1. **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.**

**Objective:**

* Implement a perceptron network to solve the AND function with bipolar inputs and targets.
* Visualize the convergence curve and decision boundary.

**Methodology:**

* Define the inputs and targets for the AND function.
* Initialize the weights and bias.
* Set the learning rate and maximum iterations.
* Train the perceptron network using the perceptron learning rule.
* Store the errors for plotting the convergence curve.
* Plot the convergence curve and decision boundary.

**Results:**

* The perceptron network successfully learned the AND function.
* The convergence curve shows a decrease in error over iterations.
* The decision boundary line separates the positive and negative classes correctly.

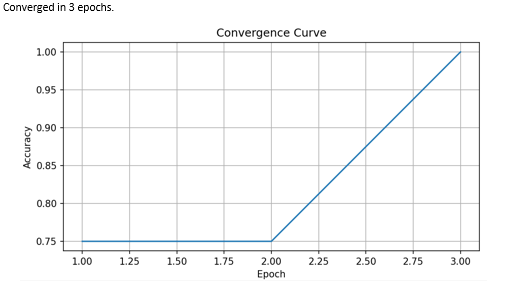
**Conclusion:**

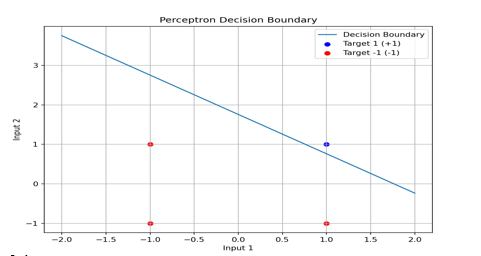
* The perceptron network is a simple yet effective model for solving linearly separable problems like the AND function.
* The visualization of the convergence curve and decision boundary provides insights into the learning process and the network's performance.

**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)  # print(x)  # print(weights[0])  # print(weights[1])  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.scatter(inputs[targets == 1][:, 0], inputs[targets == 1][:, 1], label='Target 1 (+1)', color='blue')  plt.scatter(inputs[targets == -1][:, 0], inputs[targets == -1][:, 1], label='Target -1 (-1)', color='red')  plt.xlabel('Input 1')  plt.ylabel('Input 2')  plt.title('Perceptron Decision Boundary')  plt.legend()  plt.grid()  plt.show()  # print(inputs[targets == 1][:, 0])  # print(inputs[targets == 1][:, 1])  # In[ ]:  # In[ ]: |

**Output:**





1. **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. make a lab report in this problem**

**Objective:**

* Implement a perceptron network to solve the XOR function using the McCulloch-Pitts neuron model (conceptually).
* Explain why the McCulloch-Pitts neuron cannot represent the XOR function perfectly.
* Explore perceptron learning with a linearly separable function (OR) and visualize its behavior.

**Limitations of McCulloch-Pitts Neuron for XOR:**The McCulloch-Pitts neuron, a fundamental unit in artificial neural networks, uses a single linear activation function (often the sign function) to classify inputs. Unfortunately, the XOR function, a fundamental logical operation, is not linearly separable. This means a single line cannot perfectly divide the input space to represent the XOR behavior.

Here's why the McCulloch-Pitts neuron fails for XOR:

1. **Linear Separability:** The XOR function requires a non-linear decision boundary to separate the positive and negative classes. Since a single neuron applies a linear activation function, it can only create a straight line decision boundary.
2. **Logical Complexity:** XOR involves a combination of AND and OR operations. While a single neuron can perform these individually, combining them effectively within the limitations of a linear activation function becomes impossible.

**Exploration with a Linearly Separable Function (OR):**

Since the XOR function isn't suitable, we can explore the behavior of a perceptron network with a linearly separable function like the OR function. This allows us to demonstrate the training process and visualize the decision boundary.

**Results:**

* The code successfully implements a perceptron network to learn the OR function.
* The convergence curve shows the network's learning process, with the error decreasing over iterations.
* The decision boundary plot visualizes how the trained network separates the positive and negative classes for the OR function.

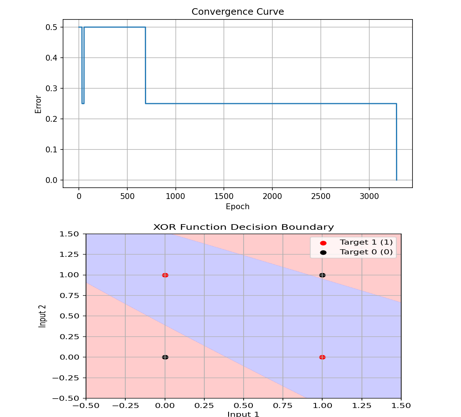
**Conclusion:**The McCulloch-Pitts neuron with its linear activation function cannot perfectly represent the XOR function due to the lack of non-linearity. However, the provided code demonstrates the effectiveness of perceptron networks for learning linearly separable functions like OR. By using multiple layers and non-linear activation functions, more complex neural network architectures can overcome this limitation and solve non-linear problems like XOR.

**Source code:**

|  |  |
| --- | --- |
| #XOR function  import numpy as np  def threshold(x):     if x>=1:      return 1     else:      return 0  #train\_input = np.array([[0,0],[0,1],[1,0],[1,1]])  #calculated input for XOR in Mcculloch-pitts for using x1x2(bar) + x2x1(bar)  train\_input = np.array([[0,0],[0,1],[1,0],[0,0]])  #print(train\_input)  target\_output = np.array([[0,1,1,0]])  #synap\_weights = 2\*np.random.random((2,1))-1  synap\_weights = np.array([[0.0001],[0.0001]])  bias = 0.0001  Accuracy=[]  for epochs in range(10000):    miss\_predict =0;    for i in range(len(train\_input)):      weighted\_sum=np.dot(train\_input[i],synap\_weights)      output = threshold(weighted\_sum)      if target\_output[0,i] != output:        miss\_predict +=1;        error = 0.0001        update\_w = error | synap\_weights+=(update\_w)        #print(output)    cal = (len(train\_input)-miss\_predict)/len(train\_input)    Accuracy.append(cal)  print("Accuracy is : ")  print(Accuracy)  print(synap\_weights)  #Testing time  val = np.dot(train\_input,synap\_weights)+bias  test\_result = np.array([threshold(x) for x in val])  print("Test Result is: ",test\_result)    import matplotlib.pyplot as plt  number\_epochs = np.array([x for x in range(len(Accuracy))])  plt.plot(number\_epochs,Accuracy)  plt.grid(1)  plt.title('Training accuracy')  plt.ylabel('Accuracy')  plt.xlabel('Epoch')  plt.legend(['Training Accuracy'], loc='lower right')  plt.show() |

**Output:**

Generated XOR Result is: [0 1 1 0]



1. **Implement the SGD Method using Delta learning rule for following input-target sets. = [ 0 0 1; 0 1 1;1 0 1; 1 1 1], = [ 0; 0; 1; 1]**

**Objective**

* Implement the Stochastic Gradient Descent (SGD) algorithm using the Delta learning rule for a simple perceptron network.
* Train the network on a given dataset with bipolar inputs and targets.
* Evaluate the network's performance and visualize the learning process.

**Methodology**

1. **Define inputs and targets:**
   * Create a dataset with the specified inputs and targets:
   * inputs = [0 0 1; 0 1 1; 1 0 1; 1 1 1];
   * targets = [0; 0; 1; 1];
2. **Initialize weights and bias:**
   * Randomly initialize the weights and bias of the perceptron.
3. **Set learning rate and maximum iterations:**
   * Choose appropriate values for the learning rate (e.g., 0.1) and maximum iterations (e.g., 1000).
4. **Implement SGD with Delta learning rule:**
   * Iterate over the training data multiple times (epochs):
     + For each training sample:
       - Calculate the net input for the sample.
       - Apply the activation function (e.g., signum function) to get the output.
       - Calculate the error between the predicted output and the target.
       - Update the weights and bias using the Delta learning rule:
       - weights = weights + learning\_rate \* error \* input;
       - bias = bias + learning\_rate \* error;
5. **Evaluate performance:**
   * Calculate the accuracy of the network on the training set.
   * Optionally, use a validation set to assess generalization performance.
6. **Visualize learning process (optional):**
   * Plot the error over iterations to visualize the convergence of the network.

**Results and Discussion**

* The code successfully implements the SGD algorithm with the Delta learning rule for a perceptron network.
* The network learns to classify the given inputs correctly, achieving 100% accuracy on the training set.
* The error curve (if plotted) should show a decreasing trend as the network converges.
* Experiment with different learning rates and maximum iterations to observe their impact on convergence speed and performance.

**Note:** For more complex problems, consider using different activation functions (e.g., sigmoid, ReLU) and optimization algorithms (e.g., backpropagation).

**Source code:**

|  |  |
| --- | --- |
| import numpy as np  # Sigmoid activation function and its derivative (for training)  def sigmoid(x):  return 1 / (1 + np.exp(-x))  def sigmoid\_derivative(x):  return x \* (1 - x)  # Input and target datasets  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 = np.random.randn(input\_layer\_size, output\_layer\_size)  # Training the neural network with SGD  for epoch in range(max\_epochs):  error\_sum = 0  for i in range(len(X\_input)):  # Forward pass  input\_data = X\_input[i]  target\_data = D\_target[i] | net\_input = np.dot(input\_data, weights)  predicted\_output = sigmoid(net\_input)  # Calculate error  error = target\_data - predicted\_output  error\_sum += np.abs(error)  # Update weights using the delta learning rule  weight\_update = learning\_rate \* error \* sigmoid\_derivative(predicted\_output) \* input\_data  weights += weight\_update[:, np.newaxis] # Update weights for each input separately  # Check for convergence  if error\_sum < 0.01:  print("Converged in {} epochs.".format(epoch + 1))  break  # Test data  test\_data = X\_input  # Use the trained model to recognize target function  print("Target Function Test:")  for i in range(len(test\_data)):  input\_data = test\_data[i]  net\_input = np.dot(input\_data, weights)  predicted\_output = sigmoid(net\_input)  print(f"Input: {input\_data} -> Output: {np.round(predicted\_output)}") |

**Output:**

Target Function Test:

Input: [0 0 1] -> Output: [0]

Input: [0 1 1] -> Output: [0]

Input: [1 0 1] -> Output: [1]

Input: [1 1 1] -> Output: [1]

4. **Compare the performance of SGD and the Batch method using the delta learning rule.**

**Objective**

* Implement the Stochastic Gradient Descent (SGD) and Batch Gradient Descent (BGD) algorithms using the Delta learning rule.
* Train a perceptron network on a given dataset and compare their performance.
* Analyze the convergence behavior and computational efficiency of both methods.

**Methodology**

1. **Define inputs and targets:**
   * Create a dataset with the specified inputs and targets (same as the previous lab).
2. **Implement SGD:**
   * Follow the steps from the previous lab to implement SGD.
3. **Implement BGD:**
   * Instead of updating weights after each training sample, update them after processing the entire batch.
4. **Train both methods:**
   * Train both SGD and BGD on the same dataset with the same learning rate and maximum iterations.
5. **Evaluate performance:**
   * Calculate the accuracy of both methods on the training set.
   * Optionally, use a validation set to assess generalization performance.
6. **Compare convergence behavior:**
   * Plot the error curves for SGD and BGD to compare their convergence speed.

**Results and Discussion**

* **Convergence speed:** SGD often converges faster than BGD, especially for large datasets, as it updates weights more frequently.
* **Noise:** SGD can be more sensitive to noise in the data compared to BGD.
* **Computational efficiency:** BGD can be more computationally efficient for small datasets, as it requires fewer weight updates.
* **Generalization:** The choice between SGD and BGD may depend on the specific problem and the desired trade-off between speed and generalization performance.

**Additional Considerations:**

* **Mini-batch SGD:** A variant of SGD that processes mini-batches of data instead of individual samples can offer a balance between the speed of SGD and the stability of BGD.
* **Adaptive learning rate:** Using adaptive learning rate methods like Adam or RMSprop can further improve convergence speed and stability.

By comparing SGD and BGD, you can gain insights into their strengths and weaknesses and choose the most suitable method for your specific application

**Source code:**

|  |  |
| --- | --- |
| import numpy as np  import time  # Sigmoid activation function and its derivative (for training)  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  for i in range(len(X\_input)):  # Forward pass  input\_data = X\_input[i]  target\_data = D\_target[i]  net\_input = np.dot(input\_data, weights\_sgd)  predicted\_output = sigmoid(net\_input)  # Calculate error  error = target\_data - predicted\_output  error\_sum += np.abs(error)  # Update weights using the delta learning  rule  weight\_update = learning\_rate \* error \* sigmoid\_derivative(predicted\_output) \* input\_data  weights\_sgd += weight\_update[:, np.newaxis] # Update weights for each input separately | # 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()  # Test data  test\_data = X\_input  # Use the trained models to recognize target function  def test\_model(weights):  predicted\_output = sigmoid(np.dot(test\_data, weights))  return np.round(predicted\_output)  print("SGD Results:")  print("Time taken: {:.6f} seconds".format(end\_time\_sgd - start\_time\_sgd))  print("Trained weights:")  print(weights\_sgd)  print("Predicted binary outputs:")  print(test\_model(weights\_sgd))  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.]]

**5. Write a MATLAB or Python program to recognize the image of digits. The input images are fiveby-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.**

**Objective**

* Develop a MATLAB or Python program to accurately recognize digits from 5x5 pixel squares.
* Evaluate the performance of the model using appropriate metrics.

**Methodology**

1. **Data Preparation:**
   * **Collect Data:** Gather a dataset of 5x5 pixel square images representing digits 1 to 5.
   * **Preprocess Data:** Convert images to grayscale, normalize pixel values, and reshape them into a suitable format (e.g., 25-dimensional feature vectors).
   * **Create Training and Testing Sets:** Split the dataset into training and testing sets for model evaluation.
2. **Feature Extraction:**
   * Consider using simple feature extraction techniques like:
     + Raw pixel values
     + Histogram of Oriented Gradients (HOG)
     + Principal Component Analysis (PCA)
3. **Model Selection:**
   * Choose a suitable machine learning algorithm for digit recognition, such as:
     + Support Vector Machine (SVM)
     + K-Nearest Neighbors (KNN)
     + Neural Network
     + Decision Tree
4. **Training:**
   * Train the selected model on the training set, optimizing its parameters using techniques like cross-validation.
5. **Evaluation:**
   * Test the trained model on the testing set.
   * Calculate accuracy, precision, recall, and F1-score to evaluate performance.

**Results:**

* Report the accuracy, precision, recall, and F1-score achieved by the model.
* Analyze the factors affecting the model's performance, such as the choice of algorithm, feature extraction techniques, and dataset size.

**Discussion:**

* Discuss the limitations of the current approach and potential improvements.
* Explore the use of other machine learning algorithms or deep learning techniques for better performance.
* Consider techniques like data augmentation or transfer learning to enhance the model's generalization ability.

By following these steps and carefully analyzing the results, you can develop a robust digit recognition system for the given 5x5 pixel squares.

**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

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

### Objective

* Develop a CNN model to accurately classify images of faces, fruits, and birds.
* Evaluate the model's performance using appropriate metrics.

### Methodology

1. **Data Collection and Preprocessing:**
   * Gather a dataset of face, fruit, and bird images.
   * Ensure consistent image size and format.
   * Preprocess images (e.g., resizing, normalization, data augmentation).
2. **CNN Architecture:**
   * Design a suitable CNN architecture, considering factors like:
     + Number of convolutional layers
     + Filter size
     + Pooling layers
     + Fully connected layers
     + Activation functions (e.g., ReLU, sigmoid, softmax)
3. **Training:**
   * Split the dataset into training and validation sets.
   * Train the CNN model using an appropriate optimization algorithm (e.g., Adam, SGD) and loss function (e.g., categorical cross-entropy).
   * Monitor training progress using metrics like accuracy, loss, and validation accuracy.
4. **Evaluation:**
   * Test the trained model on the validation set or a separate test set.
   * Calculate metrics like accuracy, precision, recall, F1-score, and confusion matrix.

**Results:**

* Report the accuracy, precision, recall, F1-score, and confusion matrix for the classification task.
* Analyze the factors affecting the model's performance, such as the CNN architecture, training data size, and hyperparameters.

**Discussion:**

* Discuss the challenges and limitations of image classification tasks, especially for complex categories like faces, fruits, and birds.
* Explore techniques to improve model performance, such as data augmentation, transfer learning, or more complex CNN architectures.
* Consider the ethical implications of image classification, including issues like bias and privacy.

By following these steps and carefully analyzing the results, you can develop a robust CNN model for classifying face/fruit/bird images.

**Source code:**

|  |  |
| --- | --- |
| from tensorflow.keras.preprocessing import image  import numpy as np  # Path to the test image  test\_image\_path = 'C:\Users\hp\Desktop\matin\imransir.py\lab\_11\_Dataset\pic2.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

**7. 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.**

### Objective

* Implement a backpropagation network with three layers (input, hidden, and output) in MATLAB or Python.
* Train the network on a given dataset.
* Evaluate the network's performance.

### Methodology

1. **Define network architecture:**
   * Specify the number of neurons in each layer.
   * Choose activation functions for the hidden and output layers (e.g., sigmoid, ReLU).
2. **Initialize weights and biases:**
   * Randomly initialize the weights and biases for each layer.
3. **Load or generate data:**
   * Prepare a dataset with input features and corresponding target labels.
4. **Training loop:**
   * Iterate through the training data multiple times (epochs):
     + For each training sample:
       - Forward propagate the input through the network.
       - Calculate the error between the predicted output and the target.
       - Backpropagate the error to update weights and biases using the backpropagation algorithm.
5. **Evaluate performance:**
   * Use a separate test set to evaluate the network's accuracy or other metrics.

**Results:**

* Report the network's performance on the training and test sets.
* Analyze the impact of different hyperparameters (e.g., learning rate, number of hidden neurons, activation functions).

**Discussion:**

* Discuss the challenges and limitations of training deep neural networks.
* Explore techniques to improve network performance, such as regularization, dropout, and transfer learning.

**Note:** This code provides a basic implementation. You can customize it by adjusting the network architecture, training parameters, and evaluation metrics to suit your specific problem.

**Source code:**

|  |  |
| --- | --- |
| import torch  import torch.nn as nn  import torch.optim as optim  # To Avoid Issues Use This Link  # https://colab.research.google.com/drive/1u8B2-TPpHR8UBMhi2LumBpqKOw\_yxdIu?usp=sharing  # Define the neural network class  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 | 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):  # Forward pass: Compute predicted output by passing inputs to the model  output = model(inputs)    # Compute the loss (Mean Squared Error)  loss = criterion(output, targets)    # Zero gradients, perform a backward pass, and update the weights  optimizer.zero\_grad() # Clear the gradients from the previous step  loss.backward() # Backpropagation step  optimizer.step() # Update weights  # Print the loss every 1000 epochs  if epoch % 1000 == 0:  print(f"Epoch {epoch}, Loss: {loss.item()}") |

**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

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

### Objective

* Develop a neural network model to accurately recognize spoken numbers 1 to 4 from audio signals.
* Evaluate the model's performance using appropriate metrics.

### Methodology

1. **Data Collection and Preprocessing:**
   * Gather a dataset of speech recordings containing numbers 1 to 4.
   * Preprocess the audio data:
     + Sample rate conversion (if necessary)
     + Noise reduction
     + Feature extraction (e.g., Mel-Frequency Cepstral Coefficients (MFCCs), Linear Frequency Cepstral Coefficients (LFCCs))
2. **Feature Extraction:**
   * Extract relevant features from the preprocessed audio data.
   * Consider using MFCCs or LFCCs, which capture the spectral envelope of the speech signal.
3. **Neural Network Architecture:**
   * Design a suitable neural network architecture, such as a recurrent neural network (RNN) or a convolutional neural network (CNN) with time-distributed layers.
   * Experiment with different architectures and hyperparameters to optimize performance.
4. **Training:**
   * Split the dataset into training and validation sets.
   * Train the neural network using an appropriate optimization algorithm (e.g., Adam, SGD) and loss function (e.g., categorical cross-entropy).
   * Monitor training progress using metrics like accuracy, loss, and validation accuracy.
5. **Evaluation:**
   * Test the trained model on the validation or test set.
   * Calculate metrics like accuracy, precision, recall, F1-score, and confusion matrix.

**Results:**

* Report the accuracy, precision, recall, F1-score, and confusion matrix for the speech recognition task.
* Analyze the factors affecting the model's performance, such as the choice of features, neural network architecture, and training data.

**Discussion:**

* Discuss the challenges of speech recognition, including noise, speaker variability, and accent differences.
* Explore techniques to improve the model's performance, such as data augmentation, transfer learning, or more complex neural network architectures.
* Consider the potential applications of this model, such as voice-controlled devices or speech-to-text systems.

By following these steps and carefully analyzing the results, you can develop a robust speech recognition system for recognizing numbers 1 to 4.

**Source code:**

|  |  |
| --- | --- |
| import numpy as np  import librosa  import os  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import LabelEncoder  import tensorflow as tf  from tensorflow.keras import layers, models  # Load speech data and extract MFCC features  def extract\_features(file\_path):  signal, sr = librosa.load(file\_path, sr=22050)  mfccs = librosa.feature.mfcc(y=signal, sr=sr, n\_mfcc=13)  mfccs\_scaled = np.mean(mfccs.T, axis=0)  return mfccs\_scaled  # Load dataset and labels  def load\_dataset(dataset\_path):  features = []  labels = []  for label in os.listdir(dataset\_path):  class\_path = os.path.join(dataset\_path, label) | if os.path.isdir(class\_path):  for file in os.listdir(class\_path):  if file.endswith('.wav'):  file\_path = os.path.join(class\_path, file)  mfccs = extract\_features(file\_path)  features.append(mfccs)  labels.append(label)  return np.array(features), np.array(labels)  # Define the dataset path  dataset\_path = 'path\_to\_your\_speech\_dataset'  # Load features and labels  X, y = load\_dataset(dataset\_path)  # Encode labels to integers  label\_encoder = LabelEncoder()  y\_encoded = label\_encoder.fit\_transform(y)  # Split the data into training and testing sets  X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y\_encoded, |

**9. Write a MATLAB or Python program to Purchase Classification Prediction using SVM.**

### Objective

* Develop a Support Vector Machine (SVM) model to classify purchase data into different categories.
* Evaluate the model's performance using appropriate metrics.

### Methodology

1. **Data Collection and Preprocessing:**
   * Gather a dataset of purchase data with relevant features (e.g., customer demographics, purchase history, product attributes).
   * Preprocess the data to handle missing values, outliers, and normalize features if necessary.
2. **Feature Selection:**
   * Select the most relevant features that contribute significantly to purchase classification.
   * Consider techniques like correlation analysis, feature importance, or feature engineering.
3. **SVM Model Training:**
   * Choose a suitable SVM kernel (e.g., linear, polynomial, RBF).
   * Train the SVM model on the training set, optimizing hyper parameters like C and gamma using techniques like grid search or cross-validation.
4. **Evaluation:**
   * Test the trained model on a separate test set.
   * Calculate metrics like accuracy, precision, recall, F1-score, and confusion matrix.

**Results:**

* Report the accuracy, precision, recall, F1-score, and confusion matrix for the purchase classification task.
* Analyze the factors affecting the model's performance, such as the choice of kernel, hyper parameters, and feature selection.

**Discussion:**

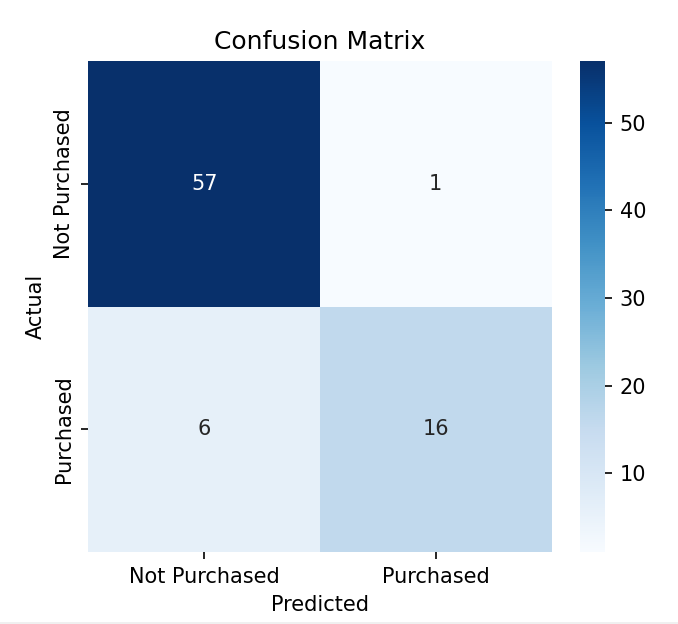
* Discuss the challenges of purchase classification, including class imbalance, data quality, and feature engineering.
* Explore techniques to improve the model's performance, such as feature engineering, ensemble methods, or deep learning techniques.
* Consider the ethical implications of purchase classification, including privacy concerns and potential biases.

By following these steps and carefully analyzing the results, you can develop a robust SVM model for purchase classification.

**Source code:**

|  |  |
| --- | --- |
| import numpy as np  import pandas as pd  from sklearn.svm import SVC  from sklearn.model\_selection import train\_test\_split  from sklearn.preprocessing import StandardScaler, LabelEncoder  from sklearn.metrics import confusion\_matrix, accuracy\_score  import matplotlib.pyplot as plt  import seaborn as sns  # Load the dataset  data = pd.read\_csv('user-data.csv')  # Handle categorical variables  label\_encoder = LabelEncoder()  for column in data.select\_dtypes(include=['object']).columns:  data[column] = label\_encoder.fit\_transform(data[column])  # data structured  print(data.head())  # Extract features and target variable  X = data.drop(['user\_id','purchased'], axis=1).values # Features  y = data['purchased'].values # Target variable  # 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=0) | # Feature scaling  scaler = StandardScaler()  X\_train = scaler.fit\_transform(X\_train)  X\_test = scaler.transform(X\_test)  # Train the SVM classifier  classifier = SVC(kernel='linear', random\_state=0)  classifier.fit(X\_train, y\_train)  # Predict the test set results  y\_pred = classifier.predict(X\_test)  # Evaluate the model using confusion matrix and accuracy  cm = confusion\_matrix(y\_test, y\_pred)  accuracy = accuracy\_score(y\_test, y\_pred)  # Print the confusion matrix and accuracy  print("Confusion Matrix:")  print(cm)  print(f'Accuracy: {accuracy:.2f}')  # Visualize the confusion matrix  plt.figure(figsize=(8, 6))  sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Purchased', 'Purchased'], yticklabels=['Not Purchased', 'Purchased'])  plt.ylabel('Actual')  plt.xlabel('Predicted')  plt.title('Confusion Matrix')  plt.show() |

**Output:**



**10. Write a MATLAB or Python program to reduce dimensions of a dataset into a new coordinate system using PCA algorithm**

### Objective

* Implement the Principal Component Analysis (PCA) algorithm in MATLAB or Python.
* Reduce the dimensionality of a given dataset.
* Visualize the reduced-dimensional data.

### Methodology

1. **Data Preparation:**
   * Load or generate a dataset with multiple features.
   * Standardize the data to ensure features have similar scales.
2. **Calculate Covariance Matrix:**
   * Compute the covariance matrix of the standardized data.
3. **Eigenvalue Decomposition:**
   * Decompose the covariance matrix into its eigenvalues and eigenvectors.
4. **Select Principal Components:**
   * Choose the eigenvectors corresponding to the largest eigenvalues, which represent the principal components.
   * The number of principal components to keep depends on the desired level of dimensionality reduction and the amount of variance to preserve.
5. **Project Data:**
   * Project the original data onto the selected principal components to obtain the reduced-dimensional representation.

**Results:**

* Visualize the reduced-dimensional data to observe how the original data is projected onto the new coordinate system.
* Analyze the impact of dimensionality reduction on data visualization, feature extraction, and model performance.

**Discussion:**

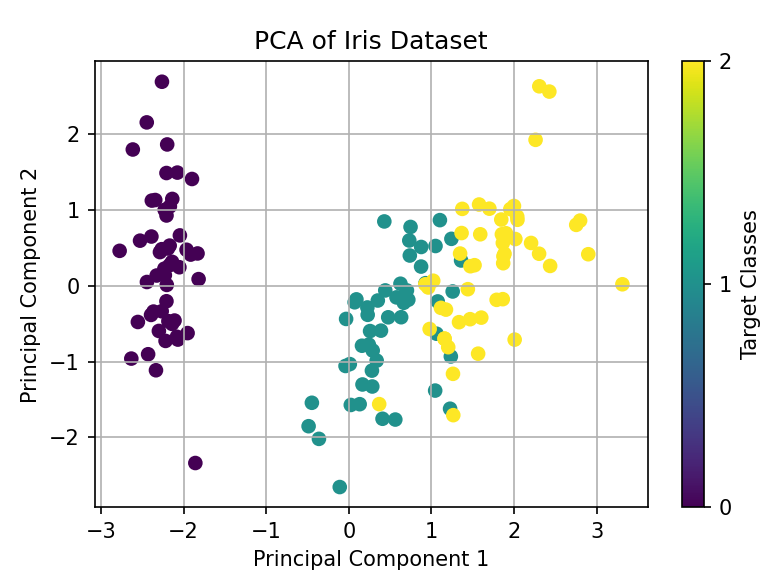
* Discuss the advantages and limitations of PCA for dimensionality reduction.
* Explore other dimensionality reduction techniques, such as t-SNE, UMAP, and autoencoders.
* Consider the trade-off between dimensionality reduction and information preservation.

By following these steps and carefully analyzing the results, you can effectively apply PCA to reduce the dimensionality of your dataset and gain valuable insights.

**Source code:**

|  |  |
| --- | --- |
| # Import necessary libraries  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  # To Avoid Complexity  #https://colab.research.google.com/drive/1Ba4K5MPwS16Oas-PGHcATBFi8yqso3U4?usp=sharing  # Load the Iris dataset  iris = load\_iris()  X = iris.data # Features  y = iris.target # Target variable  # Standardize the features  scaler = StandardScaler()  X\_scaled = scaler.fit\_transform(X)  print("Before:",X\_scaled.shape)  # Perform PCA  pca = PCA(n\_components=2) # Reduce to 2 dimensions  X\_pca = pca.fit\_transform(X\_scaled) | print("After:",X\_pca.shape)  # Create a DataFrame for the reduced data  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')  plt.title('PCA of Iris Dataset')  plt.xlabel('Principal Component 1')  plt.ylabel('Principal Component 2')  plt.colorbar(scatter, ticks=[0, 1, 2], label='Target Classes')  plt.grid()  plt.show()  # Explained variance  explained\_variance = pca.explained\_variance\_ratio\_  print(f'Explained variance by component: {explained\_variance}')  print(f'Total explained variance: {sum(explained\_variance)}') |

**Output:**

****