Experiment No : 01

Name of the Experiment: Write a Matlab or Python program using perception net for AND function with bipolar inputs and targets. The convergence curves and the decision boundary lines are also shown.

Theory:

Perceptron is one of the simplest Artificial neural network architectures. It is the simplest type of feedforward neural network, consisting of a single layer of input nodes that are fully connected to a layer of output nodes. It can learn the linearly separable patterns.

Types of Perceptron

- **Single-Layer Perceptron:** This type of perceptron is limited to learning linearly separable patterns. effective for tasks where the data can be divided into distinct categories through a straight line.

- **Multilayer Perceptron:** Multilayer perceptrons possess enhanced processing capabilities as they consist of two or more layers, adept at handling more complex patterns and relationships within the data.

A perceptron, the basic unit of a neural network, comprises essential components that collaborate in information processing.

- **Input Features:** The perceptron takes multiple input features, each input feature represents a characteristic or attribute of the input data.
- **Weights**: Each input feature is associated with a weight, determining the significance of each input feature in influencing the perceptron's output. During training, these weights are adjusted to learn the optimal values.
- **Summation Function**: The perceptron calculates the weighted sum of its inputs using the summation function. The summation function combines the inputs with their respective weights to produce a weighted sum.
- **Activation Function**: The weighted sum is then passed through an activation function. Perceptron uses Heaviside step function functions. which take the summed values as input and compare with the threshold and provide the output as 0 or 1.
- **Output:** The final output of the perceptron, is determined by the activation function's result. For example, in binary classification problems, the output might represent a predicted class (0 or 1).
- **Bias:** A bias term is often included in the perceptron model. The bias allows the model to make adjustments that are independent of the input. It is an additional parameter that is learned during training.
- **Learning Algorithm (Weight Update Rule):** During training, the perceptron learns by adjusting its weights and bias based on a learning algorithm. A common approach is the perceptron learning algorithm, which updates weights based on the difference between the predicted output and the true output.

Convergence Curve

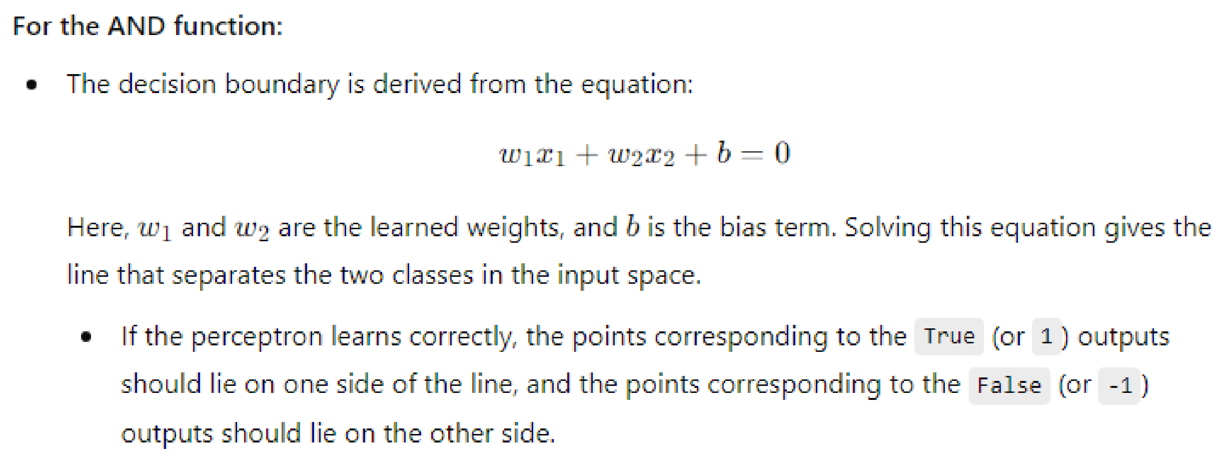
The **convergence curve** in the context of training a perceptron (or any learning model) is a plot that shows how the error rate changes over time (usually measured in epochs or iterations). It visualizes the learning process of the model and indicates whether and how quickly the perceptron is learning to classify inputs correctly.

- **X-axis**: The number of epochs (iterations over the entire dataset).
- **Y-axis**: The number of errors or some form of loss/error metric (in this case, how many misclassifications occurred in each epoch).

Decision Boundary

The **decision boundary** is a line (or hyperplane, in higher dimensions) that separates the input space into different regions corresponding to different classes. In the case of a 2D input space (like the AND function with two inputs), the decision boundary is a straight line that divides the input space into two regions:

- One region where the perceptron will classify inputs as belonging to one class (e.g., 1 or True).
- The other region where the perceptron will classify inputs as belonging to the other class (e.g., -1 or False).



For the AND function:

- The decision boundary is derived from the equation:

$$w_1x_1 + w_2x_2 + b = 0$$

  Here, w_1 and w_2 are the learned weights, and b is the bias term. Solving this equation gives the line that separates the two classes in the input space.

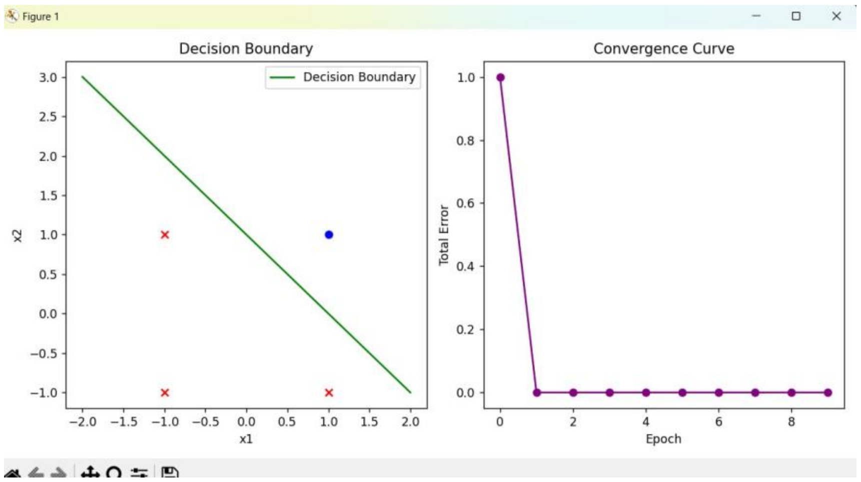
  - If the perceptron learns correctly, the points corresponding to the `True` (or `1`) outputs should lie on one side of the line, and the points corresponding to the `False` (or `-1`) outputs should lie on the other side.

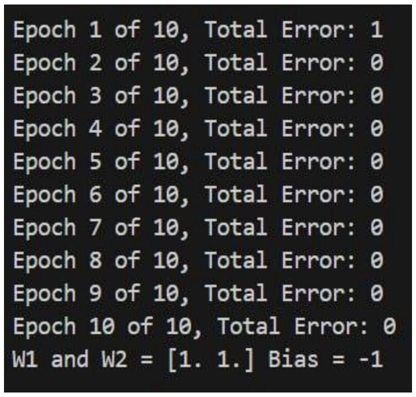
Python Code:

```
import numpy as np
import matplotlib.pyplot as plt
# Step function for bipolar input
def step_function(x):
   return np.where(x > 0, 1, np.where(x<0,-1,0))
# Perceptron training function
def perceptron_training(inputs, targets, learning_rate=1, epochs=10):
    n_samples, n_features = inputs.shape
   weights = np.zeros(n_features)
   bias = 0
   errors = []
  for epoch in range(epochs):
      total_error = 0
      for x, target in zip(inputs, targets):
         # Weighted sum
         linear_output = np.dot(x, weights) + bias
```

```
            # Perceptron output using the step function
            predicted = step_function(linear_output)
            # Update rule
            error = target - predicted
            weights += learning_rate * error * x
            bias += learning_rate * error
            total_error += abs(error)
           errors.append(total_error)
        print(f'Epoch {epoch+1} of {epochs}, Total Error: {total_error}')

    return weights, bias, errors
# Plot decision boundary and convergence curve
def plot_results(inputs, targets, weights, bias, errors):
    # Plot decision boundary
    plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
    for i, input_point in enumerate(inputs):
        if targets[i] == 1:
            plt.scatter(input_point[0], input_point[1], color='blue', marker='o')
        else:
            plt.scatter(input_point[0], input_point[1], color='red', marker='x')
      # Equation of decision boundary: w1*x1 + w2*x2 + bias = 0 => x2 = -(w1*x1 + bias)/w2
    x_values = np.linspace(-2, 2, 100)
    if weights[1] != 0:  # To avoid division by zero in case of vertical line
        decision_boundary = -(weights[0] * x_values + bias) / weights[1]
        plt.plot(x_values, decision_boundary, color='green', label='Decision Boundary')
    plt.title('Decision Boundary')
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.legend()
  # Plot convergence curve
    plt.subplot(1, 2, 2)
    plt.plot(errors, marker='o', color='purple')
    plt.title('Convergence Curve')
    plt.xlabel('Epoch')
    plt.ylabel('Total Error')
    plt.tight_layout()
    plt.show()
# Bipolar inputs for AND function
inputs = np.array([[-1, -1], [-1, 1], [1, -1], [1, 1]])
# Bipolar targets for AND function (-1 for False, 1 for True)
targets = np.array([-1, -1, -1, 1])
# Train the perceptron
learning_rate = 1
epochs = 10
weights, bias, errors = perceptron_training(inputs, targets, learning_rate, epochs)
print("W1 and W2 =",weights,"Bias =",bias)
# Plot results
plot_results(inputs, targets, weights, bias, errors)
```

Output:





```
Epoch 1 of 10, Total Error: 1
Epoch 2 of 10, Total Error: 0
Epoch 3 of 10, Total Error: 0
Epoch 4 of 10, Total Error: 0
Epoch 5 of 10, Total Error: 0
Epoch 6 of 10, Total Error: 0
Epoch 7 of 10, Total Error: 0
Epoch 8 of 10, Total Error: 0
Epoch 9 of 10, Total Error: 0
Epoch 10 of 10, Total Error: 0
W1 and W2 = [1. 1.] Bias = -1
```

Experiment No : 02

Name of the Experiment: 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:

The McCulloch-Pitts (M-P) neuron is one of the earliest models of a biological neuron, proposed in 1943 by Warren McCulloch and Walter Pitts. It is a simple threshold logic gate that processes binary inputs (either 0 or 1) and produces a binary output (0 or 1). The M-P neuron's decision-making is based on a weighted sum of inputs and a threshold.

Mathematical Model of McCulloch-Pitts Neuron:

$$y = \begin{cases} 1 & \text{if } \sum_i w_i x_i \geq \theta \\ 0 & \text{if } \sum_i w_i x_i < \theta \end{cases}$$

Where:

- x_i are the binary inputs.
- w_i are the weights associated with each input.
- θ is the threshold of the neuron.
- y is the binary output (0 or 1).

The XOR (exclusive OR) function is a two-input binary function that returns 1 only when the inputs are different.

The XOR function is not linearly separable, meaning that a single McCulloch-Pitts neuron cannot model the XOR function directly. This is because the output of the XOR is not a simple linear combination of the inputs. However, XOR can be modeled using a multi-layer network (at least 2 layers) of McCulloch-Pitts neurons.

Constructing XOR with McCulloch-Pitts Neurons

The XOR function can be expressed in terms of the basic logic functions AND, OR, and NOT, which can be implemented by McCulloch-Pitts neurons. The key observation is:

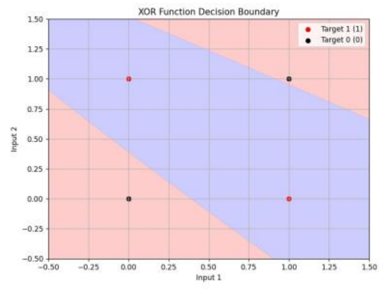
$$\text{XOR}(x_1, x_2) = (x_1 \text{ AND } \neg x_2) \text{ OR } (\neg x_1 \text{ AND } x_2)$$

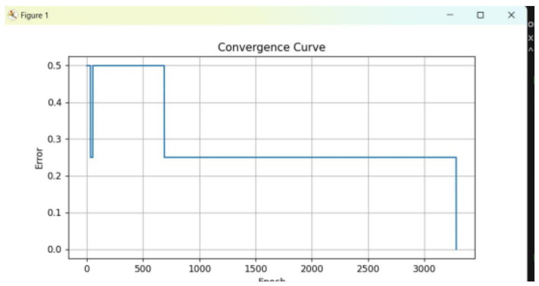
Python Code:

```
import numpy as np
import matplotlib.pyplot as plt
# 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

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)):
        # Forward pass
        hidden_layer_input = np.dot(inputs[i], weights_input_hidden) + bias_hidden
        hidden_layer_output = sigmoid(hidden_layer_input)
         output_layer_input = np.dot(hidden_layer_output, weights_hidden_output) + bias_output
        predicted_output = sigmoid(output_layer_input)
        # Backpropagation
        error = targets[i] - predicted_output
        if targets[i] != np.round(predicted_output):
            misclassified +=1
        output_delta = error * sigmoid_derivative(predicted_output)
        hidden_delta = output_delta.dot(weights_hidden_output.T) * sigmoid_derivative(hidden_layer_output)
       # 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)
# Function to predict outputs over the grid
def predict(x1, x2):
    hidden_input = np.dot(np.c_[x1.ravel(), x2.ravel()], weights_input_hidden) + bias_hidden
    hidden_output = sigmoid(hidden_input)
    output_input = np.dot(hidden_output, weights_hidden_output) + bias_output
    output = sigmoid(output_input)
    return output.reshape(x1.shape)
Z = predict(X1, X2)
# Plot decision boundaries
plt.figure(figsize=(8, 6))
plt.contourf(X1, X2, Z, levels=[0, 0.5, 1], colors=['#FFAAAA', '#AAAAFF'], alpha=0.6)
# Plot data points
plt.scatter(inputs[targets == 1][:, 0], inputs[targets == 1][:, 1], label='Target 1 (1)', color='red')
plt.scatter(inputs[targets == 0][:, 0], inputs[targets == 0][:, 1], label='Target 0 (0)', color='black')
plt.xlabel('Input 1')
plt.ylabel('Input 2')
plt.title('XOR Function Decision Boundary')
plt.legend()
plt.grid()
plt.show()
```

Output:





Experiment No : 03

Name of the Experiment: 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]

Theory:

Gradient Descent is an iterative optimization process that searches for an objective function's optimum value (Minimum/Maximum). It is one of the most used methods for changing a model's parameters in order to reduce a cost function in machine learning projects. The primary goal of gradient descent is to identify the model parameters that provide the maximum accuracy on both training and test datasets.

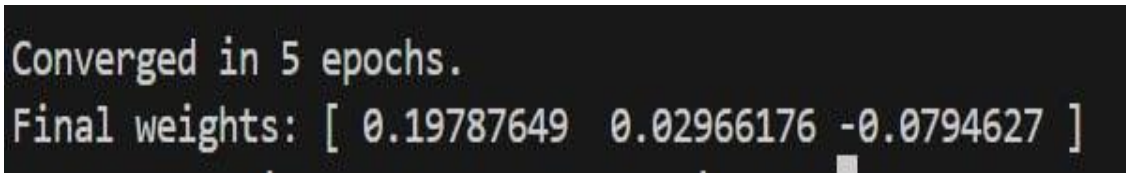
Stochastic Gradient Descent (SGD) is a variant of the Gradient Descent algorithm that is used for optimizing machine learning models. In SGD, instead of using the entire dataset for each iteration, only a single random training example (or a small batch) is selected to calculate the gradient and update the model parameters. This random selection introduces randomness into the optimization process, hence the term "stochastic" in stochastic Gradient Descent

The advantage of using SGD is its computational efficiency, especially when dealing with large datasets. By using a single example or a small batch, the computational cost per iteration is significantly reduced compared to traditional Gradient Descent methods that require processing the entire dataset.

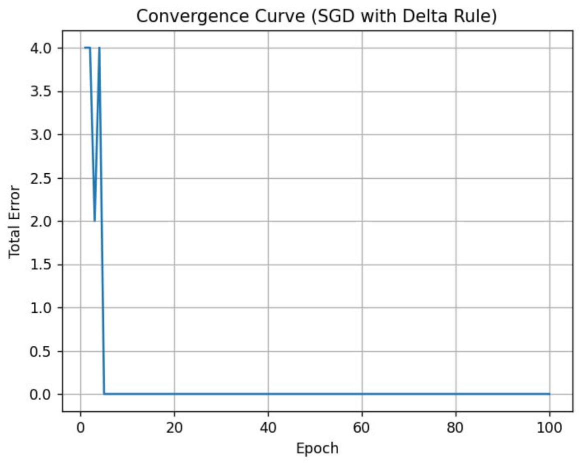
Python Code :

```
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 5 epochs.
Final weights: [ 0.19787649  0.02966176 -0.0794627 ]
```



Experiment No : 04

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

Theory:

In order to train a Linear Regression model, we have to learn some model parameters such as feature weights and bias terms. An approach to do the same is Gradient Descent which is an iterative optimization algorithm capable of tweaking the model parameters by minimizing the cost function over the train data. It is a complete algorithm i.e it is guaranteed to find the global minimum (optimal solution) given there is enough time and the learning rate is not very high. Two Important variants of Gradient Descent which are widely used in Linear Regression as well as Neural networks are Batch Gradient Descent and Stochastic Gradient Descent(SGD).

Batch Gradient Descent: Batch Gradient Descent involves calculations over the full training set at each step as a result of which it is very slow on very large training data. Thus, it becomes very computationally expensive to do Batch GD. However, this is great for convex or relatively smooth error manifolds.

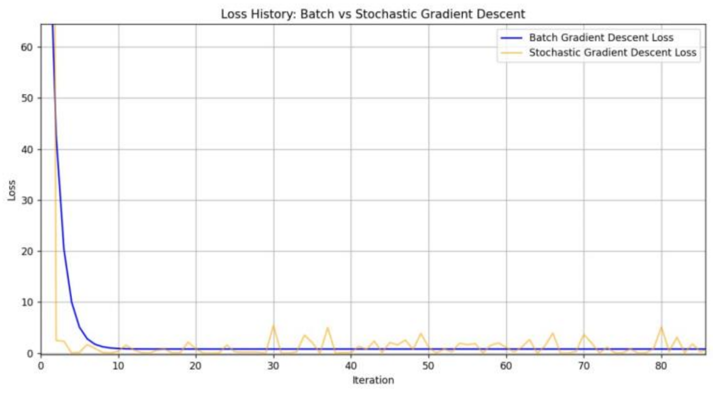
Comparison between BGD and SGD.

BGD	SGD
Computes gradient using the whole Training sample	Computes gradient using a single Training sample
Slow and computationally expensive algorithm	Faster and less computationally expensive than Batch GD
Not suggested for huge training samples.	Can be used for large training samples.
Convergence is slow.	Reaches the convergence much faster.
It typically converges to the global minimum for convex loss functions.	It may converge to a local minimum or saddle point.
It may suffer from overfitting if the model is too complex for the dataset.	It can help reduce overfitting by updating the model parameters more frequently.

Python Code:

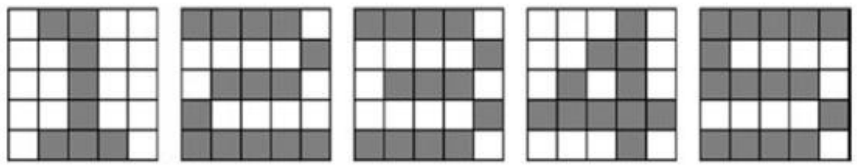
```
import numpy as np
import matplotlib.pyplot as plt
# Generate synthetic data
np.random.seed(42)
X = np.random.rand(100, 1) * 10  # 100 data points
y = 2.5 * X + np.random.randn(100, 1)  # Linear relation with noise
# Delta Learning Rule implementation for Batch Gradient Descent
class BatchGradientDescent:
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.bias = None
 def fit(self, X, y):
        m, n = X.shape
        self.weights = np.zeros((n, 1))
        self.bias = 0
        self.loss_history = []
 for _ in range(self.n_iterations):
            y_pred = X.dot(self.weights) + self.bias
            error = y_pred - y
            loss = np.mean(error ** 2)
            self.loss_history.append(loss)
           # Gradient calculation
            dw = (1/m) * X.T.dot(error)
            db = (1/m) * np.sum(error)
          # Update weights and bias
            self.weights -= self.learning_rate * dw
            self.bias -= self.learning_rate * db
 return self.weights, self.bias
# Delta Learning Rule implementation for Stochastic Gradient Descent
class StochasticGradientDescent:
    def __init__(self, learning_rate=0.01, n_iterations=1000):
        self.learning_rate = learning_rate
        self.n_iterations = n_iterations
        self.weights = None
        self.bias = None
 def fit(self, X, y):
        m, n = X.shape
        self.weights = np.zeros((n, 1))
        self.bias = 0
        self.loss_history = []
    for _ in range(self.n_iterations):
            for i in range(m):
                xi = X[i]
                yi = y[i]
                # Prediction
                y_pred = xi.dot(self.weights) + self.bias
                error = y_pred - yi
                loss = np.mean(error ** 2)
                self.loss_history.append(loss)
                # Gradient calculation
                dw = xi.T.dot(error)
                db = np.sum(error)
              # Update weights and bias
                self.weights -= self.learning_rate * dw
                self.bias -= self.learning_rate * db

        return self.weights, self.bias
# Instantiate and train both models
batch_model = BatchGradientDescent(learning_rate=0.01, n_iterations=1000)
batch_weights, batch_bias = batch_model.fit(X, y)
sgd_model = StochasticGradientDescent(learning_rate=0.01, n_iterations=1000)
sgd_weights, sgd_bias = sgd_model.fit(X, y)
# Plotting the loss history
plt.figure(figsize=(12, 6))
plt.plot(batch_model.loss_history, label='Batch Gradient Descent Loss', color='blue')
plt.plot(sgd_model.loss_history, label='Stochastic Gradient Descent Loss', color='orange', alpha=0.6)
plt.title('Loss History: Batch vs Stochastic Gradient Descent')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.legend()
plt.grid()
plt.show()
```

Output:



Experiment No : 05

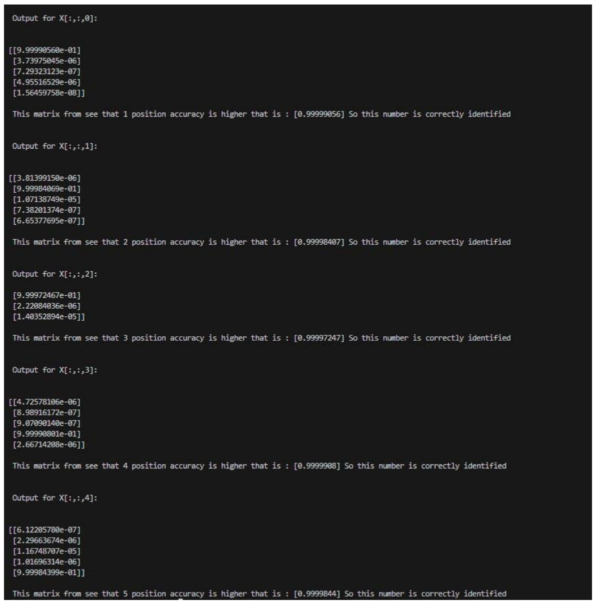
Name of the Experiment: Write a MATLAB or Python program to recognize the image of digits. The input images are five by-five pixel squares, which display five numbers from 1 to 5, as shown in Figure 1.



Python 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]:

 [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
```

Experiment No : 06

Name of the Experiment: 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 algorithm that is particularly well-suited for image recognition and processing tasks. It is made up of multiple layers, including convolutional layers, pooling layers, and fully connected layers. The architecture of CNNs is inspired by the visual processing in the human brain, and they are well-suited for capturing hierarchical patterns and spatial dependencies within images.

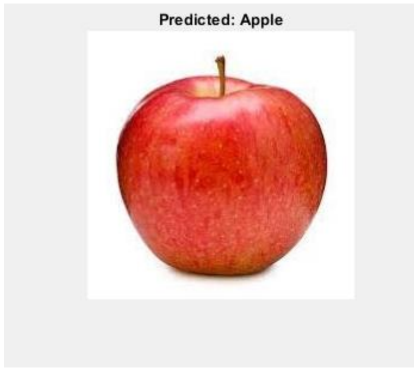
Key components of a Convolutional Neural Network include:

1. Convolutional Layers: These layers apply convolutional operations to input images, using filters (also known as kernels) to detect features such as edges, textures, and more complex patterns. Convolutional operations help preserve the spatial relationships between pixels.
2. Pooling Layers: Pooling layers downsample the spatial dimensions of the input, reducing the computational complexity and the number of parameters in the network. Max pooling is a common pooling operation, selecting the maximum value from a group of neighboring pixels.
3. Activation Functions: Non-linear activation functions, such as Rectified Linear Unit (ReLU), introduce non-linearity to the model, allowing it to learn more complex relationships in the data.
4. Fully Connected Layers: These layers are responsible for making predictions based on the high-level features learned by the previous layers. They connect every neuron in one layer to every neuron in the next layer.

Python Code:

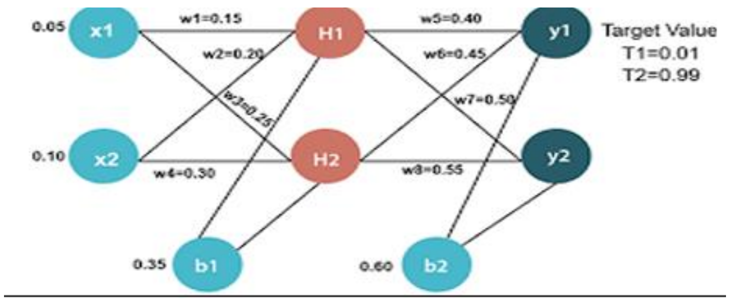
```
import os
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras import layers, models
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import load_model
from tensorflow.keras.preprocessing import image
# Set paths to your dataset
fruits_path = '/content/path_to_sample_fruit_image/path_to_sample_fruit_image/'
# Create image data generator for training dataset
datagen = ImageDataGenerator(rescale=1./255, validation_split=0.2)
# Create training and validation generators
train_generator = datagen.flow_from_directory(
    fruits_path,
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical',
    subset='training'
)
validation_generator = datagen.flow_from_directory(
    fruits_path,
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical',
    subset='validation'
)
# Define CNN architecture
model = models.Sequential([
    layers.Input(shape=(224, 224, 3)),
    layers.Conv2D(32, (3, 3), padding='same', activation='relu'),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(64, (3, 3), padding='same', activation='relu'),
    layers.MaxPooling2D(pool_size=(2, 2)),
    layers.Conv2D(128, (3, 3), padding='same', activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(train_generator.num_classes, activation='softmax')  # numClasses is the number of fruit classes
])

# Compile the model
model.compile(optimizer=Adam(learning_rate=1e-4),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
# Train the CNN
model.fit(train_generator,
          validation_data=validation_generator,
          epochs=20,
          verbose=1)
# Save the trained model
model.save('fruit_classifier_model.h5')
# Load and preprocess a random sample fruit image for classification
sample_image_file = np.random.choice(train_generator.filepaths)
sample_image = image.load_img(sample_image_file, target_size=(224, 224))
input_image = image.img_to_array(sample_image) / 255.0  # Normalize the image
input_image = np.expand_dims(input_image, axis=0)  # Add batch dimension
# Classify the sample image
predicted_label = model.predict(input_image)
predicted_class = np.argmax(predicted_label, axis=1)
# Get the class labels from the generator
class_labels = list(train_generator.class_indices.keys())
predicted_label_name = class_labels[predicted_class[0]]
# Display the predicted label and the sample image
plt.imshow(sample_image)
plt.title(f'Predicted: {predicted_label_name}')
plt.axis('off')
plt.show()
```

Output:



Experiment No : 07

Name of the Experiment: 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.

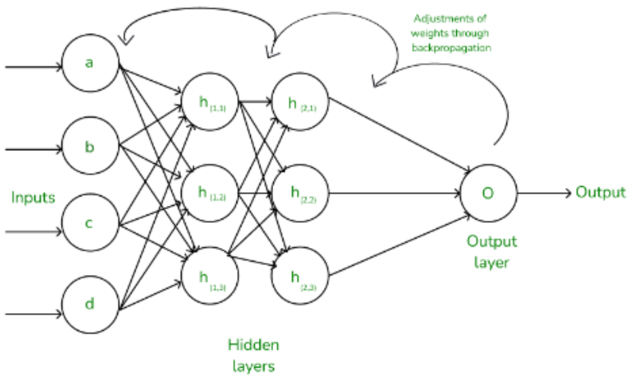


Theory:

Artificial Neural Networks contain artificial neurons which are called **units** . These units are arranged in a series of layers that together constitute the whole Artificial Neural Network in a system. A layer can have only a dozen units or millions of units as this depends on how the complex neural networks will be required to learn the hidden patterns in the dataset.

In machine learning, backpropagation is an effective algorithm used to train artificial neural networks, especially in feed-forward neural networks.

Backpropagation is an iterative algorithm, that helps to minimize the cost function by determining which weights and biases should be adjusted. During every epoch, the model learns by adapting the weights and biases to minimize the loss by moving down toward the gradient of the error. Thus, it involves the two most popular optimization algorithms, such as gradient descent or stochastic gradient descent. Computing the gradient in the backpropagation algorithm helps to minimize the cost function and it can be implemented by using the mathematical rule called chain rule from calculus to navigate through complex layers of the neural network.



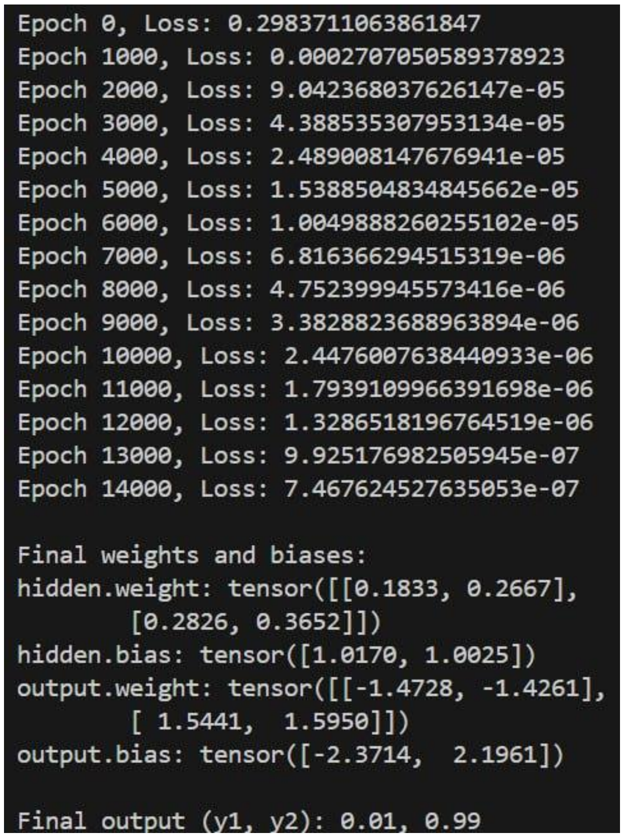
fig(a) A simple illustration of how the backpropagation works by adjustments of weights

Python 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
# 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):
    # 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()}")
# 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:



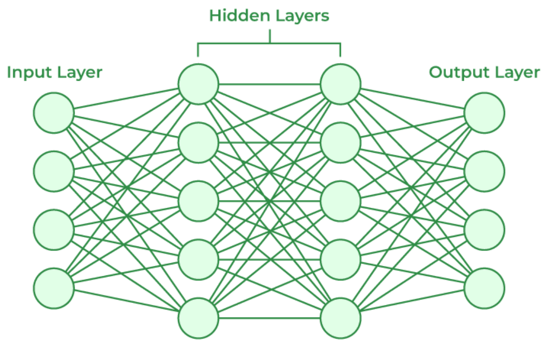
```
Epoch 0, Loss: 0.2983711063861847
Epoch 1000, Loss: 0.0002707050589378923
Epoch 2000, Loss: 9.042368037626147e-05
Epoch 3000, Loss: 4.388535307953134e-05
Epoch 4000, Loss: 2.489008147676941e-05
Epoch 5000, Loss: 1.5388504834845662e-05
Epoch 6000, Loss: 1.0049888260255102e-05
Epoch 7000, Loss: 6.816366294515319e-06
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
```

Experiment No : 08

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

Theory: Artificial Neural Networks contain artificial neurons which are called units . These units are arranged in a series of layers that together constitute the whole Artificial Neural Network in a system. A layer can have only a dozen units or millions of units as this depends on how the complex neural networks will be required to learn the hidden patterns in the dataset. Commonly, Artificial Neural Network has an input layer, an output layer as well as hidden layers. The input layer receives data from the outside world which the neural network needs to analyze or learn about. Then this data passes through one or multiple hidden layers that transform the input into data that is valuable for the output layer. Finally, the output layer provides an output in the form of a response of the Artificial Neural Networks to input data provided.

In the majority of neural networks, units are interconnected from one layer to another. Each of these connections has weights that determine the influence of one unit on another unit. As the data transfers from one unit to another, the neural network learns more and more about the data which eventually results in an output from the output layer.



Python Code:

Experiment No : 09

Name of the Experiment: Write a MATLAB or Python program to Purchase Classification Prediction using SVM.

Theory :

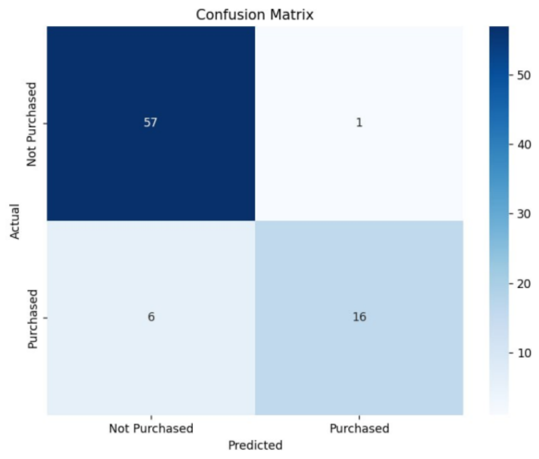
Support Vector Machine (SVM) is a supervised machine learning algorithm used for both classification and regression. Though we say regression problems as well it’s best suited for classification. The main objective of the SVM algorithm is to find the optimal hyperplane in an N-dimensional space that can separate the data points in different classes in the feature space.

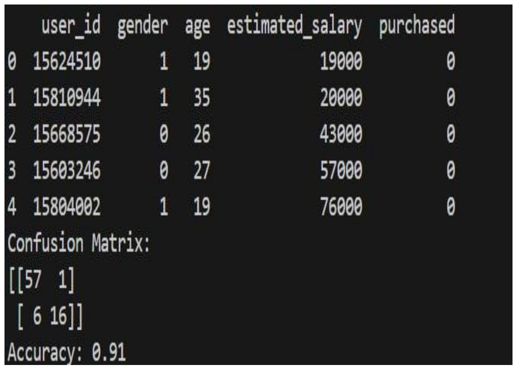
Python Code:

```
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
#Avoid Complication
# https://colab.research.google.com/drive/1hRoOy9fDLL6Gt0fL40phhDdmuipp5p8I?usp=sharing
# 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:





```
   user_id  gender  age  estimated_salary  purchased
0  15624510       1   19             19000          0
1  15810944       1   35             20000          0
2  15668575       0   26             43000          0
3  15603246       0   27             57000          0
4  15804002       1   19             76000          0
Confusion Matrix:
[[57  1]
 [ 6 16]]
Accuracy: 0.91
```

Experiment No : 10

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

Theory:

Principal Component Analysis(PCA) technique was introduced by the mathematician **Karl Pearson** in 1901**.** It works on the condition that while the data in a higher dimensional space is mapped to data in a lower dimension space, the variance of the data in the lower dimensional space should be maximum.

- Principal Component Analysis (PCA) is a statistical procedure that uses an orthogonal transformation that converts a set of correlated variables to a set of uncorrelated variables. PCA is the most widely used tool in exploratory data analysis and in machine learning for predictive models. Moreover,
- Principal Component Analysis (PCA) is an unsupervised learning algorithm technique used to examine the interrelations among a set of variables. It is also known as a general factor analysis where regression determines a line of best fit.
- The main goal of Principal Component Analysis (PCA) is to reduce the dimensionality of a dataset while preserving the most important patterns or relationships between the variables without any prior knowledge of the target variables.

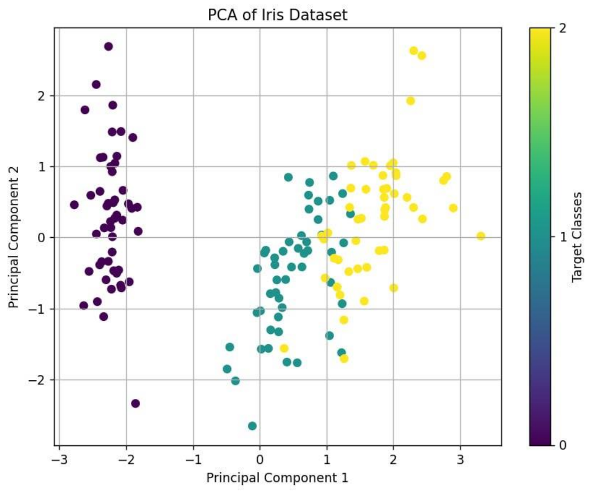
Principal Component Analysis (PCA) is used to reduce the dimensionality of a data set by finding a new set of variables, smaller than the original set of variables, retaining most of the sample's information, and useful for the regression and classification of data.

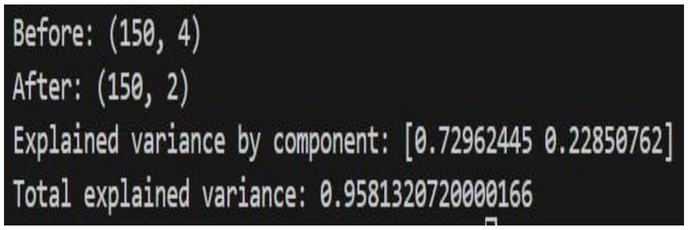
Python Code:

```
# 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:





```
Before: (150, 4)
After: (150, 2)
Explained variance by component: [0.72962445 0.22850762]
Total explained variance: 0.9581320720000166
```