xlabel('Epoch')
plt.ylabel('Error')
plt.title('Convergence Curve')
plt.grid()
plt.show()

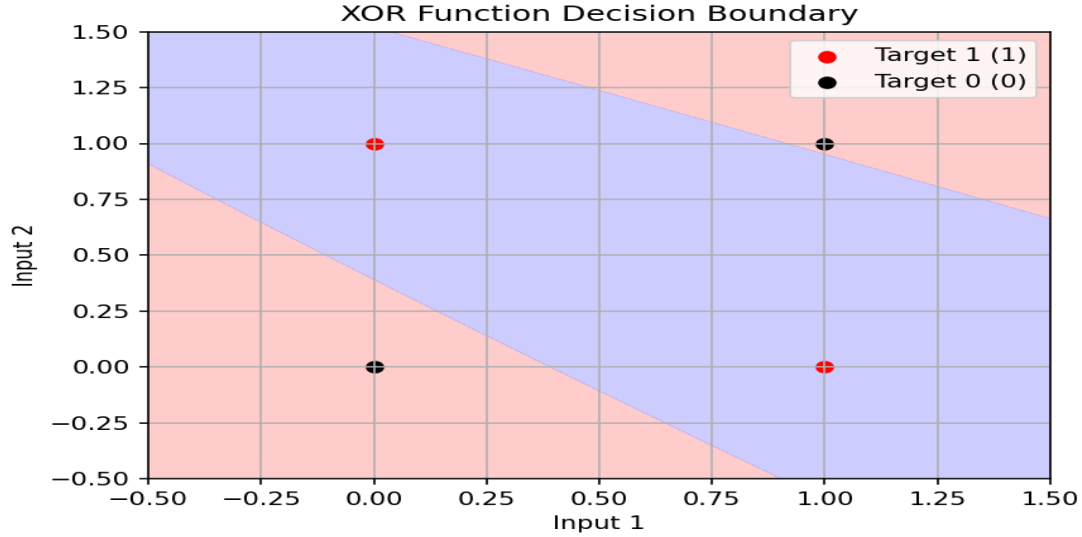
# Create a grid of points to plot decision boundaries
x1 = np.linspace(-0.5, 1.5, 200)
x2 = np.linspace(-0.5, 1.5, 200)
X1, X2 = np.meshgrid(x1, x2)
Z = predict(X1, X2)
# Plot decision boundaries
plt.figure(figsize=(8, 6))
plt.xlabel('Input 1')
plt.ylabel('Input 2')
plt.title('XOR Function Decision Boundary')
plt.legend()
plt.grid()
plt.show()

```

Output:

Generated XOR Result is: [0 1 1 0]





Problem No: 03

Problem Name: Implement the SGD Method using Delta learning rule for following input-target sets. $X_{Input} = [0\ 0\ 1; 0\ 1\ 1; 1\ 0\ 1; 1\ 1\ 1]$, $D_{Target} = [0; 0; 1; 1]$.

Theory: To implement the Stochastic Gradient Descent (SGD) method using the Delta learning rule, we need to build a simple perceptron model that updates its weights iteratively based on the error between the predicted output and the target output.

Delta Learning Rule: The Delta learning rule, also known as the Widrow-Hoff rule or Least Mean Squares (LMS) rule, is a supervised learning rule used for adjusting the weights of a single-layer neural network. The rule minimizes the difference between the desired output and the predicted output by making incremental adjustments to the weights.

The weight update formula using the Delta rule is given by:

$$\Delta w_i = \eta \cdot (D - Y) \cdot x_i$$

where:

- Δw_i is the change in the weight w_i .
- η is the learning rate, a small positive constant that determines the step size.
- D is the desired target output.
- Y is the predicted output of the perceptron.
- x_i is the input feature corresponding to the weight w_i .

The weights are updated iteratively as:

$$w_i = w_i + \Delta w_i$$

Stochastic Gradient Descent (SGD): Stochastic Gradient Descent (SGD) is an optimization algorithm commonly used in training machine learning models, especially neural networks. Unlike Batch Gradient Descent, which computes the gradient of the entire dataset, SGD updates the model weights for each training example, making it faster and more efficient for large datasets.

Steps in SGD:

1. Initialize weights randomly.
2. For each training sample:
 - Compute the output of the perceptron using the current weights.
 - Calculate the error (difference between the predicted output and the actual target).
 - Update the weights using the Delta rule.
3. Repeat the process for a specified number of epochs or until convergence.

SGD is particularly useful when the dataset is large, as it reduces computation time by updating weights based on a single sample or a small batch of samples.

Activation Functions: An activation function in a perceptron decides whether a neuron should be activated or not by calculating a weighted sum and applying a non-linear transformation. Common activation functions include:

- **Step Function:** Produces binary output based on a threshold. It is often used in simple perceptrons.

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

- **Sigmoid Function:** Produces an output between 0 and 1, making it suitable for probabilities and differentiable, which is useful for backpropagation in more complex neural networks.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

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

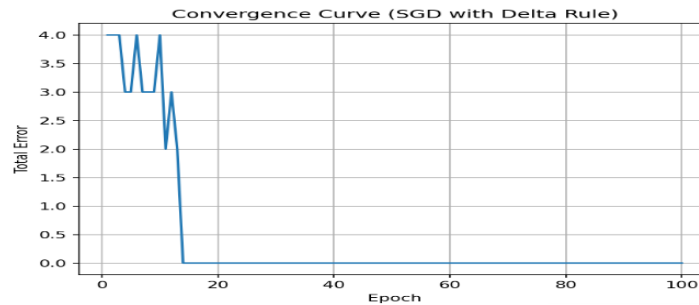
    convergence_curve.append(total_error)
# Stop early if there is no error
if total_error == 0 and converged == False:
    print(f"Converged in {epoch + 1} epochs.")
    converged = True
# Print final weights
print("Final weights:", weights)
# Plot convergence curve
plt.plot(range(1, len(convergence_curve) + 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]



Problem No: 04

Problem Name: Compare the performance of SGD and the Batch method using the delta learning rule.

Theory: When comparing the performance of **Stochastic Gradient Descent (SGD)** and the **Batch Gradient Descent** method using the **delta learning rule**, several aspects such as speed of convergence, computational cost, and stability need to be considered. Here's an overview of how these two methods differ:

1. Stochastic Gradient Descent (SGD): SGD is a type of gradient descent where the model parameters are updated for each training example, one at a time.

Update Rule: For a single training example, the update rule in SGD with the delta learning rule is:

$$\theta := \theta - \eta \nabla L(\theta; x_i)$$

where θ represents the model parameters, η is the learning rate, $L(\theta; x_i)$ is the loss function for the i -th example, and $\nabla L(\theta; x_i)$ is its gradient.

Convergence: Since SGD updates the parameters more frequently, it can often converge faster than batch methods, especially for large datasets. However, it may not settle as smoothly around the minimum and could oscillate near it due to the noisy updates.

Computational Efficiency: SGD can be more computationally efficient per iteration because it requires only one example to compute the gradient and update the parameters. This makes SGD suitable for large datasets where the batch method might be infeasible.

Advantages:

- Faster convergence for large datasets.
- More efficient memory usage.
- Better generalization due to the inherent noise in updates, which may help avoid local minima.

Disadvantages:

- The noise in updates can cause oscillations and make it harder to converge precisely.
- The learning rate might need careful tuning to balance convergence speed and stability.

2. Batch Gradient Descent (BGD): Batch Gradient Descent updates the model parameters after computing the gradient of the loss function with respect to all training examples.

Update Rule: For all training examples, the update rule using the delta learning rule is:

$$\theta := \theta - \eta \nabla L(\theta; X)$$

where $L(\theta; X)$ is the loss function over the entire dataset X , and $\nabla L(\theta; X)$ is its gradient.

Convergence: Since BGD uses the entire dataset to compute gradients, the updates are smoother and more stable. However, it can take a longer time to converge, especially for large datasets.

Computational Efficiency: BGD can be computationally expensive for large datasets because it requires the computation of the gradient using all training examples in each iteration. This can lead to slow iterations and may require more memory.

Advantages:

- Stable convergence, with smooth updates that consistently move toward the global minimum.
- Less noisy updates than SGD.

Disadvantages:

- Slower convergence for large datasets.
- High memory and computational costs for large datasets.
- Can get stuck in local minima without additional strategies like momentum or adaptive learning rates.

3. Comparison and Summary:

Speed: SGD generally converges faster for large datasets since it updates parameters more frequently. BGD converges slower as it requires full passes through the dataset.

Stability: BGD offers more stable convergence due to averaging over the entire dataset, whereas SGD introduces noise in updates, which can cause oscillations around the minimum.

Computational Cost: SGD is more memory and computationally efficient per iteration, especially suitable for large datasets. BGD requires more memory and computation since it processes the entire dataset in each iteration.

Suitability: SGD is preferred for large datasets and online learning scenarios, while BGD is more suited for smaller datasets where computational cost is manageable and precise convergence is desired.

In summary, the choice between SGD and BGD using the delta learning rule depends on the dataset size, computational resources, and the specific requirements for convergence speed and stability in the given application.

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.]]

Problem No: 05

Problem Name: 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.

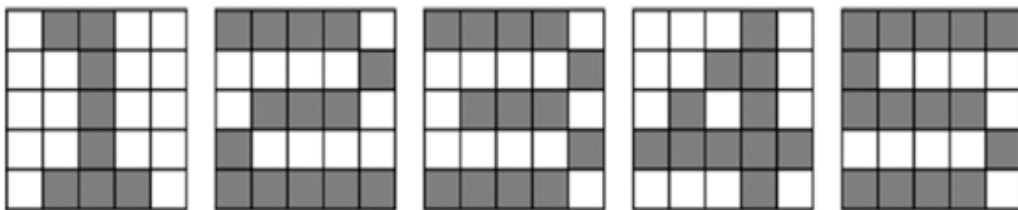


Figure 1: Five-by-five pixel squares that display five numbers from 1 to 5

Theory: Digit recognition is a classic problem in the field of machine learning and computer vision. In this context, the problem involves identifying digits from very simple 5x5 pixel images. Each image is a grid where each cell can be either black (foreground) or white (background), resulting in a binary image with values of 0 or 1.

Image Representation: In this problem, each 5x5 pixel image is represented as a vector of length 25. This is because a 5x5 grid can be flattened into a single array of 25 elements. For example, a 5x5 image (for digit 1) is converted into a vector as follows:

Image:

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

Vector:

[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]

Machine Learning Classification: The goal of digit recognition is to classify an input image vector into one of the predefined classes (digits 1 through 5). We can use various machine learning algorithms for this classification task.

K-Nearest Neighbors (KNN): KNN is a simple and effective classification algorithm. The core idea is to classify a new data point based on the majority class of its nearest neighbors in the feature space.

- **Training:** The algorithm stores all training samples and their labels.
- **Prediction:** To predict the class of a new sample, the algorithm finds the k closest training samples (neighbors) and assigns the most common class label among them to the new sample.

Pros:

- Simple and easy to understand.
- No explicit training phase.

Cons:

- Computationally expensive during prediction, especially with large datasets.
- Performance depends on the choice of k and the distance metric.

Other Classifiers: While KNN is used here for simplicity, other classifiers such as Support Vector Machines (SVM), Decision Trees, and Neural Networks can be more effective, especially for larger datasets or more complex problems.

Data Preparation: The steps to prepare and use the data are as follows:

1. **Data Collection:** Collect a dataset of 5x5 images with their corresponding digit labels.
2. **Flattening:** Convert each 5x5 image into a 25-element vector.
3. **Splitting:** Divide the dataset into training and testing sets to evaluate model performance.
4. **Training:** Train the classifier on the training set.
5. **Testing and Evaluation:** Test the classifier on the testing set and measure its accuracy.

Evaluation Metrics: To evaluate the performance of the classifier, common metrics include:

- **Accuracy:** The percentage of correctly classified samples out of the total number of samples. It is calculated as:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

- **Confusion Matrix:** A matrix that shows the number of true positives, false positives, true negatives, and false negatives. It provides more insight into which classes are being confused.

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

Problem No: 06

Problem Name: Write a MATLAB or Python program to classify face/fruit/bird using Convolution Neural Network (CNN).

Theory: A **Convolutional Neural Network (CNN)** is a type of Deep Learning neural network architecture commonly used in Computer Vision. Computer vision is a field of Artificial Intelligence that enables a computer to understand and interpret the image or visual data.

In a regular Neural Network there are three types of layers:

1. Input layer
2. Hidden layers
3. Output layer

Convolutional Neural Network (CNN) is the extended version of artificial neural networks (ANN) which is predominantly used to extract the feature from the grid-like matrix dataset. For example visual datasets like images or videos where data patterns play an extensive role.

CNN architecture: Convolutional Neural Network consists of multiple layers like the input layer, Convolutional layer, Pooling layer, and fully connected layers.

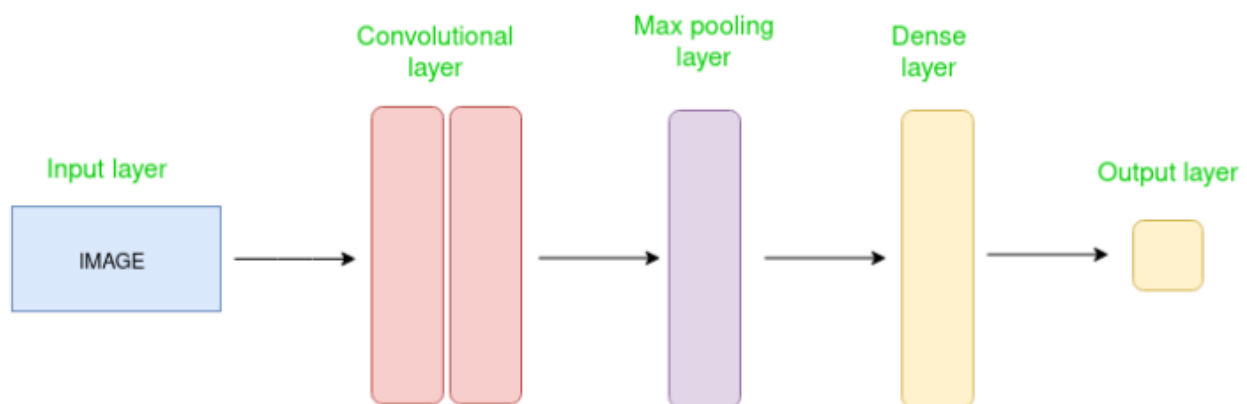


Figure-1: Simple CNN architecture

The Convolutional layer applies filters to the input image to extract features, the Pooling layer downsamples the image to reduce computation, and the fully connected layer makes the final prediction. The network learns the optimal filters through backpropagation and gradient descent.

Classification of images using Convolution Neural Network (CNN): In a classification task using a convolutional neural network (CNN), the goal is to categorize input data into predefined classes. CNNs are particularly effective for image classification due to their ability to capture spatial hierarchies in data. Here's a general outline of how you might approach a classification task with a CNN:

1. Data Preparation:
 - a. Collect Data
 - b. Preprocess Data
2. Define the CNN Architecture:
 - a. Input Layer
 - b. Convolutional Layers

- c. Activation Function
 - d. Pooling Layers
 - e. Fully Connected Layers
 - f. Output Layer
3. Compile the Model:
 - a. Loss Function
 - b. Optimizer
 - c. Metrics
 4. Train the Model
 5. Evaluate the Model
 6. Tune and Improve
 7. Deploy the Model

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')

```

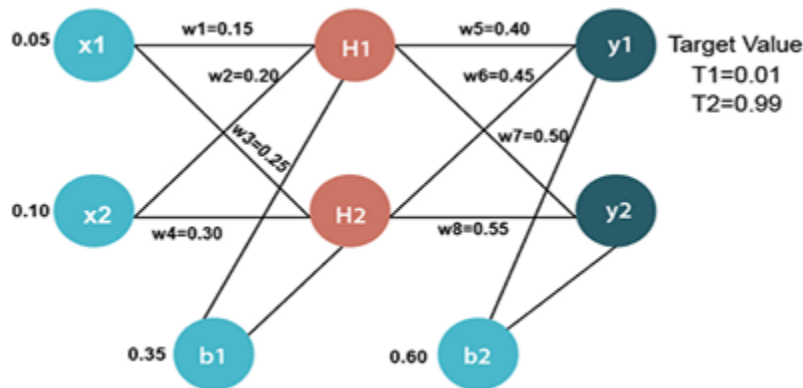
Output:

Prediction accuracy is: [[0.9634724]]

This is Cucumber

Problem No: 07

Problem Name: 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.



Theory:

Artificial Neural Network (ANN): An Artificial Neural Network (ANN) is a computational model inspired by the way biological neural networks in the human brain process information. ANNs consist of interconnected layers of nodes (neurons), where each connection represents a synapse with an associated weight.

The basic components of an ANN include:

1. **Input Layer:** Receives input signals (features).
2. **Hidden Layer(s):** Intermediate layers that process the input data and extract patterns.
3. **Output Layer:** Produces the final output (predictions or classifications).

Structure of the Neural Network Given in the Diagram:

The given neural network is a **three-layer network**:

1. **Input Layer:**
 - Two input nodes: x_1 and x_2 with values 0.05 and 0.10.
2. **Hidden Layer:**
 - Two hidden neurons: H_1 and H_2 with biases b_1 and b_2 set to 0.35 each.
 - Weights between input and hidden layer:

$$w_1 = 0.15, w_2 = 0.20, w_3 = 0.25, w_4 = 0.30.$$

3. **Output Layer:**
 - Two output neurons: y_1 and y_2 with biases 0.60.
 - Weights between hidden and output layer:

$$w_5 = 0.40, w_6 = 0.45, w_7 = 0.50, w_8 = 0.55.$$

4. **Target Values:**

- The target values for the output nodes are $T1 = 0.01$ and $T2 = 0.99$.

Backpropagation Algorithm: Backpropagation is an algorithm used to train neural networks, by minimizing the error between the predicted output and the actual target values. It involves calculating the gradient of the loss function with respect to each weight by the chain rule, propagating backward through the network.

1. Compute Output Layer Error:

- Calculate the error term for the output layer:

$$\delta_{\text{output}} = (T_i - O_i) \cdot \sigma'(O_i)$$

where $\sigma'(O_i)$ is the derivative of the activation function (sigmoid in our case).

2. Compute Hidden Layer Error:

- Backpropagate the error to the hidden layer:

$$\delta_{\text{hidden}} = \delta_{\text{output}} \cdot w_{\text{output-hidden}} \cdot \sigma'(\text{Output}_H)$$

3. Update Weights and Biases:

- Adjust the weights and biases using the gradient descent algorithm:

$$w_{\text{new}} = w_{\text{old}} + \eta \cdot \delta \cdot \text{Input}$$

$$b_{\text{new}} = b_{\text{old}} + \eta \cdot \delta$$

where η is the learning rate.

Learning Rate (η): The learning rate (η) is a hyperparameter that determines the step size at each iteration while moving toward the minimum of the loss function. A smaller learning rate ensures more precise convergence but takes longer, whereas a larger learning rate might converge faster but can overshoot the minimum.

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 diagram
with 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 epochs
    if 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

Problem No: 08

Problem Name: Write a MATLAB or Python program to recognize the numbers 1 to 4 from speech signal using artificial neural network (ANN).

Theory:

Speech Signal Processing:

Speech Signal Characteristics:

- **Time Domain:** Speech signals can be represented in the time domain, where they are simply waveforms varying over time.
- **Frequency Domain:** To analyze the content of speech signals, especially for recognizing phonetic content, it's useful to convert these signals into the frequency domain.

Feature Extraction:

- **MFCC (Mel-Frequency Cepstral Coefficients):** MFCCs are a popular feature extraction technique for speech processing. They capture the short-term power spectrum of a sound and are effective in representing the characteristics of speech. They are computed by:
 - Applying a Fourier transform to the signal to get the frequency components.
 - Mapping these frequencies to the Mel scale (a scale based on human ear perception).
 - Taking the logarithm of the power spectrum and applying a Discrete Cosine Transform (DCT) to obtain MFCCs.

Artificial Neural Networks (ANNs):

Basic Concept:

- **Neurons and Layers:** An ANN is composed of layers of neurons. Each neuron receives input, applies a weight, and passes it through an activation function.

- **Feedforward Neural Networks:** The simplest type of ANN where data moves in one direction—from input to output.

Learning Process:

- **Training:** The network learns by adjusting weights based on the error between the predicted and actual outputs. This process is usually performed using backpropagation and optimization algorithms like Gradient Descent.
- **Activation Functions:** Functions such as ReLU (Rectified Linear Unit) and Softmax are used to introduce non-linearity and to output probabilities, respectively.

Application to Speech Recognition:

Problem Formulation:

- **Input Data:** In this case, the input data are MFCC features extracted from speech recordings.
- **Output Data:** The output is a classification task where each audio sample is categorized into one of four classes (representing the numbers 1 to 4).

Model Architecture:

- **Input Layer:** Accepts the feature vector (MFCCs).
- **Hidden Layers:** Layers with neurons that help the network learn complex patterns. In the example provided, Dense layers are used.
- **Output Layer:** Uses the Softmax activation function to output a probability distribution over the classes (1 to 4).

Training:

- **Loss Function:** The model uses Sparse Categorical Crossentropy, suitable for multi-class classification.
- **Optimizer:** An optimizer like Adam adjusts the weights of the network during training.

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."

Problem No: 09

Problem Name: Write a MATLAB or Python program to Purchase Classification Prediction using SVM.

Theory: Classification is a supervised learning task where the goal is to predict the categorical label of new instances based on a training dataset with known labels. In the context of purchase classification, the objective might be to predict whether a customer will purchase a product (e.g., yes/no) based on features such as demographic information, past purchase behavior, etc.

Support Vector Machines (SVM): SVM is a popular classification technique used in machine learning. It works by finding a hyperplane that best separates the data into different classes. The main concepts involved are:

- **Hyperplane:** In an n -dimensional space, a hyperplane is an $(n-1)$ -dimensional plane that separates the space into two halves. For instance, in a 2D space, a hyperplane is a line; in 3D space, it is a plane.
- **Support Vectors:** These are the data points that are closest to the hyperplane and are critical in defining the position and orientation of the hyperplane.
- **Margin:** The margin is the distance between the hyperplane and the nearest support vectors. SVM aims to maximize this margin to create a robust classifier.

SVM Types:

- **Linear SVM:** Used when the data is linearly separable. The algorithm finds a linear hyperplane that separates the classes.
- **Non-Linear SVM:** Used when the data is not linearly separable. The algorithm uses kernel functions to transform the data into a higher-dimensional space where a linear separation is possible.

Application in Purchase Classification: In the context of purchase classification, the SVM model can be used to predict whether a customer will make a purchase based on various features. For example, features might include:

- **Demographics:** Age, income, etc.
- **Behavioral Data:** Past purchase history, browsing behavior, etc.

Conclusion: Using SVM for purchase classification is effective for many practical scenarios where you need to classify instances based on feature vectors. SVM's ability to handle both linear and non-linear data through different kernels makes it a versatile tool in machine learning. Proper preprocessing, model training, and evaluation are crucial for developing an accurate and reliable classifier.

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

R^2 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

Problem No: 10

Problem Name: Write a MATLAB or Python program to reduce dimensions of a dataset into a new coordinate system using PCA algorithm.

Theory:

Dimensionality Reduction Using PCA: Principal Component Analysis (PCA) is a statistical method used for dimensionality reduction while preserving as much variance as possible in the data. It transforms the original dataset into a new coordinate system such that the greatest variances by any projection of the data come to lie on the first coordinate (called the first principal component), the second greatest variances on the second coordinate, and so on.

The necessity of PCA:

- **High-Dimensional Data:** In many real-world datasets, the number of features can be very high (hundreds or thousands). This high dimensionality can lead to challenges in data processing, visualization, and computational costs.
- **Curse of Dimensionality:** High-dimensional data can lead to overfitting in machine learning models and make them less generalizable.
- **Visualization:** Reducing the number of dimensions helps in visualizing the data, especially when reduced to 2 or 3 dimensions.
- **Noise Reduction:** PCA helps to filter out the noise by ignoring less significant components, improving the performance of machine learning algorithms.

Theoretical Foundation of PCA:

1. **Centering the Data:** PCA begins by centering the data around the mean. This step is crucial to ensure that PCA does not attribute a significant amount of variance to any particular feature that simply has a high mean. Centering involves subtracting the mean of each feature from the data points:

$$X_{Centered} = X - mean(X)$$

2. **Covariance Matrix Computation:** The covariance matrix is computed to understand the relationships between different features. For a dataset X with n samples and d features, the covariance matrix C is a $d \times d$ matrix that is calculated as:

$$C = \frac{1}{n-1} (X^T X)$$

where each element c_{ij} represents the covariance between the i -th and j -th features.

3. **Eigenvalues and Eigenvectors:** PCA involves finding the eigenvalues and eigenvectors of the covariance matrix. The eigenvectors determine the directions (principal components) of the new feature space, while the eigenvalues determine their magnitude (importance).

4. **Selecting Principal Components:** The principal components are ranked by their corresponding eigenvalues in descending order. The eigenvectors associated with the largest eigenvalues represent the directions where the data varies the most. The number of components to retain is often determined based on the amount of variance one wishes to retain (e.g., 95% of total variance).
5. **Transforming the Data:** The original data is projected onto the selected principal components to obtain a new, reduced-dimension dataset. This transformation can be expressed as:

$$X_{reduced} = X_{centered} \cdot W$$

where W is the matrix of eigenvectors corresponding to the top k eigenvalues (principal components).

6. **Explained Variance:** The proportion of variance that each principal component explains is given by:

$$\text{Explained Variance Ratio} = \frac{\text{Eigenvalue of the Component}}{\text{Sum of all Eigenvalues}}$$

This helps in understanding how much information (variance) is retained by each component.

Advantages of PCA:

- **Dimensionality Reduction:** Reduces the number of dimensions while retaining the most important information.
- **Noise Reduction:** Helps remove noise and redundancy from the data.
- **Improved Performance:** Reduces computational costs and improves the performance of machine learning models.
- **Better Visualization:** Makes it possible to visualize high-dimensional data in 2D or 3D space.

Limitations of PCA:

- **Linear Assumptions:** PCA assumes linear relationships between features; it may not perform well on data with non-linear relationships.
- **Loss of Interpretability:** The principal components are linear combinations of the original features, which can make them difficult to interpret.
- **Sensitive to Scaling:** PCA is sensitive to the relative scaling of the original variables; therefore, data needs to be standardized before applying PCA.

Conclusion: PCA is a powerful tool for dimensionality reduction that helps in visualizing, interpreting, and improving the performance of machine learning models on high-dimensional data. By retaining only the most significant principal components, PCA effectively reduces noise and computational cost while retaining the core information.

Source Code:

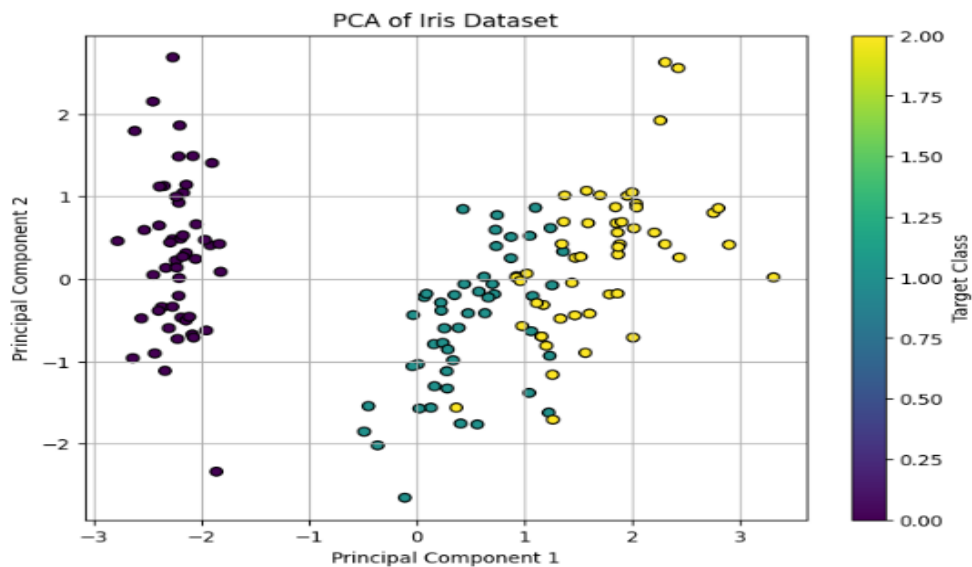
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
# Load the Iris dataset
```

```

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