Aim: Write a Python program to plot a few activation functions that are being used in neural network.

Software for Python: Jupyter Notebook

Theory:

An activation function is a mathematical function that controls the output of a neural network. Activation functions help in determining whether a neuron is to be fired or not as shown in Fig.1.

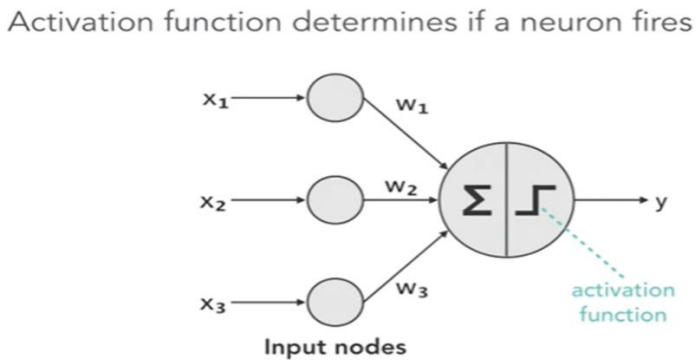


Fig.1.Activation function

Activation is responsible for adding non-linearity to the output of a neural network model. Without an activation function, a neural network is simply a linear regression.

The mathematical equation for calculating the output of a neural network is:

$$Y = \text{Activation}(\Sigma(weight * input) + bias)$$

Some of the popular activation functions are:

1. Linear
2. Sigmoid
3. ReLU
4. Tanh
5. Softmax
6. Leaky ReLU

1.Linear activation function-Linear functions are simple. It returns what it gets as input.

It is defined as f(x)=x, for all x

Here input=output

2.Sigmoid activation function-Sigmoid function returns the value between 0 and 1.

For activation function in deep learning network, Sigmoid function is considered not good since near the boundaries the network doesn't learn quickly.

This is because gradient is almost zero near the boundaries.

Sigmoid functions are so-called because their graphs are “S-shaped”.

3.RELU activation function-RELU is more well known activation function which is used in the deep learning networks. RELU is less computational expensive than the other non-linear activation functions.

RELU returns 0 if the x (input) is less than 0

RELU returns x if the x (input) is greater than 0

4.Tanh activation function-Tanh is another nonlinear activation function.

Tanh outputs between -1 and 1.

Tanh also suffers from gradient problem near the boundaries just as Sigmoid activation function does.

$$\tanh x = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

5.Softmax activation function: Softmax is an activation function that scales numbers/logits into probabilities.

The output of a Softmax is a vector (say v) with probabilities of each possible outcome.

The probabilities in vector v sums to one for all possible outcomes or classes.

Mathematically, Softmax is defined as,

$$S(y)_i = \frac{\exp(y_i)}{\sum_{j=1}^{n} \exp(y_j)}$$

where,

y	is an input vector to a softmax function, S. It consist of n elements for n classes (possible outcomes)
y_i	the i-th element of the input vector. It can take any value between -inf and +inf
$exp(y_i)$	standard exponential function applied on y_i. The result is a small value (close to 0 but never 0) if $y_i < 0$ and a large value if y_i is large. eg • $\exp(55) = 7.69e{+}23$ (A very large value) • $\exp(-55) = 1.30e\text{-}24$ (A very small value close to 0) **Note**: $\exp(*)$ is just $\mathbf{e}^*$ where $\mathbf{e} = 2.718$, the Euler's number.
$\sum_{j=1}^{n} \exp(y_j)$	A normalization term. It ensures that the values of output vector $S(y)_i$ sums to 1 for i-th class and each of them and each of them is in the range 0 and 1 which makes up a valid probability distribution.
n	Number of classes (possible outcomes)

6.Leaky relu activation function-The Leaky ReLu function is an improvisation of the regular ReLu function.

To address the problem of zero gradient for negative value, Leaky ReLu gives an extremely small linear component of x to negative inputs.

Mathematically we can express Leaky ReLu as:

f(x)= 0.01x, x<0

= x, x>=0

Mathematically:

f(x)=1 (x<0)

(αx)+1 (x>=0)(x)

Here a is a small constant like the 0.01 we've taken above.

Procedure:

1. First import the libraries matplotlib.pyplot as plt and numpy as np.

For Linear Activation function

2. Define the linear function as linear(x) and return x.
3. Define the x in the range of -10 to +10
4. Set y equal to linear (x)
5. Plot x and y
6. Label x-axis as Input and label y-axis as Output
7. Give the title as Linear Activation function.
8. Plot the grid function with the command plt.grid()
9. Display the graph of Linear activation function with the command plt.show().

For Sigmoid Activation Function

10. Define the sigmoid function as sigmoid(x) and return 1/(1+np.exp(-x)).
11. Define the x in the range of -10 to +10
12. Set y equal to sigmoid (x)
13. Plot x and y
14. Label x-axis as Input and Label y-axis as Output
15. Give the title as Sigmoid Activation function.
16. Plot the grid function with the command plt.grid()
17. Display the graph of Sigmoid activation function with the command plt.show().

For RELU activation function

18.Define the RELU function as relu(x) and return np.maximum(0,x).
19.Define the x in the range of -10 to +10

20.Set y equal to relu(x)
21.Plot x and y
22.Label x-axis as Input and label y-axis as Output
23.Give the title as ReLU Activation function.
24.Plot the grid function with the command plt.grid()
25.Display the graph of ReLU activation function with the command plt.show().

For Hyperbolic Tangent function(tanh)

18.Define the hyperbolic tangent function as tanh(x) and return np.tanh(x)
19.Define the x in the range of -10 to +10
20.Set y equal to tanh(x)
21.Plot x and y
22.Label x-axis as Input and Label y-axis as Output
23.Give the title as Tanh Activation function.
24.Plot the grid function with the command plt.grid()
25.Display the graph of tanh activation function with the command plt.show().

For Softmax Activation function

26.Define the softmax function as softmax(x) and return np.exp(x)/np.sum(np.exp(x), axis=0)
27.Define the x in the range of -10 to +10
28.Set y equal to softmax(x)
29.Plot x and y
30.Label x-axis as Input and Label y-axis as Output
31.Give the title as Softmax Activation function.
32.Plot the grid function with the command plt.grid()
33.Display the graph of softmax activation function with the command plt.show().

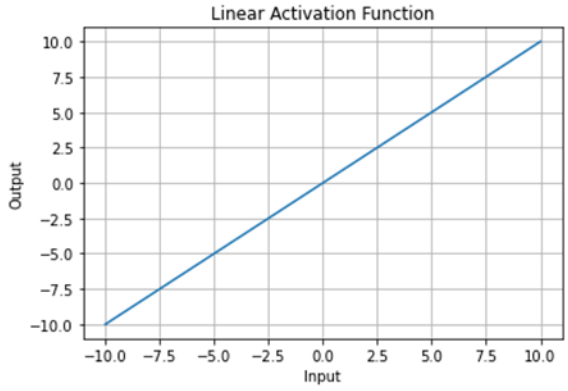
For Leaky Relu Activation function

34.Define the leaky relu function as leaky_relu(x.alpha=0.01) and return np.maxima(alpha*x,x)
35.Define the x in the range of -10 to +10
36.Set y equal to leaky_relu(x)
37.Plot x and y
38.Label x-axis as Input and Label y-axis as Output
39.Give the title as Leaky ReLU Activation function.
40.Plot the grid function with the command plt.grid()
41.Display the graph of Leaky ReLU activation function with the command plt.show().

Conclusion: Thus, the various activation functions that are being used in neural network are studied.

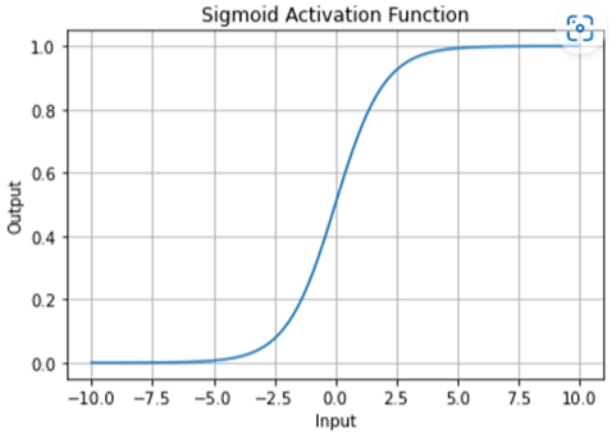
Program Code and Output

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np

def linear(x):
  return x

x = np.linspace(-10, 10, 100)
y = linear(x)

plt.plot(x, y)
plt.xlabel("Input")
plt.ylabel("Output")
plt.title("Linear Activation Function")
plt.grid()
plt.show()
```



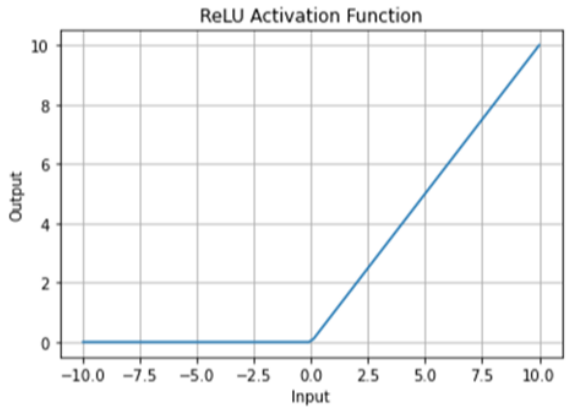
In [2]:

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

def sigmoid(x):
  return 1 / (1 + np.exp(-x))

x = np.linspace(-10, 10, 100)
y = sigmoid(x)

plt.plot(x, y)
plt.xlabel("Input")
plt.ylabel("Output")
plt.title("Sigmoid Activation Function")
plt.grid()
plt.show()
```



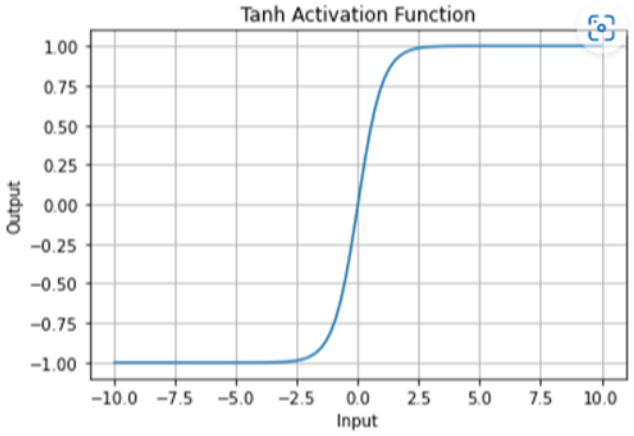
In [3]:

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

def relu(x):
  return np.maximum(0, x)

x = np.linspace(-10, 10, 100)
y = relu(x)

plt.plot(x, y)
plt.xlabel("Input")
plt.ylabel("Output")
plt.title("ReLU Activation Function")
plt.grid()
plt.show()
```

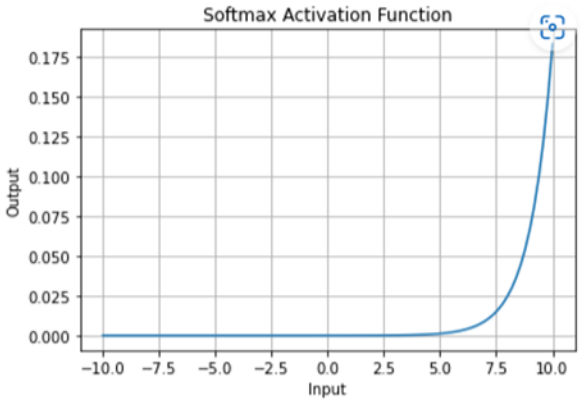


In [4]:

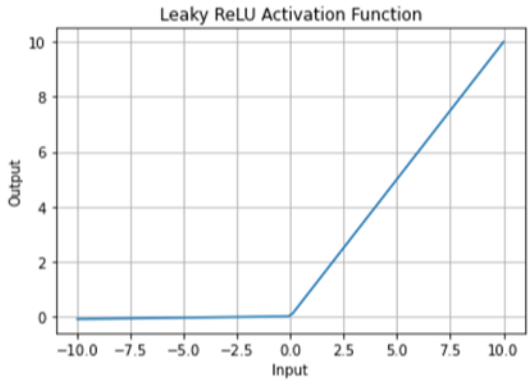
```
import matplotlib.pyplot as plt
import numpy as np

def tanh(x):
  return np.tanh(x)

x = np.linspace(-10, 10, 100)
y = tanh(x)

plt.plot(x, y)
plt.xlabel("Input")
plt.ylabel("Output")
plt.title("Tanh Activation Function")
plt.grid()
plt.show()
```



In [5]:
```
import matplotlib.pyplot as plt
import numpy as np

def softmax(x):
  return np.exp(x) / np.sum(np.exp(x), axis=0)

x = np.linspace(-10, 10, 100)
y = softmax(x)

plt.plot(x, y)
plt.xlabel("Input")
plt.ylabel("Output")
plt.title("Softmax Activation Function")
plt.grid()
plt.show()
```



In [6]:
```
import matplotlib.pyplot as plt
import numpy as np

def leaky_relu(x, alpha=0.01):
  return np.maximum(alpha * x, x)

x = np.linspace(-10, 10, 100)
y = leaky_relu(x)

plt.plot(x, y)
plt.xlabel("Input")
plt.ylabel("Output")
plt.title("Leaky ReLU Activation Function")
plt.grid()
plt.show()
```



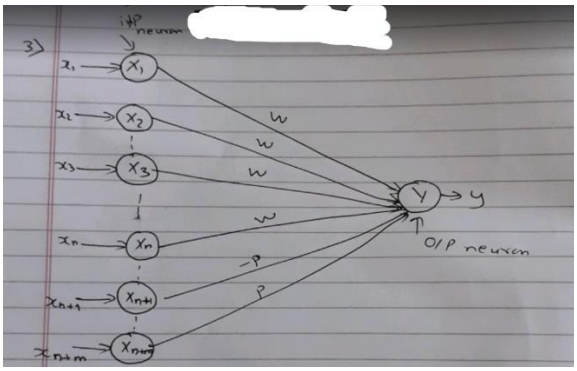
Aim-Generate ANDNOT function using McCulloch-Pitts neural net by python program.

Software for Python: Jupyter Notebook

Theory:

In McCulloch-Pitts neural net, there is a fixed threshold Θ for each neuron and if the net input to the neuron is greater than threshold then the neuron fires.

It is most widely used in logic functions.



X1,X2,X3,….,Xn,Xn+1,and Xn+m are input neurons and Y is the output neuron.

The input neurons are connected to the output neurons with excitatory weights w (w>0 or positive weights) or inhibitory weights p (p<0 or negative weights).

Since firing of output neuron is based on threshold Θ, activation function is defined as

f(yin)=1, yin≥Θ

=0, yin<Θ

If inhibitory weights are used, threshold with activation function should satisfy following condition

Θ= nw-p

n-number of input vectors

w-no of excitatory weights

p-no of inhibitory weights

1)Draw the Truth Table of ANDNOT Function

X1	X2	Y
0	0	0
0	1	0
1	0	1
1	1	0

2)From the truth table, when the first input X1=1 and second input X2=0,the neuron fires so Y=1.

3)Assume both weights w1 and w2 as excitatory, w1=w2=1

4)calculate the net input for 4 inputs using the formula

yin=x1w1+x2w2

for x1=0,x2=0, yin1=0×1+0×1=0+0=0

for x1=0, x2=1, yin2=0×1+1×1=0+1=1

for x1=1, x2=0, yin3=1×1+0×1=1+0=1

for x1=1, x2=1, yin4=1×1+1×1=1+1=2

5)If we set Θ≥1, all neurons with yin1, yin2, yin3, and yin4 will get fired as their values are greater than or equal to 1.

6)It is not possible to fire neurons for inputs x1=1 and x2-0.Hence these weights are not suitable.

7) Assume one weights w1 is excitatory, w1=1 and one weight w2 as inhibitory w2=-1 and bias b=1

8) calculate the net input for 4 inputs using the formula

yin=x1w1+x2w2

for x1=0, x2=0, yin1=0×1+0×-1=0+0=0

for x1=0, x2=1, yin2=0×1+1×-1=0-1=-1

for x1=1, x2=0, yin3=1×1+0×-1=1+0=1

for x1=1, x2=1, yin4=1×1+1×-1=1-1=0

9) Now it is possible to fire the neuron for input x1=1 and x2=0 only by fixing the threshold of 1.

Θ≤1

Thus w1=1, w2=-1, Θ≤1

Value of Θ can be calculated by equation

Θ≤nw-p

Θ≤ (2×1)-1

Θ≤2-1

Θ≤1

10)The output neuron Y can be written as

y=f(yin)=1, yin≥1

=0, yin<1

In our case yin3=1

y=f(yin3) =1

y=1

Procedure:

1. First import the library numpy as np.
2. The mcculloch_pitts_neuron function implements a single Mcculloch-Pitts neuron, which computes the dot product of the weights and input arrays and adds bias to obtain the weighted input.
3. If the weighted input is greater than or equal to zero, the function returns 1.0, otherwise it returns 0.0.
4. The andnot_neural _net function implements the ANDNOT logic gate using two McCullochs-Pitts neurons and the appropriate weight and bias.
5. The code defines the inputs x1 and x2, and runs the andnot_neural_net function for each pair of inputs to demonstrate the behavior of the ANDNOT function.

Conclusion: Thus we have studied to generate ANDNOT function using McCulloch-Pitts neural net by python program.

Program Code and Ootput

In [2]:
```
import numpy as np

def mcculloch_pitts_neuron(weights, inputs, bias):
  weighted_input = np.dot(weights, inputs) + bias
  return 1.0 if weighted_input >= 0 else 0.0

def andnot_neural_net(x1, x2):
  weights = np.array([-1, -1])
  bias = 1.0
  inputs = np.array([x1, x2])
  return mcculloch_pitts_neuron(weights, inputs, bias)

x1 = [0, 0, 1, 1]
x2 = [0, 1, 0, 1]

for i in range(len(x1)):
  print("x1: {}, x2: {}, ANDNOT: {}".format(x1[i], x2[i], andnot_neural_net(x1[i], x2[i])))
```

```
x1: 0, x2: 0, ANDNOT: 1.0
x1: 0, x2: 1, ANDNOT: 1.0
x1: 1, x2: 0, ANDNOT: 1.0
x1: 1, x2: 1, ANDNOT: 0.0
```

Aim: Write a Python program using Perceptron Neural Network to recognize even and odd numbers. Given numbers are in ASCII form from 0 to 9.

Software for Python: Jupyter Notebook

Theory:

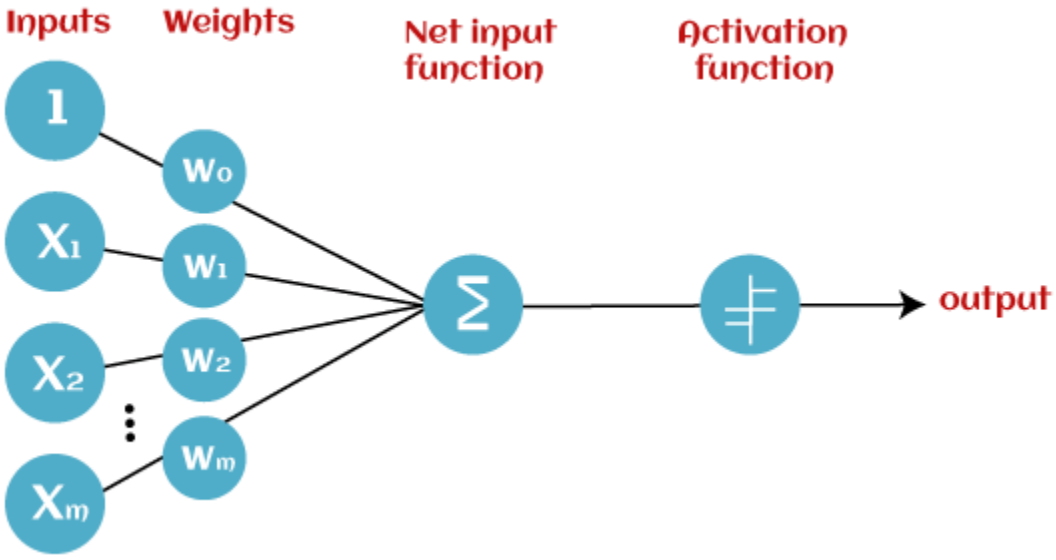
Perceptron Neural Network

Perceptron is Machine Learning algorithm for supervised learning of various binary classification tasks. Further, Perceptron is also understood as an Artificial Neuron or neural network unit that helps to detect certain input data computations in business intelligence.

Perceptron model is also treated as one of the best and simplest types of Artificial Neural networks. However, it is a supervised learning algorithm of binary classifiers. Hence, we can consider it as a single-layer neural network with four main parameters, i.e., input values, weights and Bias, net sum, and an activation function.

Basic Components of Perceptron

Mr. Frank Rosenblatt invented the perceptron model as a binary classifier which contains three main components. These are as follows:



Input Nodes or Input Layer:

This is the primary component of Perceptron which accepts the initial data into the system for further processing. Each input node contains a real numerical value.

Weight and Bias:

Weight parameter represents the strength of the connection between units. This is another most important parameter of Perceptron components. Weight is directly proportional to the strength of the

associated input neuron in deciding the output. Further, Bias can be considered as the line of intercept in a linear equation.

Activation Function:

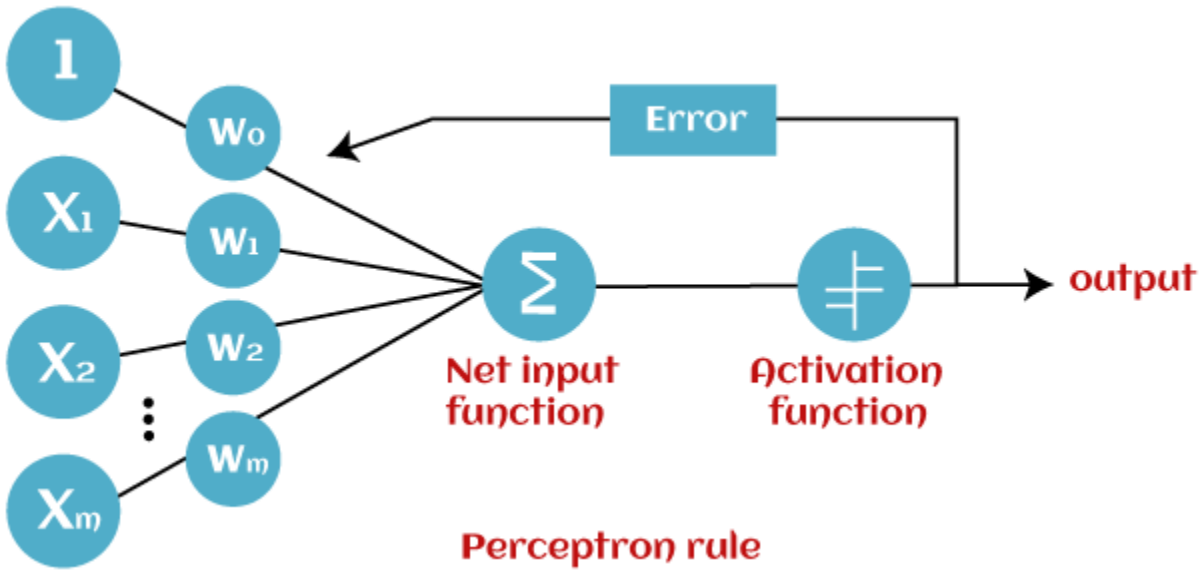
These are the final and important components that help to determine whether the neuron will fire or not. Activation Function can be considered primarily as a step function.

Types of Activation functions:

1. Sign function
2. Step function, and
3. Sigmoid function

How does Perceptron work?

In Machine Learning, Perceptron is considered as a single-layer neural network that consists of four main parameters named input values (Input nodes), weights and Bias, net sum, and an activation function. The perceptron model begins with the multiplication of all input values and their weights, then adds these values together to create the weighted sum. Then this weighted sum is applied to the activation function 'f' to obtain the desired output. This activation function is also known as the step function and is represented by 'f'.



This step function or Activation function plays a vital role in ensuring that output is mapped between required values (0,1) or (-1,1). It is important to note that the weight of input is indicative of the strength of a node. Similarly, an input's bias value gives the ability to shift the activation function curve up or down.

Perceptron model works in two important steps as follows:

Step-1

In the first step first, multiply all input values with corresponding weight values and then add them to determine the weighted sum. Mathematically, we can calculate the weighted sum as follows:

$wi*xi = x1*w1 + x2*w2 +…wn*xn$

Add a special term called bias 'b' to this weighted sum to improve the model's performance.

$\sum wi*xi + b$

Step-2

In the second step, an activation function is applied with the above-mentioned weighted sum, which gives us output either in binary form or a continuous value as follows:

$Y = f(\sum wi*xi + b)$

Procedure:

1. This code uses the Perceptron Neural Network to identify even and odd numbers.
2. The problem is solved by converting ASCII representation of numbers from 0 to 9 into binary representation and then training the Perceptron model on these inputs.
3. The sigmoid function calculates the sigmoid activation function for a given input. The sigmoid_derivative function calculates the derivative of the sigmoid function.
4. The predict function takes in the inputs and weights, computes the weighted sum, passes it through the sigmoid activation function and returns the output.
5. The train function trains the Perceptron model. It takes in the inputs, targets, initial weights, learning rate and the number of epochs as inputs.
6. It iterates over the number of epochs and updates the weights based on the error between the target and the output.
7. The even_or_odd function takes in a number and returns 0 if it is even and 1 if it is odd.
8. The ascii_to_binary function takes in a number and converts it into binary representation.
9. Finally, the inputs, targets, and weights are prepared.
10. The Perceptron model is trained using the train function and the final weights are obtained.
11. The Perceptron model is then tested on a number of test cases, and the result is printed indicating whether the ASCII representation of the number is even or odd.

Conclusion: Thus we have studied Perceptron Neural Network to recognize even and odd numbers using python programming.

Program code and Output

In [1]:
```
import numpy as np

def sigmoid(x):
  return 1.0 / (1 + np.exp(-x))

def sigmoid_derivative(x):
  return x * (1.0 - x)

def predict(inputs, weights):
  weighted_sum = np.dot(inputs, weights)
  output = sigmoid(weighted_sum)
  return output

def train(inputs, targets, weights, learning_rate, epochs):
  for i in range(epochs):
    outputs = []
    for j in range(len(inputs)):
      output = predict(inputs[j], weights)
      error = targets[j] - output
      weights += learning_rate * error * sigmoid_derivative(output) * inputs[j]
      outputs.append(output)
    print("Epoch: {}/{}".format(i+1, epochs))
    for j in range(len(inputs)):
      print("Input: {} Output: {} Target: {}".format(inputs[j], outputs[j], targets[j]))
  return weights

def even_or_odd(number):
  return 0 if int(number) % 2 == 0 else 1

def ascii_to_binary(number):
  binary = bin(int(number))[2:]
  binary = [int(digit) for digit in binary.zfill(7)]
  return binary

inputs = []
targets = []

for i in range(10):
  binary = ascii_to_binary(str(i))
  inputs.append(binary)
  targets.append(even_or_odd(i))

inputs = np.array(inputs)
targets = np.array(targets)
weights = np.random.rand(7)
```

```
Input: [0 0 0 0 0 1 0] Output: 0.374048650667305 Target: 0
Input: [0 0 0 0 0 1 1] Output: 0.8471099196871616 Target: 1
Input: [0 0 0 0 1 0 0] Output: 0.3391334988080453 Target: 0
Input: [0 0 0 0 1 0 1] Output: 0.8267801532326151 Target: 1
Input: [0 0 0 0 1 1 0] Output: 0.23255457327696114 Target: 0
Input: [0 0 0 0 1 1 1] Output: 0.7384587767589553 Target: 1
Input: [0 0 0 1 0 0 0] Output: 0.38748885380793996 Target: 0
Input: [0 0 0 1 0 0 1] Output: 0.8554733884361188 Target: 1
The ASCII representation of 0 is odd
The ASCII representation of 1 is odd
The ASCII representation of 2 is even
The ASCII representation of 3 is odd
The ASCII representation of 4 is even
The ASCII representation of 5 is odd
The ASCII representation of 6 is even
The ASCII representation of 7 is odd
The ASCII representation of 8 is even
The ASCII representation of 9 is odd
```

Aim: With a suitable example demonstrate the perceptron learning law with its decision region using python. Give the output in Graphical form.

Software for Python: Jupyter Notebook

Theory:

Perceptron Learning rule

It was introduced by Rosenblatt. It is an error-correcting rule of a single-layer feedforward network. it is supervised in nature and calculates the error between the desired and actual output and if the output is present then only adjustments of weight are done.

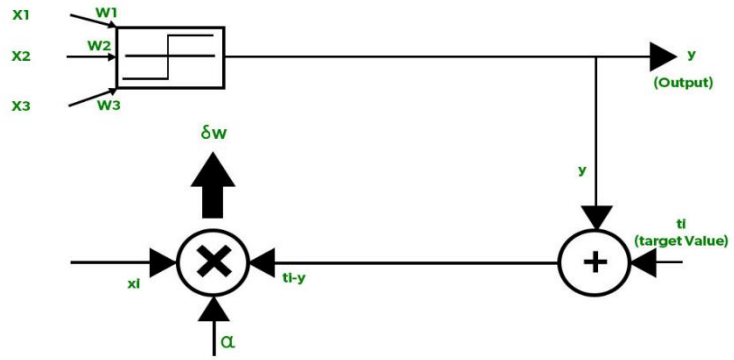


Fig.1.Perceptron Learning Rule

Computed as follows:

Assume (x1, x2, x3……. xn) –>set of input vectors and (w1, w2, w3……. wn) –>set of weights (Fig.1).

y=actual output

wo=initial weight

wnew=new weight

δw=change in weight

α=learning rate

actual output (y)=wi*xi

learning signal (ej)=ti-y difference between desired and actual output)

δw=α*xi*ej

wnew=wo+δw

Now, the output can be calculated on the basis of the input and the activation function applied over the net input and can be expressed as:

y=1, if net input>=θ

y=0, if net input<θ

Procedure:

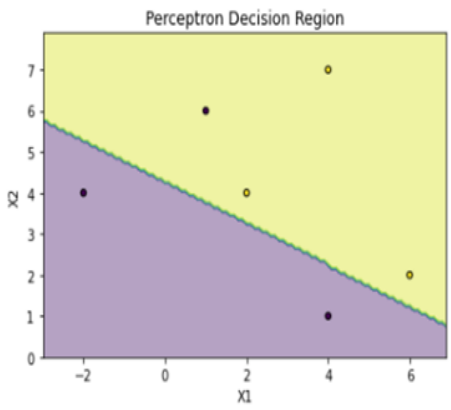
1.In this code, we first define our training data, which consists of six data points with two features each.
2.We label these points as either -1 or 1, depending on which class they belong to.
3.Next, we initialize the weights and bias of the perceptron to zero.
4.We then define the perceptron function, which takes the training data, weights, and bias as inputs, and trains the perceptron using the perceptron learning algorithm.
5.This algorithm updates the weights and bias for each data point, based on whether the perceptron's prediction is correct or not.
6.Finally, we call the perceptron function to train the perceptron on our training data.
7.We then plot the training data, along with the decision boundary learned by the perceptron.
8.The decision boundary is represented as a contour plot, which separates the two classes of data points.

Conclusion: Thus, we have studied to demonstrate the perceptron learning law with its decision region as the output in Graphical form using python.

Program code and Output

In [1]:

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

# Define the training data
X = np.array([    [-2, 4],
    [4, 1],
    [1, 6],
    [2, 4],
    [6, 2],
    [4, 7]
])
y = np.array([-1, -1, -1, 1, 1, 1])

# Initialize the weights and bias
w = np.zeros(X.shape[1])
b = 0

# Define the perceptron function
def perceptron(X, y, w, b, learning_rate=0.1, num_epochs=100):
    for epoch in range(num_epochs):
        for i in range(X.shape[0]):
            # Compute the output of the perceptron
            z = np.dot(X[i], w) + b
            if z > 0:
                y_pred = 1
            else:
                y_pred = -1
            # Update the weights and bias if the prediction is wrong
            if y_pred != y[i]:
                w = w + learning_rate * y[i] * X[i]
                b = b + learning_rate * y[i]
    return w, b

# Train the perceptron
w, b = perceptron(X, y, w, b)

# Plot the training data and the decision boundary
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1), np.arange(y_min, y_max, 0.1))
Z = np.sign(np.dot(np.c_[xx.ravel(), yy.ravel()], w) + b)
Z = Z.reshape(xx.shape)
plt.contourf(xx, yy, Z, alpha=0.4)
plt.scatter(X[:, 0], X[:, 1], c=y, s=20, edgecolor='k')
plt.title('Perceptron Decision Region')
plt.xlabel('X1')
plt.ylabel('X2')
plt.show()
```


Perceptron Decision Region
X1
X2
−2
0
2
4
6
0
1
2
3
4
5
6
7

Aim: Write a python program for Bidirectional Associative Memory with two pairs of vectors.

Software for Python: Jupyter Notebook

Theory:

Bidirectional Associative Memory (BAM) is a supervised learning model in Artificial Neural Network. This is hetero-associative memory, for an input pattern, it returns another pattern which is potentially of a different size. This phenomenon is very similar to the human brain. Human memory is necessarily associative. It uses a chain of mental associations to recover a lost memory like associations of faces with names, in exam questions with answers, etc.

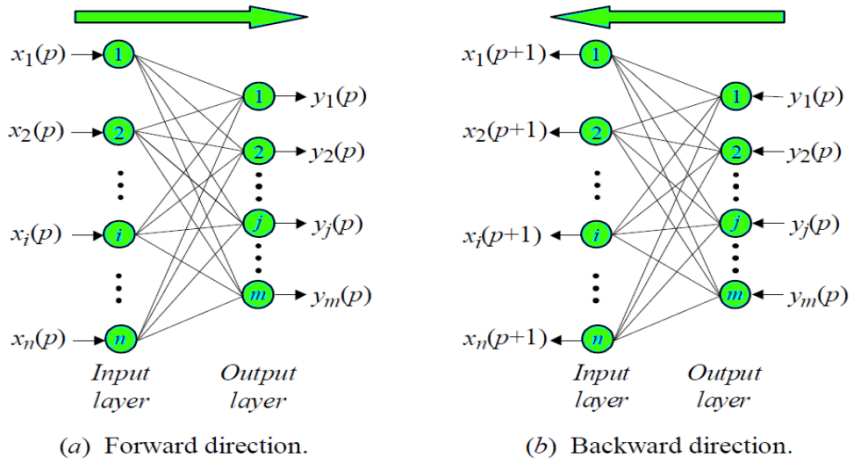
In such memory associations for one type of object with another, a Recurrent Neural Network (RNN) is needed to receive a pattern of one set of neurons as an input and generate a related, but different, output pattern of another set of neurons.

Why BAM is required?

The main objective to introduce such a network model is to store hetero-associative pattern pairs. This is used to retrieve a pattern given a noisy or incomplete pattern.

BAM Architecture:

When BAM accepts an input of n-dimensional vector X from set A then the model recalls m-dimensional vector Y from set B. Similarly, when Y is treated as input, the BAM recalls X.



(*a*) Forward direction. (*b*) Backward direction.

Fig.1.BAM Architecture

Algorithm:

1.Storage (Learning): In this learning step of BAM, weight matrix is calculated between M pairs of patterns (fundamental memories) are stored in the synaptic weights of the network following the equation.

$$W = \sum_{m=1}^{M} X_m Y_m^T$$

2.Testing: We have to check that the BAM recalls perfectly Ym for corresponding Xm and recalls Xm for corresponding Ym. Using,

$$Y_m = \text{sign}\left(W^T X_m\right), \quad m = 1.2, \ldots, M$$

$$X_m = \text{sign}\left(W Y_m\right), \quad m = 1.2, \ldots, M$$

All pairs should be recalled accordingly.

3.Retrieval: For an unknown vector X (a corrupted or incomplete version of a pattern from set A or B) to the BAM and retrieve a previously stored association:

$$X \neq X_m, \quad m = \mathbf{1}, \mathbf{2}, \ldots, M$$

Initialize the BAM:
$X(0) = X, \quad p = 0$
Calculate the BAM output at iteration p
$Y(p) = \text{sign}\left[W^T X(p)\right]$
Update the input vector X(p):
$X(p+1) = \text{sign}[WY(p)]$
Repeat the iteration until convergence, when input and output remain unchanged.

Procedure:
This code implements a Bidirectional Associative Memory (BAM) neural network.

First, it defines two input/output pairs, input1/output1 and input2/output2. It then defines a weight matrix W, which is the sum of the outer products of the two input/output pairs. The weight matrix W is used to associate the input vectors with the output vectors.

The code then defines two activation functions: f and g. The function f returns 1 if the input is greater than 0, and -1 otherwise. The function g returns 1 if the input is greater than or equal to 0, and -1 otherwise. These functions are used to threshold the output of the network.

The BAM function takes an input vector as input and uses the weight matrix W to compute an output vector. The output vector is then thresholded using the g function to produce a new input vector. This new input vector is then thresholded using the f function to produce a new output vector. This process is repeated until the input/output pairs converge to a stable state.

Finally, the code tests the BAM function by applying it to an input vector and printing the input vector, the output vector, and the new input vector produced by the BAM function.

Conclusion: Thus, we have studied Bidirectional Associative Memory (BAM) with two pairs of vectors using Python Programming.

Program Code and Output

In [1]:

```
import numpy as np

# Define input/output vectors
input1 = np.array([1, 0, 1, 0])
output1 = np.array([0, 1])

input2 = np.array([1, 1, 0, 0])
output2 = np.array([1, 0])

# Define weight matrix
W = np.outer(output1, input1) + np.outer(output2, input2)

# Define activation functions
def f(x):
    return 1 if x > 0 else -1

def g(x):
    return 1 if x >= 0 else -1

# Define bidirectional associative memory function
def BAM(input_vec):
    y = np.dot(W, input_vec)
    output_vec = np.array([g(x) for x in y])
    x = np.dot(W.T, output_vec)
    input_vec = np.array([f(x) for x in x])
    return input_vec, output_vec

# Test the BAM function
input_vec = np.array([1, 0, 0, 1])
input_vec_new, output_vec = BAM(input_vec)

print("Input vector: ", input_vec)
print("Output vector: ", output_vec)
print("New input vector: ", input_vec_new)
```

```
Input vector:  [1 0 0 1]
Output vector:  [1 1]
New input vector:  [ 1  1  1 -1]
```

Aim: Implement Artificial Neural Network training process in Python using Forward Propagation and Back Propagation.

Software for Python: Jupyter Notebook

Theory:

Forward Propagation

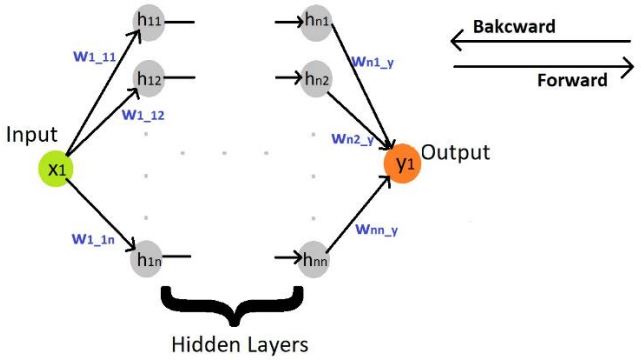


Fig.1-Forward feed in an example neural network structure

In the Neural Network, our journey starting from the input to the output is called the forward direction. The weights entering each node are multiplied with the available value (if input is x feature value, if hidden layer is the value from the sum of previous multiplications entering that node) and bias is added. This multiplication operation is called the "dot product", and that's because we're dealing with vectors.

$$z = W^1.x + b_1$$

$$h = \emptyset(z)$$

$$o = W^2.h + b_2$$

Now we can move on to our numerical example. Let's go through a simple example and focus on the operations rather than the complexity of the model. There are two input neurons (node), two hidden neurons, and two output neurons. In addition to these, the hidden layer and the output layer contain two biases. Let's use the "sigmoid" activation function in the hidden layer.

i is abbreviated to represent the input layer
h is abbreviated to represent the hidden layer
o is abbreviated to represent the output layer.

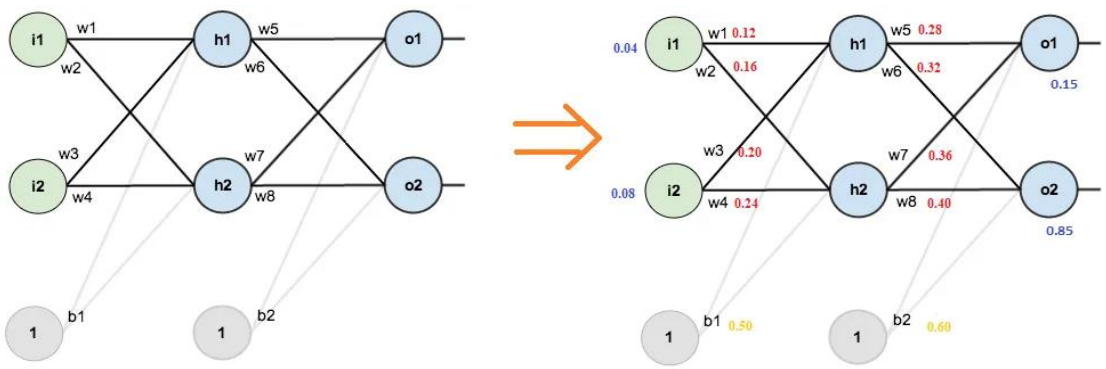


Fig.2-Example with parameter values

Let's calculate h1 first. While doing this, it is seen that there are i1, i2 and b1 nodes entering the h1 node. Therefore:

$$h_1 = w_1 * i_1 + w_3 * i_2 + b_1 * 1 = 0.12 * 0.04 + 0.2 * 0.08 + 0.50 * 1 = 0.5208$$

$$out_{h_1} = \frac{1}{1 + e^{-h_1}} = \frac{1}{1 + e^{-0.5208}} = 0.6273$$

Then let's find h2:

$$h_2 = (w_2 * i_1) + (w_4 * i_2) + (b_1 * 1) = 0.16 * 0.05 + 0.24 * 0.1 + 0.5 * 1 = 0.5256$$

$$out_{h_2} = \frac{1}{1+e^{-h_2}} = \frac{1}{1+e^{-0.584}} = 0.6284$$

Now we have calculated our hidden layers. The queue is in the output layer. Let's start with o1 first:

$$o_1 = (w_5 * out_{h_1}) + (w_7 * out_{h_2}) + (b_2 * 1) =$$

$$(0.28 * 0.6273) + (0.36 * 0.6304) + 0.6 * 1 = 0.9350$$

$$out_{o_1} = \frac{1}{1+e^{-o_1}} = \frac{1}{1+e^{-1}} = 0.7181$$

Finally, let's calculate o2:

$$o_2 = w_6 * out_{h_1} + w_8 * out_{h_2} + b_2 * 1 =$$

$$(0.32 * 0.6273) + (0.40 * 0.6284) + 0.6 * 1 = 0.9768$$

$$out_{o_2} = \frac{1}{1+e^{-o_2}} = \frac{1}{1+e^{-1.1084}} = 0.7265$$

Since we have calculated our outputs, it is time to find the loss function value. Let's say our target values are both 0.15:

$$E_{o_1} = \frac{1}{2}(\hat{y} - out_{o_1})^2 = \frac{1}{2}(0.15 - 0.7181)^2 = 0.1614$$

$$E_{o_1} = \frac{1}{2}(\hat{y} - out_{o_2})^2 = \frac{1}{2}(0.85 - 0.7265)^2 = 0.0076$$

$$E_{total} = E_{o_1} + E_{o_2} = 0.1688 + 0.0076 = 0.169$$

E_{total} is the total loss of forward feed.

Backward Propagation

This algorithm is called backpropagation because it tries to reduce errors from output to input. It looks for the minimum value of the error function in the weight field using a technique called gradient descent.

With this method, we will try to reduce the error by changing the weight and bias values. Since we will do a single step and show the weight update, let's take the learning rate as a big value like 0.5 to make a little bigger progress and start back propagation. Let's choose w5 as the node whose weight we want to update and let's do the operations on it. (All weights are updated with this method, but I chose w5 for illustration)

Backpropagation is the core of neural network training. It is a method of adjusting the weights of a neural network based on the loss value obtained in the previous epoch. Correctly adjusting the weights allows us to reduce the error rate and increase its generalization, making the model reliable.

Instead of going directly to the formula, let's look at the above network again. This will guide us in a way to implement Chain Rule. The value taken from Loss is given back as input and affects o1's sigmoid state, which in turn affects o1 before sigmoid and o1, w5 as the last link of the chain:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial o_1} * \frac{\partial o_1}{\partial w_5}$$

Let's consider all three derivative operations above one by one. The formulas of these variables are the same as in forward propagation, but it will be enough to take the derivative and substitute it:

$$E_{total} = \frac{1}{2}.(y_{o1} - out_{o1})^2 + \frac{1}{2}.(y_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2.\left(\frac{1}{2}\right).(y_{o1} - out_{o1})^{2-1}.(-1) + 0 = -(y_{o1} - out_{o1}) = -(0.15 - 0.7181) = 0.5681$$

Since we took the derivative with respect to o1, the part with o2 behaved like a constant and was equal to 0.

$$o_1 = w5 * out_{h1} + w6 * out_{h2} + b_2$$

$$\frac{\partial o_1}{\partial w5} = 1 * out_{h1} * w5^{(1-1)} + 0 + 0 = out_{h1} = 0.6273$$

→ Again, the parts with o2 were equal to zero.

Now that we have the derivatives, we can apply the chain rule by multiplying them:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial o_1} * \frac{\partial o_1}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_5} = -(y_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * out_{h1} = 0.5681 * 0.2024 * 0.6273 = 0.0721$$

As a final step, we can now update the w5 weight. While doing this, we multiply the derivative we found from the previous weight with a fixed learning rate that we determine:

$$w_5^+ = w_5 - \eta.\left(\frac{\partial E_{total}}{\partial w5}\right) = 0.28 - 0.5.(0.0721) = 0.24395$$

Updated w5 weight

Procedure:

This code defines a simple neural network using the sigmoid activation function and backpropagation for training.

Here's how it works:

First, the code defines the sigmoid activation function and its derivative using the numpy library. The sigmoid function maps any input to a value between 0 and 1, which is useful for creating a binary output for classification problems.

Then, the code defines a NeuralNetwork class. The initialization method takes the input and output data as arguments, and initializes the weights randomly. The feedforward method computes the output of the network given the input and current weights, while the backprop method updates the weights based on the difference between the predicted output and the actual output.

Next, the code defines the training data as X (input) and y (output). X is a 4x3 matrix representing four training examples, each with three input features. y is a 4x1 matrix representing the target output for each training example.

Finally, the code trains the neural network using a for loop that calls the feedforward and backprop methods repeatedly for a fixed number of iterations (in this case, 1500). After training, the code prints the final output of the network.

Note that this code implements a very simple neural network with only one hidden layer and four hidden units. For more complex problems, you may need to adjust the number of layers and units, as well as the activation function and optimization algorithm.

Conclusion: Thus, we have studied how to implement Artificial Neural Network training process in Python using Forward Propagation and Back Propagation.

Program Code and Output

In [6]:
```
import numpy as np

# Define the sigmoid function for activation
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Define the derivative of sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)

# Define the neural network class
class NeuralNetwork:

    def __init__(self, x, y):
        self.input = x
        self.weights1 = np.random.rand(self.input.shape[1],4)
        self.weights2 = np.random.rand(4,1)
        self.y = y
        self.output = np.zeros(self.y.shape)

    def feedforward(self):
        self.layer1 = sigmoid(np.dot(self.input, self.weights1))
        self.output = sigmoid(np.dot(self.layer1, self.weights2))

    def backprop(self):
        d_weights2 = np.dot(self.layer1.T, (2*(self.y - self.output) * sigmoid_derivative(self.output)))
        d_weights1 = np.dot(self.input.T,  (np.dot(2*(self.y - self.output) * sigmoid_derivative(self.output),
                                                   self.weights2.T) * sigmoid_derivative(self.layer1)))
        self.weights1 += d_weights1
        self.weights2 += d_weights2

# Define the training data
X = np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]])
y = np.array([[0],[1],[1],[0]])

# Initialize the neural network
nn = NeuralNetwork(X,y)

# Train the neural network
for i in range(1500):
    nn.feedforward()
    nn.backprop()

# Print the output after training
print(nn.output)
```

```
print(nn.output)
```

```
[[0.01030782]
 [0.96569   ]
 [0.96580405]
 [0.04033994]]
```

In [ ]:

Aim: Write a Python program to illustrate ART Neural Network.

Software requirement for Python: Jupyter Notebook

Theory:

Adaptive Resonance Theory ART networks, as the name suggests, is always open to new learning adaptive without losing the old patterns resonance. Basically, ART network is a vector classifier which accepts an input vector and classifies it into one of the categories depending upon which of the stored pattern it resembles the most.

The main operation of ART classification can be divided into the following phases −

Recognition phase − The input vector is compared with the classification presented at every node in the output layer. The output of the neuron becomes “1” if it best matches with the classification applied, otherwise it becomes “0”.

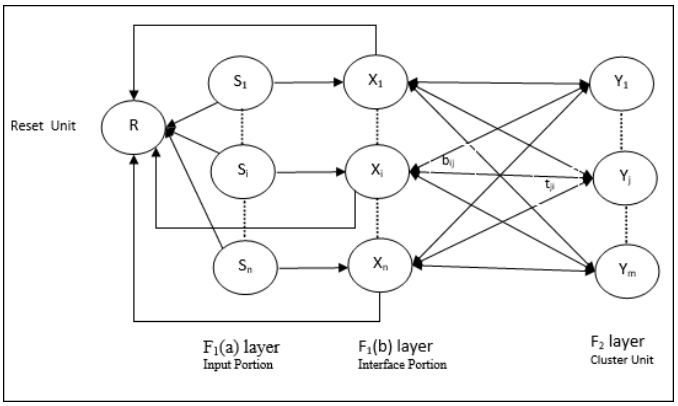
Comparison phase − In this phase, a comparison of the input vector to the comparison layer vector is done. The condition for reset is that the degree of similarity would be less than vigilance parameter.

Search phase − In this phase, the network will search for reset as well as the match done in the above phases. Hence, if there would be no reset and the match is quite good, then the classification is over. Otherwise, the process would be repeated and the other stored pattern must be sent to find the correct match.

ART1

It is a type of ART, which is designed to cluster binary vectors. We can understand about this with the architecture of it.

Architecture of ART1



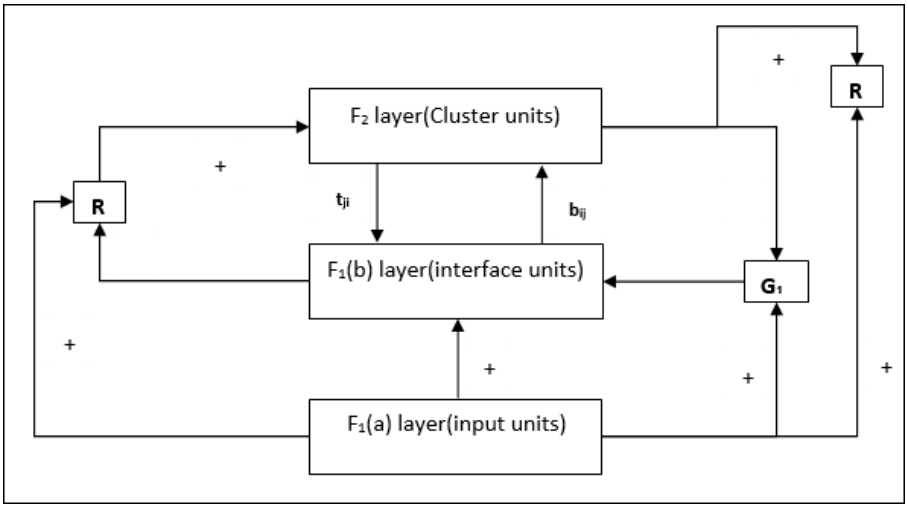


Fig.1-ART1 ARCHITECTURE

The ART1 Architecture (Fig.1) consists of the following two units –

Computational Unit − It is made up of the following −

Input unit (F1 layer) − It further has the following two portions −

F1a layer Input portion

− In ART1, there would be no processing in this portion rather than having the input vectors only. It is connected to F1b layer interface portion.

F1b layer Interface portion

− This portion combines the signal from the input portion with that of F2 layer. F1b layer is connected to F2 layer through bottom-up weights bij and F2 layer is connected to F1b layer through top-down weights tji.

Cluster Unit (F2 layer) – This is a competitive layer. The unit having the largest net input is selected to learn the input pattern. The activation of all other cluster unit is set to 0.

Reset Mechanism – The work of this mechanism is based upon the similarity between the top-down weight and the input vector. Now, if the degree of this similarity is less than the vigilance parameter, then the cluster is not allowed to learn the pattern and a rest would happen.

Supplement Unit – Actually the issue with Reset mechanism is that the layer F2 must have to be inhibited under certain conditions and must also be available when some learning happens. That is why two supplemental units namely, G1 and G2 is added along with reset unit, R. They are called gain control units. These units receive and send signals to the other units present in the network. '+' indicates an excitatory signal, while '–' indicates an inhibitory signal.

Procedure:

This code defines a class called ARTNetwork which is an implementation of an Adaptive Resonance Theory neural network. The network is trained on a set of input vectors represented by a numpy array X and can make predictions on new input vectors.

The network has a fixed number of categories specified by num_categories and a vigilance parameter specified by vigilance. The categories list holds the category vectors learned by the network during training.

During training, the network tries to fit each input vector to an existing category vector or create a new category vector if no match is found. This is done by first computing the similarity between the input vector and each existing category vector. If the similarity is above the vigilance threshold, the input vector is added to the category after updating it. If not, the input vector is checked for complementarity with the existing category vectors. If a complementary category is found, a new category vector is created by taking the maximum element-wise values between the input vector and the complementary category vector.

The predict method of the network returns the index of the matching category vector for a given input vector or None if no match is found.

The code tests the network on three input vectors and prints the predicted category index for each.

Conclusion: Thus, we have studied ART Neural Network with Python Programming.

Program Code and Output

```
import numpy as np
class ARTNetwork:
    def __init__(self, num_categories, vigilance):
        self.num_categories = num_categories
        self.vigilance = vigilance
        self.categories = []

    def _initialize_categories(self, input_vec):
        self.categories.append(input_vec)

    def _compute_similarity(self, input_vec, category):
        return np.dot(input_vec, category) / np.sum(input_vec)

    def _get_matching_category(self, input_vec):
        for category in self.categories:
            if self._compute_similarity(input_vec, category) >= self.vigilance:
                return category
        return None

    def _get_complementary_category(self, input_vec):
        for category in self.categories:
            if np.all(category + input_vec <= 1):
                return category
        return None

    def _update_category(self, input_vec, category):
        return np.maximum(input_vec, category)
```

```
    def _create_new_category(self, input_vec, complementary_category):
        return np.maximum(input_vec, complementary_category)

    def learn(self, input_vec):
        if not self.categories:
            self._initialize_categories(input_vec)
        else:
            matching_category = self._get_matching_category(input_vec)
            if matching_category is not None:
                self.categories.remove(matching_category)
                self.categories.append(self._update_category(input_vec, matching_category))
            else:
                complementary_category = self._get_complementary_category(input_vec)
                if complementary_category is not None:
                    self.categories.append(self._create_new_category(input_vec, complementary_category))
                else:
                    self.categories.append(input_vec)

    def predict(self, input_vec):
        matching_category = self._get_matching_category(input_vec)
        if matching_category is not None:
            return self.categories.index(matching_category)
        else:
            return None

# Define the input data
X = np.array([[1, 0, 0, 1],
              [1, 1, 0, 0],
```

```
            [0, 0, 1, 1]])

# Initialize the ART network
art = ARTNetwork(num_categories=2, vigilance=0.5)

# Train the ART network
for input_vec in X:
    art.learn(input_vec)

# Test the ART network
print("Prediction for [1, 1, 1, 0]:", art.predict(np.array([1, 1, 1, 0])))
print("Prediction for [0, 1, 1, 0]:", art.predict(np.array([0, 1, 1, 0])))
print("Prediction for [0, 0, 1, 0]:", art.predict(np.array([0, 0, 1, 0])))
```

OUTPUT OF THE CODE

Prediction for [1, 1, 1, 0]: 0

Prediction for [0, 1, 1, 0]: 0

Prediction for [0, 0, 1, 0]: 0

The output of 0 for all three predictions indicates that the ART network has categorized each input vector into the first category it created during initialization. This may be due to the chosen vigilance value of 0.5, which is relatively high and may lead to a lower number of categories being formed. Additionally, the input vectors in the X array are quite similar, which may have contributed to the ART network grouping them together into a single category. To improve the accuracy of the predictions, the vigilance parameter and input vectors could be adjusted to create more distinct categories.

Aim: Write a Python program to design a Hopfield network which stores 4 vectors.

Software requirement for Python: Jupyter Notebook

Theory:

Hopfield neural network was invented by Dr. John J. Hopfield in 1982. It consists of a single layer which contains one or more fully connected recurrent neurons. The Hopfield network is commonly used for auto-association and optimization tasks.

Discrete Hopfield Network

A Hopfield network which operates in a discrete line fashion or in other words, it can be said the input and output patterns are discrete vector, which can be either binary 0,1 or bipolar +1,−1 in nature. The network has symmetrical weights with no self-connections i.e., wij = wji and wii = 0 (Fig.1).

Following are some important points to keep in mind about discrete Hopfield network –

1. This model consists of neurons with one inverting and one non-inverting output.
2. The output of each neuron should be the input of other neurons but not the input of self.
3. Weight/connection strength is represented by wij.
4. Connections can be excitatory as well as inhibitory. It would be excitatory, if the output of the neuron is same as the input, otherwise inhibitory.
5. Weights should be symmetrical, i.e. wij = wji

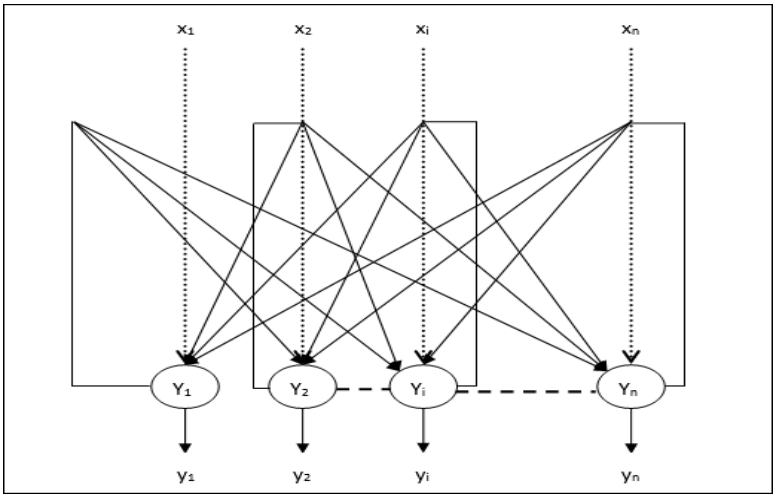


Fig.1- Discrete Hopfield Network

The output from Y1 going to Y2, Yi and Yn have the weights w12, w1i and w1n respectively. Similarly, other arcs have the weights on them.

Training Algorithm

During training of discrete Hopfield network, weights will be updated. As we know that we can have the binary input vectors as well as bipolar input vectors.

Hence, in both the cases, weight updates can be done with the following relation

Case 1 – Binary input patterns

For a set of binary patterns sp, p = 1 to P

Here, sp = s1p, s2p,..., sip,..., snp

Weight Matrix is given by

$$w_{ij} = \sum_{p=1}^{P}[2s_i(p) - 1][2s_j(p) - 1] \quad for\ i \neq j$$

Case 2 – Bipolar input patterns

For a set of binary patterns sp, p = 1 to P

Here, sp = s1p, s2p,..., sip,..., snp

Weight Matrix is given by

$$w_{ij} = \sum_{p=1}^{P}[s_i(p)][s_j(p)] \quad for\ i \neq j$$

Testing Algorithm

Step 1 – Initialize the weights, which are obtained from training algorithm by using Hebbian principle.

Step 2 – Perform steps 3-9, if the activations of the network is not consolidated.

Step 3 – For each input vector X, perform steps 4-8.

Step 4 – Make initial activation of the network equal to the external input vector X as follows –

$$y_i = x_i \quad for\, i = 1\, to\, n$$

Step 5 – For each unit $\mathbf{Y_i}$, perform steps 6-9.

Step 6 – Calculate the net input of the network as follows –

$$y_{ini} = x_i + \sum_j y_j w_{ji}$$

Step 7 – Apply the activation as follows over the net input to calculate the output –

$$y_i = \begin{cases} 1 & if\, y_{ini} > \theta_i \\ y_i & if\, y_{ini} = \theta_i \\ 0 & if\, y_{ini} < \theta_i \end{cases}$$

Here θ_i is the threshold.

Step 8 − Broadcast this output yi to all other units.

Step 9 − Test the network for conjunction.

Procedure:

This code implements a Hopfield network, a type of recurrent neural network used for content-addressable memory and pattern recognition tasks.

The HopfieldNetwork class is defined with an init method that initializes the network with the number of neurons (n) and weights initialized as an n x n matrix of zeros. The train method takes in a set of patterns and updates the weights matrix based on the outer product of each pattern with itself. The diagonal elements of the weights matrix are set to zero to prevent self-connections.

The recall method takes in a pattern and uses it to update the activation of the network until it converges to a stable state. The energy of the system is calculated at each iteration, and the activation is updated using the sign function applied to the dot product of the weights matrix and the current activation. If the new activation is the same as the previous one, the iteration stops and the output is returned. If the maximum number of iterations is reached before convergence, the current activation is returned as the output.

The main code initializes the Hopfield network with 4 neurons, trains it on a set of 4 input patterns, and then recalls each pattern to check if it is correctly recognized by the network.

Conclusion: Thus, we have studied to design a Hopfield network which stores 4 vectors.

Program Code and Output

```
import numpy as np
class HopfieldNetwork:
    def __init__(self, n):
        self.n = n
        self.weights = np.zeros((n, n))

    def train(self, patterns):
        self.weights = np.zeros((self.n, self.n))
        for p in patterns:
            self.weights += np.outer(p, p)
        np.fill_diagonal(self.weights, 0)

    def recall(self, pattern, max_iters=100):
        for i in range(max_iters):
            energy = -0.5 * np.dot(np.dot(pattern, self.weights), pattern)
            new_pattern = np.sign(np.dot(self.weights, pattern))
            if np.array_equal(new_pattern, pattern):
                return new_pattern
            pattern = new_pattern
        return pattern

# Define the input patterns
patterns = np.array([[1, 0, 1, 0], [0, 1, 0, 1], [1, 1, 0, 0], [0, 0, 1, 1]])
# Initialize the Hopfield network
hopfield = HopfieldNetwork(n=4)
```

```
# Train the network
hopfield.train(patterns)
# Recall the patterns
for pattern in patterns:
    recalled_pattern = hopfield.recall(pattern)
    print("Input pattern:", pattern)
    print("Recalled pattern:", recalled_pattern)
```

OUTPUT OF THE CODE

Input pattern: [1 0 1 0]

Recalled pattern: [1. 1. 1. 1.]

Input pattern: [0 1 0 1]

Recalled pattern: [1. 1. 1. 1.]

Input pattern: [1 1 0 0]

Recalled pattern: [1. 1. 1. 1.]

Input pattern: [0 0 1 1]

Recalled pattern: [1. 1. 1. 1.]

Aim: Write a Python program to implement CNN object detection. Discuss numerous performance evaluation metrics for evaluating the object detecting algorithm performance.

Software requirement for Python: Jupyter Notebook

Theory:

Object detection is a computer vision technique whose aim is to detect objects such as cars, buildings, and human beings, just to mention a few. The objects can generally be identified from either pictures or video feeds.

Object detection has been applied widely in video surveillance, self-driving cars, and object/people tracking. In this piece, we'll look at the basics of object detection and review some of the most commonly-used algorithms and a few brand new approaches, as well.

How Object Detection Works

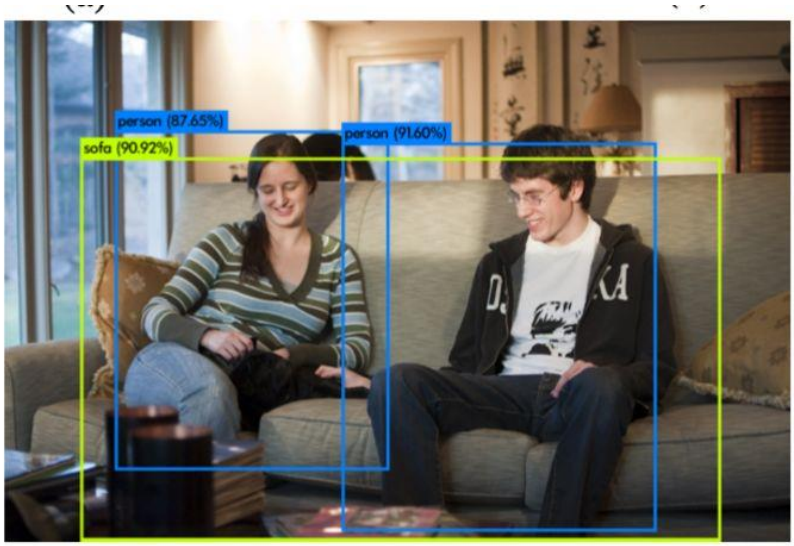
Object detection locates the presence of an object in an image and draws a bounding box around that object. This usually involves two processes; classifying and object's type, and then drawing a box around that object. Some of the common model architectures used for object detection:

1. R-CNN
2. Fast R-CNN
3. Faster R-CNN
4. Mask R-CNN
5. SSD (Single Shot MultiBox Defender)
6. YOLO (You Only Look Once)
7. Objects as Points
8. Data Augmentation Strategies for Object Detection

Once you have trained your first object detector, the next step is to know its performance. Sure enough, you can see the model finds all the objects in the pictures you feed it. Great! But how do you quantify that? How should we decide which model is better?

Since the classification task only evaluates the probability of the class object appearing in the image, it is a straightforward task for a classifier to identify correct predictions from incorrect ones. However, the object detection task localizes the object further with a bounding box associated with its corresponding confidence score to report how certain the bounding box of the object class is detected.

A detector outcome is commonly composed of a list of bounding boxes, confidence levels and classes, as seen in the following Figure:



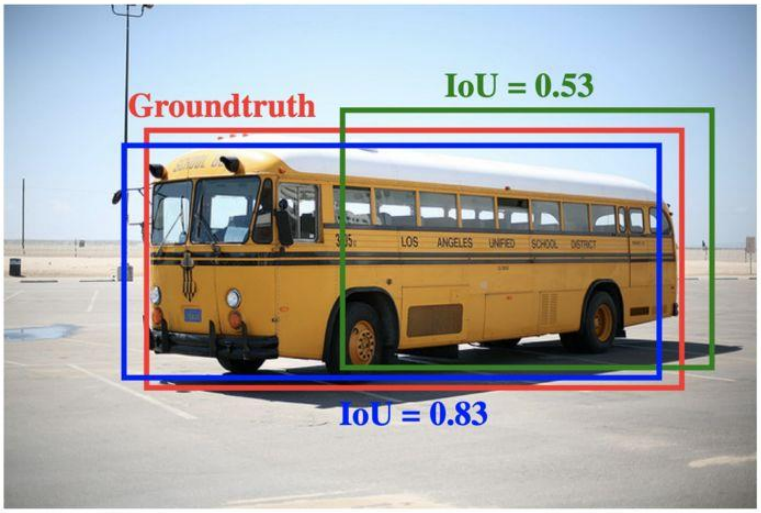
Object detection metrics serve as a measure to assess how well the model performs on an object detection task. It also enables us to compare multiple detection systems objectively or compare them to a benchmark. In most competitions, the average precision (AP) and its derivations are the metrics adopted to assess the detections and thus rank the teams.

Understanding the various metric:

IoU: Guiding principle in all state-of-the-art metrics is the so-called Intersection-over-Union (IoU) overlap measure. It is quite literally defined as the intersection over union of the detection bounding box and the ground truth bounding box.

Dividing the area of overlap between predicted bounding box and ground truth by the area of their union yields the **Intersection over Union (IoU).**

An Intersection over Union score > 0.5 is normally considered a “good” prediction.



IoU metric determines how many objects were detected correctly and how many false positives were generated (will be discussed below).

True Positives [TP]

Number of detections with IoU>0.5

False Positives [FP]

Number of detections with IoU<=0.5 or detected more than once

False Negatives [FN]

Number of objects that not detected or detected with IoU<=0.5

Precision

Precision measures how accurate your predictions are. i.e. the percentage of your predictions that are correct.

Precision = True positive / (True positive + False positive)

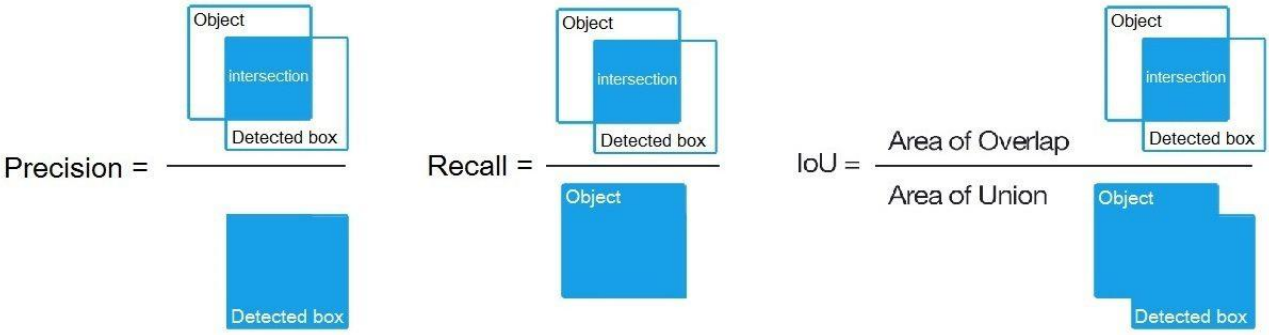
Recall

Recall measures how good you find all the positives.

Recall = True positive / (True positive + False negative)

F1 Score

F1 score is HM (Harmonic Mean) of precision and recall.

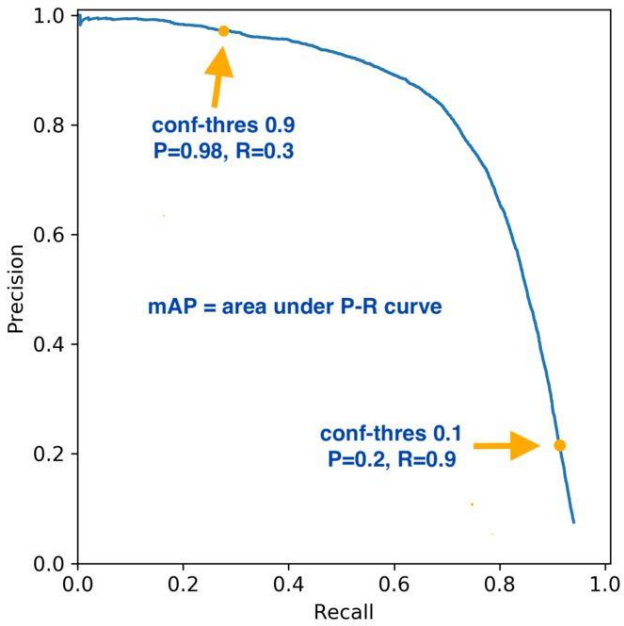


Average Precision (AP)

The general definition for the Average Precision (AP) is finding the area under the precision-recall curve.

Map

The mAP for object detection is the average of the AP calculated for all the classes. mAP@0.5 means that it is the mAP calculated at IOU threshold 0.5.



Procedure:

This code defines and trains a Convolutional Neural Network (CNN) model using the CIFAR-10 dataset. The CNN architecture consists of several convolutional layers, followed by max-pooling and dense layers. The model is trained to classify images as either containing an object of interest (in this case, a specific type of object labeled as 2) or not.

The code first imports necessary libraries such as NumPy, TensorFlow, and its Keras API for defining the CNN model architecture. Then, it defines the CNN model using the Keras functional API, which includes convolutional, max-pooling, and dense layers. The dataset (CIFAR-10) is loaded and preprocessed by normalizing the pixel values and transforming the labels to binary values (1 for the object of interest and 0 otherwise).

The model is then compiled with Adam optimizer, binary cross-entropy loss function, and several metrics such as accuracy, precision, and recall. Finally, the model is trained for a fixed number of epochs using the training data and evaluated using the test data. The results are then printed, which include the test loss, accuracy, precision, and recall of the model.

Conclusion: Thus, we have studied to implement CNN object detection.

Program Code and Output

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import Model
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dense, Flatten, Input

# Define the CNN model
def create_model():
    input_tensor = Input(shape=(32, 32, 3))
    x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_tensor)
    x = MaxPooling2D((2, 2))(x)
    x = Conv2D(64, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2))(x)
    x = Conv2D(128, (3, 3), activation='relu', padding='same')(x)
    x = MaxPooling2D((2, 2))(x)
    x = Flatten()(x)
    x = Dense(256, activation='relu')(x)
    x = Dense(1, activation='sigmoid')(x)
    model = Model(inputs=input_tensor, outputs=x)
    return model

# Load the dataset
(X_train, y_train), (X_test, y_test) = tf.keras.datasets.cifar10.load_data()

# Normalize the images
X_train = X_train / 255.0
X_test = X_test / 255.0
```

```
# Define the object detection labels
y_train = (y_train == 2).astype(int)
y_test = (y_test == 2).astype(int)

# Create the CNN model
model = create_model()

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy',
tf.keras.metrics.Precision(), tf.keras.metrics.Recall()])

# Train the model
model.fit(X_train, y_train, epochs=10, batch_size=64, validation_data=(X_test, y_test))

# Evaluate the model
results = model.evaluate(X_test, y_test, batch_size=64)
print("Test loss, Test accuracy, Test precision, Test recall:", results)
```

OUTPUT OF THE CODE

Epoch 1/10

782/782 [==============================] - 43s 53ms/step - loss: 0.2696 - accuracy: 0.9019 - precision: 0.6091 - recall: 0.0536 - val_loss: 0.2391 - val_accuracy: 0.9099 - val_precision: 0.6334 - val_recall: 0.2350

Epoch 2/10

782/782 [==============================] - 45s 58ms/step - loss: 0.2278 - accuracy: 0.9136 - precision: 0.6854 - recall: 0.2506 - val_loss: 0.2237 - val_accuracy: 0.9151 - val_precision: 0.6982 - val_recall: 0.2660

Epoch 3/10

782/782 [==============================] - 52s 67ms/step - loss: 0.2027 - accuracy: 0.9244 - precision: 0.7276 - recall: 0.3894 - val_loss: 0.2016 - val_accuracy: 0.9233 - val_precision: 0.7271 - val_recall: 0.3730

Epoch 4/10

782/782 [==============================] - 70s 89ms/step - loss: 0.1809 - accuracy: 0.9326 - precision: 0.7587 - recall: 0.4774 - val_loss: 0.2005 - val_accuracy: 0.9262 - val_precision: 0.7937 - val_recall: 0.3540

Epoch 5/10

782/782 [==============================] - 74s 95ms/step - loss: 0.1639 - accuracy: 0.9380 - precision: 0.7781 - recall: 0.5322 - val_loss: 0.1867 - val_accuracy: 0.9308 - val_precision: 0.7541 - val_recall: 0.4570

Epoch 6/10

782/782 [==============================] - 65s 83ms/step - loss: 0.1448 - accuracy: 0.9439 - precision: 0.7917 - recall: 0.5952 - val_loss: 0.1924 - val_accuracy: 0.9286 - val_precision: 0.7449 - val_recall: 0.4350

Epoch 7/10

782/782 [==============================] - 67s 86ms/step - loss: 0.1265 - accuracy: 0.9518 - precision: 0.8201 - recall: 0.6636 - val_loss: 0.2067 - val_accuracy: 0.9225 - val_precision: 0.6144 - val_recall: 0.6040

Epoch 8/10

782/782 [==============================] - 67s 85ms/step - loss: 0.1073 - accuracy: 0.9590 - precision: 0.8454 - recall: 0.7220 - val_loss: 0.2137 - val_accuracy: 0.9204 - val_precision: 0.5951 - val_recall: 0.6380

Epoch 9/10

782/782 [==============================] - 67s 85ms/step - loss: 0.0900 - accuracy: 0.9657 - precision: 0.8710 - recall: 0.7708 - val_loss: 0.2167 - val_accuracy: 0.9264 - val_precision: 0.6507 - val_recall: 0.5700

Epoch 10/10

782/782 [==============================] - 65s 83ms/step - loss: 0.0746 - accuracy: 0.9715 - precision: 0.8864 - recall: 0.8196 - val_loss: 0.2465 - val_accuracy: 0.9316 - val_precision: 0.7416 - val_recall: 0.4850

157/157 [==============================] - 4s 24ms/step - loss: 0.2465 - accuracy: 0.9316 - precision: 0.7416 - recall: 0.4850

Test loss, Test accuracy, Test precision, Test recall: [0.2465059608221054, 0.9315999746322632, 0.7415902018547058, 0.48500001430511475]

Aim: Write a Python program to show Back Propagation Network for XOR function with binary input and output

Software requirement for Python: Jupyter Notebook

Theory:

The XOR function is a binary function that takes two binary inputs and returns a binary output based on the logical operation of "exclusive OR." The truth table for the XOR function (Fig.1.) is as follows:

X_1	X_2	$Y = X_1$ XOR X_2
0	0	0
0	1	1
1	0	1
1	1	0

Fig.1. The truth table for the XOR function

To create a backpropagation neural network for XOR function, we first need to define the architecture of the network. We can use a feedforward neural network with one hidden layer. The input layer will have two neurons, representing the two binary inputs. The hidden layer will have two or more neurons with a sigmoid activation function. Finally, the output layer will have one neuron with a sigmoid activation function to represent the binary output.

To train the network, we need a set of training data consisting of the binary inputs and outputs for the XOR function. We can use stochastic gradient descent with backpropagation to train the network. The backpropagation algorithm involves the following steps:

1. Feed the input values forward through the network and calculate the output.

2. Calculate the error between the output and the target output.

3. Calculate the derivative of the error with respect to the output neuron.

4. Calculate the derivative of the output neuron with respect to the hidden layer neurons.

5. Calculate the derivative of the hidden layer neurons with respect to the input layer neurons.

6. Use the chain rule to multiply the derivatives together to obtain the partial derivatives of the error with respect to the weights in the network.

7. Update the weights in the network using the partial derivatives and the learning rate.

8. Repeat steps 1-7 for each training example in the dataset, adjusting the weights after each example.

9. Repeat steps 1-8 for a fixed number of epochs or until the error on the training data is minimized.

By adjusting the weights in the network using backpropagation, the network will learn to approximate the XOR function for binary inputs and outputs.

Procedure:

This code implements a backpropagation neural network for solving the XOR function with binary input and output.

First, the sigmoid activation function and its derivative are defined. Then, the XOR function dataset is created and the weights and biases are initialized with random values.

The network is trained using a loop that performs forward propagation, backpropagation, and weight and bias updates for a specified number of iterations. The learning rate is also set in this loop.

Finally, the trained network is tested by applying the sigmoid function to the dot product of the input with the weights and adding the bias for each layer. The predicted output is printed by rounding the test output to the nearest integer.

Conclusion: Thus, we have studied Back Propagation Neural Network for XOR function with binary input and output.

Program Code and Output

```
import numpy as np
# Define the sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
# Define the derivative of the sigmoid function
def sigmoid_derivative(x):
    return x * (1 - x)
# Define the XOR function dataset
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])
# Set the random seed for reproducibility
np.random.seed(42)
# Initialize the weights and biases with random values
weights1 = np.random.rand(2, 2)
bias1 = np.random.rand(1, 2)
weights2 = np.random.rand(2, 1)
bias2 = np.random.rand(1, 1)
# Set the learning rate and number of iterations
learning_rate = 0.1
num_iterations = 10000
# Training loop
for iteration in range(num_iterations):
    # Forward propagation
    layer1_output = sigmoid(np.dot(X, weights1) + bias1)
    layer2_output = sigmoid(np.dot(layer1_output, weights2) + bias2)
```

```
  # Backpropagation
    layer2_error = y - layer2_output
    layer2_delta = layer2_error * sigmoid_derivative(layer2_output)
    layer1_error = np.dot(layer2_delta, weights2.T)
    layer1_delta = layer1_error * sigmoid_derivative(layer1_output)

    # Update weights and biases
    weights2 += np.dot(layer1_output.T, layer2_delta) * learning_rate
    bias2 += np.sum(layer2_delta, axis=0, keepdims=True) * learning_rate
    weights1 += np.dot(X.T, layer1_delta) * learning_rate
    bias1 += np.sum(layer1_delta, axis=0, keepdims=True) * learning_rate

# Test the trained network
test_output = sigmoid(np.dot(sigmoid(np.dot(X, weights1) + bias1), weights2) + bias2)
print("Predicted Output:")
print(test_output.round())
```

OUTPUT OF THE CODE

Predicted Output:

```
[[0.]
 [1.]
 [1.]
 [0.]]
```

Aim: PyTorch implementation of CNN

Software requirement for Python: Jupyter Notebook

Theory:

To implement a Convolutional Neural Network (CNN) using PyTorch, you can follow these general steps:

1. Import the necessary modules and packages, including torch, torchvision.datasets, torchvision.transforms, torch.nn, and torch.optim.
2. Define the hyperparameters for your model, including the number of epochs, batch size, and learning rate.
3. Define the architecture of your CNN using PyTorch's nn.Module class. This involves defining the various layers of your CNN, including convolutional layers, pooling layers, and fully connected layers.
4. Load your dataset and preprocess the data using the transforms module from torchvision. This includes transformations such as converting images to tensors and normalizing pixel values.
5. Create data loaders for your training and test datasets using torch.utils.data.DataLoader.
6. Instantiate your CNN model and define the loss function and optimizer.
7. Train your CNN by iterating over the training dataset, computing forward and backward passes, and optimizing the model weights using the optimizer.
8. Evaluate the performance of your model on the test dataset by computing predictions and comparing them to the true labels.
9. Save the trained model parameters using torch. save for future use.

Procedure:

The code is implementing a convolutional neural network (CNN) using PyTorch to perform image classification on the MNIST dataset.

First, the code defines hyperparameters such as the number of epochs, batch size, and learning rate. Then, it defines the CNN architecture as a class with several convolutional layers, activation functions, and a fully connected layer.

The MNIST dataset is loaded and transformed using PyTorch's built-in functions, and then the data is split into training and test sets using data loaders.

The CNN model is instantiated, and the loss function (cross-entropy) and optimizer (Adam) are defined. The model is then trained for the specified number of epochs, with the loss being printed every 100 steps.

After training, the model is tested on the test set to calculate its accuracy. Finally, the trained model is saved to a file using PyTorch's state_dict() function.

Conclusion: Thus, we have studied PyTorch implementation of CNN

Program Code and Output

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision.datasets as datasets
import torchvision.transforms as transforms

# Define hyperparameters
num_epochs = 5
batch_size = 64
learning_rate = 0.001

# Define the CNN architecture
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, stride=1, padding=1)
        self.relu1 = nn.ReLU()
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, stride=1, padding=1)
        self.relu2 = nn.ReLU()
        self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(in_features=7*7*32, out_features=10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.pool1(x)
        x = self.conv2(x)
```

```
        x = self.relu2(x)
        x = self.pool2(x)
        x = x.view(-1, 7*7*32)
        x = self.fc1(x)
        return x

# Load the MNIST dataset
train_dataset = datasets.MNIST(root='./data', train=True, transform=transforms.ToTensor(), download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=transforms.ToTensor())

# Create the dataloaders
train_loader = torch.utils.data.DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=False)

# Instantiate the CNN model and define the loss function and optimizer
model = CNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Train the model
for epoch in range(num_epochs):
    for i, (images, labels) in enumerate(train_loader):
        # Forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)

        # Backward and optimize
        optimizer.zero_grad()
```

```
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                .format(epoch+1, num_epochs, i+1, len(train_loader), loss.item()))

# Test the model
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    print('Test Accuracy of the model on the 10000 test images: {} %'.format(100 * correct / total))
# Save the trained model
torch.save(model.state_dict(), 'cnn.ckpt')
```

OUTPUT OF THE CODE

```
>>> %Run -c $EDITOR_CONTENT

Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz

Processing

Done!

C:\Users\ted7\AppData\Roaming\Python\Python310\site-packages\torchvision\transforms.py:36:
UserWarning: TypedStorage is deprecated. It will be removed in the future and UntypedStorage will be
the only storage class. This should only matter to you if you are using storages directly.  To access
UntypedStorage directly, use tensor.untyped_storage() instead of tensor.storage()

  img = torch.ByteTensor(torch.ByteStorage.from_buffer(pic.tobytes()))

Epoch [1/5], Step [100/938], Loss: 0.2831

Epoch [1/5], Step [200/938], Loss: 0.3648

Epoch [1/5], Step [300/938], Loss: 0.1767

Epoch [1/5], Step [400/938], Loss: 0.1572

Epoch [1/5], Step [500/938], Loss: 0.1258

Epoch [1/5], Step [600/938], Loss: 0.1229

Epoch [1/5], Step [700/938], Loss: 0.0968

Epoch [1/5], Step [800/938], Loss: 0.0415

Epoch [1/5], Step [900/938], Loss: 0.0414

Epoch [2/5], Step [100/938], Loss: 0.0179

Epoch [2/5], Step [200/938], Loss: 0.0380

Epoch [2/5], Step [300/938], Loss: 0.0672

Epoch [2/5], Step [400/938], Loss: 0.1538

Epoch [2/5], Step [500/938], Loss: 0.0923

Epoch [2/5], Step [600/938], Loss: 0.1063

Epoch [2/5], Step [700/938], Loss: 0.1835

Epoch [2/5], Step [800/938], Loss: 0.0842
```

Epoch [2/5], Step [900/938], Loss: 0.0146

Epoch [3/5], Step [100/938], Loss: 0.0270

Epoch [3/5], Step [200/938], Loss: 0.1467

Epoch [3/5], Step [300/938], Loss: 0.0634

Epoch [3/5], Step [400/938], Loss: 0.0171

Epoch [3/5], Step [500/938], Loss: 0.0640

Epoch [3/5], Step [600/938], Loss: 0.0037

Epoch [3/5], Step [700/938], Loss: 0.0324

Epoch [3/5], Step [800/938], Loss: 0.0483

Epoch [3/5], Step [900/938], Loss: 0.0074

Epoch [4/5], Step [100/938], Loss: 0.0138

Epoch [4/5], Step [200/938], Loss: 0.0166

Epoch [4/5], Step [300/938], Loss: 0.0386

Epoch [4/5], Step [400/938], Loss: 0.0541

Epoch [4/5], Step [500/938], Loss: 0.0296

Epoch [4/5], Step [600/938], Loss: 0.0387

Epoch [4/5], Step [700/938], Loss: 0.0138

Epoch [4/5], Step [800/938], Loss: 0.0690

Epoch [4/5], Step [900/938], Loss: 0.0088

Epoch [5/5], Step [100/938], Loss: 0.0230

Epoch [5/5], Step [200/938], Loss: 0.0729

Epoch [5/5], Step [300/938], Loss: 0.0081

Epoch [5/5], Step [400/938], Loss: 0.0435

Epoch [5/5], Step [500/938], Loss: 0.0470

Epoch [5/5], Step [600/938], Loss: 0.0420

Epoch [5/5], Step [700/938], Loss: 0.0339

Epoch [5/5], Step [800/938], Loss: 0.0064

Epoch [5/5], Step [900/938], Loss: 0.0098

Test Accuracy of the model on the 10000 test images: 98.75 %

Aim: For an image classification challenge create and train a ConvNet in python using Tensorflow. Also try to improve the performance of the model by applying various hyperparameter tuning to reduce overfitting or underfitting problem that might occur. Maintain graphs of comparisons.

Software requirement for Python: Jupyter Notebook

Theory:

To implement an image classification challenge using Tensorflow, follow these steps:

1. Load the dataset: Load the image dataset you want to use for the classification task. You can use popular image datasets such as MNIST, CIFAR-10, or ImageNet.
2. Preprocess the dataset: Preprocess the image dataset by resizing the images, normalizing the pixel values, and dividing it into training, validation, and testing sets.
3. Define the ConvNet architecture: Define the architecture of the ConvNet using the Keras Sequential model. Add Conv2D layers, MaxPooling2D layers, and Dropout layers as required.
4. Compile the model: Compile the model using an optimizer, loss function, and evaluation metric.
5. Train the model: Train the model on the training set for a fixed number of epochs using the fit() method. Use the validation set to monitor the performance of the model and avoid overfitting.
6. Evaluate the model: Evaluate the performance of the model on the testing set using the evaluate() method.
7. Hyperparameter tuning: Use techniques such as grid search, random search, and Bayesian optimization to find the optimal hyperparameters for the model. Tune the learning rate, batch size, number of filters, and other hyperparameters to improve the performance of the model.
8. Regularization techniques: Apply regularization techniques such as L1/L2 regularization, dropout, and data augmentation to reduce overfitting or underfitting problems that might occur.

Procedure:

1. This code is implementing an image classification challenge using a Convolutional Neural Network (ConvNet) in Python with TensorFlow. The CIFAR-10 dataset is being used for this task, which consists of 50,000 training images and 10,000 test images, with 10 different classes of objects. The following steps are performed in the code:
2. Load and preprocess the CIFAR-10 dataset: The dataset is loaded into memory and preprocessed by scaling the pixel values of the images to the range of [0,1].
3. Define the ConvNet architecture: A sequential model is created using the Keras API with a series of convolutional layers, max pooling layers, dropout layers, and fully connected layers. The architecture consists of 2 blocks of convolutional layers with a max pooling layer and a dropout layer following each block, followed by a flatten layer, two dense layers, and a final output layer with softmax activation.
4. Compile the model: The model is compiled with the Adam optimizer and 'sparse_categorical_crossentropy' as the loss function.
5. Set up data augmentation: Data augmentation is used to increase the size of the dataset and reduce overfitting. The ImageDataGenerator class from Keras is used to apply random transformations to the training images, such as rotation, width and height shifts, and horizontal flipping.

6. Train the model: The fit method of the model is called with the data generator and other hyperparameters, such as batch size and number of epochs.
7. Evaluate the model: The evaluate method is called on the test set to get the test loss and accuracy of the model.
8. Plot the accuracy and loss curves: The training and validation accuracy and loss curves are plotted using the Matplotlib library to visualize the performance of the model over the training epochs.

Conclusion: Thus, we have studied an image classification challenge to create and train a ConvNet in python using Tensorflow.

Program Code and Output

```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt

# Load and preprocess the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train = x_train / 255.0
x_test = x_test / 255.0

# Define the ConvNet architecture
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(32, 32, 3)))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.5))
```

```
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer=Adam(learning_rate=0.001),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Set up data augmentation
datagen = ImageDataGenerator(rotation_range=15,
                             width_shift_range=0.1,
                             height_shift_range=0.1,
                             horizontal_flip=True)

# Train the model
history = model.fit(datagen.flow(x_train, y_train, batch_size=128),
                    epochs=50,
                    validation_data=(x_test, y_test))

# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f'Test Loss: {test_loss:.4f}')
print(f'Test Accuracy: {test_acc:.4f}')

# Plot the accuracy and loss curves
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Accuracy')
```

```
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.tight_layout()
plt.show()
```

OUTPUT OF THE CODE

```
>>> %Run 'GROUP C_4.py'

Epoch 1/50

391/391 [==============================] - 99s 250ms/step - loss: 1.7248 - accuracy: 0.1030 - val_loss: 1.3772 - val_accuracy: 0.0655

Epoch 2/50

391/391 [==============================] - 127s 325ms/step - loss: 1.3879 - accuracy: 0.0958 - val_loss: 1.2349 - val_accuracy: 0.1099

Epoch 3/50

391/391 [==============================] - 114s 292ms/step - loss: 1.2416 - accuracy: 0.0983 - val_loss: 1.0503 - val_accuracy: 0.0805

Epoch 4/50

391/391 [==============================] - 116s 297ms/step - loss: 1.1595 - accuracy: 0.0993 - val_loss: 0.9859 - val_accuracy: 0.0880

Epoch 5/50

391/391 [==============================] - 118s 303ms/step - loss: 1.0877 - accuracy: 0.0986 - val_loss: 0.9002 - val_accuracy: 0.0991

Epoch 6/50

391/391 [==============================] - 119s 304ms/step - loss: 1.0244 - accuracy: 0.0999 - val_loss: 0.8383 - val_accuracy: 0.0931

Epoch 7/50

391/391 [==============================] - 131s 335ms/step - loss: 0.9825 - accuracy: 0.0997 - val_loss: 0.7868 - val_accuracy: 0.1026

Epoch 8/50

391/391 [==============================] - 141s 360ms/step - loss: 0.9517 - accuracy: 0.1001 - val_loss: 0.7742 - val_accuracy: 0.0811

Epoch 9/50

391/391 [==============================] - 147s 377ms/step - loss: 0.9256 - accuracy: 0.1006 - val_loss: 0.7405 - val_accuracy: 0.1036

Epoch 10/50

391/391 [==============================] - 142s 362ms/step - loss: 0.8991 - accuracy: 0.0996 - val_loss: 0.7711 - val_accuracy: 0.0972

Epoch 11/50
```

391/391 [==============================] - 139s 357ms/step - loss: 0.8799 - accuracy: 0.1024 - val_loss: 0.7556 - val_accuracy: 0.1119

Epoch 12/50

391/391 [==============================] - 135s 344ms/step - loss: 0.8639 - accuracy: 0.1008 - val_loss: 0.8286 - val_accuracy: 0.0987

Epoch 13/50

391/391 [==============================] - 82s 209ms/step - loss: 0.8410 - accuracy: 0.1019 - val_loss: 0.7132 - val_accuracy: 0.1033

Epoch 14/50

391/391 [==============================] - 87s 223ms/step - loss: 0.8264 - accuracy: 0.1009 - val_loss: 0.7772 - val_accuracy: 0.0989

Epoch 15/50

391/391 [==============================] - 97s 247ms/step - loss: 0.8172 - accuracy: 0.1035 - val_loss: 0.7462 - val_accuracy: 0.0923

Epoch 16/50

391/391 [==============================] - 91s 234ms/step - loss: 0.7979 - accuracy: 0.1023 - val_loss: 0.6487 - val_accuracy: 0.0955

Epoch 17/50

391/391 [==============================] - 94s 239ms/step - loss: 0.8003 - accuracy: 0.1029 - val_loss: 0.6650 - val_accuracy: 0.1051

Epoch 18/50

391/391 [==============================] - 90s 229ms/step - loss: 0.7865 - accuracy: 0.1022 - val_loss: 0.6724 - val_accuracy: 0.0995

Epoch 19/50

391/391 [==============================] - 88s 225ms/step - loss: 0.7717 - accuracy: 0.1033 - val_loss: 0.6833 - val_accuracy: 0.0951

Epoch 20/50

391/391 [==============================] - 88s 225ms/step - loss: 0.7625 - accuracy: 0.1020 - val_loss: 0.6469 - val_accuracy: 0.0969

Epoch 21/50

391/391 [==============================] - 87s 222ms/step - loss: 0.7555 - accuracy: 0.1020 - val_loss: 0.6600 - val_accuracy: 0.1045

Epoch 22/50

391/391 [==============================] - 87s 223ms/step - loss: 0.7479 - accuracy: 0.1013 - val_loss: 0.6610 - val_accuracy: 0.0972

Epoch 23/50

391/391 [==============================] - 90s 229ms/step - loss: 0.7477 - accuracy: 0.1023 - val_loss: 0.6384 - val_accuracy: 0.1121

Epoch 24/50

391/391 [==============================] - 91s 232ms/step - loss: 0.7366 - accuracy: 0.1028 - val_loss: 0.7233 - val_accuracy: 0.0957

Epoch 25/50

391/391 [==============================] - 88s 225ms/step - loss: 0.7281 - accuracy: 0.1023 - val_loss: 0.6643 - val_accuracy: 0.1054

Epoch 26/50

391/391 [==============================] - 89s 228ms/step - loss: 0.7222 - accuracy: 0.1020 - val_loss: 0.6386 - val_accuracy: 0.0993

Epoch 27/50

391/391 [==============================] - 88s 225ms/step - loss: 0.7097 - accuracy: 0.1032 - val_loss: 0.6283 - val_accuracy: 0.1012

Epoch 28/50

391/391 [==============================] - 88s 225ms/step - loss: 0.7086 - accuracy: 0.1021 - val_loss: 0.6461 - val_accuracy: 0.0954

Epoch 29/50

391/391 [==============================] - 90s 231ms/step - loss: 0.7081 - accuracy: 0.1023 - val_loss: 0.6026 - val_accuracy: 0.1013

Epoch 30/50

391/391 [==============================] - 89s 227ms/step - loss: 0.6984 - accuracy: 0.1027 - val_loss: 0.5945 - val_accuracy: 0.0939

Epoch 31/50

391/391 [==============================] - 88s 224ms/step - loss: 0.6926 - accuracy: 0.1016 - val_loss: 0.6842 - val_accuracy: 0.0996

Epoch 32/50

391/391 [==============================] - 88s 224ms/step - loss: 0.6868 - accuracy: 0.1026 - val_loss: 0.6042 - val_accuracy: 0.0948

Epoch 33/50

391/391 [==============================] - 88s 226ms/step - loss: 0.6919 - accuracy: 0.1023 - val_loss: 0.6046 - val_accuracy: 0.0975

Epoch 34/50

391/391 [==============================] - 89s 227ms/step - loss: 0.6845 - accuracy: 0.1020 - val_loss: 0.6066 - val_accuracy: 0.0944

Epoch 35/50

391/391 [==============================] - 88s 226ms/step - loss: 0.6814 - accuracy: 0.1030 - val_loss: 0.6341 - val_accuracy: 0.1030

Epoch 36/50

391/391 [==============================] - 88s 226ms/step - loss: 0.6771 - accuracy: 0.1028 - val_loss: 0.6575 - val_accuracy: 0.0953

Epoch 37/50

391/391 [==============================] - 89s 227ms/step - loss: 0.6741 - accuracy: 0.1027 - val_loss: 0.5642 - val_accuracy: 0.0929

Epoch 38/50

391/391 [==============================] - 89s 227ms/step - loss: 0.6721 - accuracy: 0.1029 - val_loss: 0.6416 - val_accuracy: 0.0919

Epoch 39/50

391/391 [==============================] - 90s 231ms/step - loss: 0.6735 - accuracy: 0.1024 - val_loss: 0.5674 - val_accuracy: 0.1008

Epoch 40/50

391/391 [==============================] - 86s 220ms/step - loss: 0.6622 - accuracy: 0.1032 - val_loss: 0.6514 - val_accuracy: 0.0896

Epoch 41/50

391/391 [==============================] - 91s 233ms/step - loss: 0.6578 - accuracy: 0.1031 - val_loss: 0.6505 - val_accuracy: 0.1189

Epoch 42/50

391/391 [==============================] - 93s 237ms/step - loss: 0.6585 - accuracy: 0.1028 - val_loss: 0.5454 - val_accuracy: 0.1092

Epoch 43/50

391/391 [==============================] - 90s 231ms/step - loss: 0.6624 - accuracy: 0.1030 - val_loss: 0.6359 - val_accuracy: 0.1016

Epoch 44/50

391/391 [==============================] - 92s 234ms/step - loss: 0.6503 - accuracy: 0.1038 - val_loss: 0.5993 - val_accuracy: 0.0920

Epoch 45/50

391/391 [==============================] - 88s 226ms/step - loss: 0.6499 - accuracy: 0.1038 - val_loss: 0.5999 - val_accuracy: 0.0940

Epoch 46/50

391/391 [==============================] - 88s 226ms/step - loss: 0.6500 - accuracy: 0.1022 - val_loss: 0.6407 - val_accuracy: 0.0937

Epoch 47/50

391/391 [==============================] - 89s 226ms/step - loss: 0.6467 - accuracy: 0.1020 - val_loss: 0.6009 - val_accuracy: 0.1112

Epoch 48/50

391/391 [==============================] - 88s 226ms/step - loss: 0.6494 - accuracy: 0.1030 - val_loss: 0.6091 - val_accuracy: 0.0937

Epoch 49/50

391/391 [==============================] - 88s 225ms/step - loss: 0.6468 - accuracy: 0.1035 - val_loss: 0.5794 - val_accuracy: 0.1005

Epoch 50/50

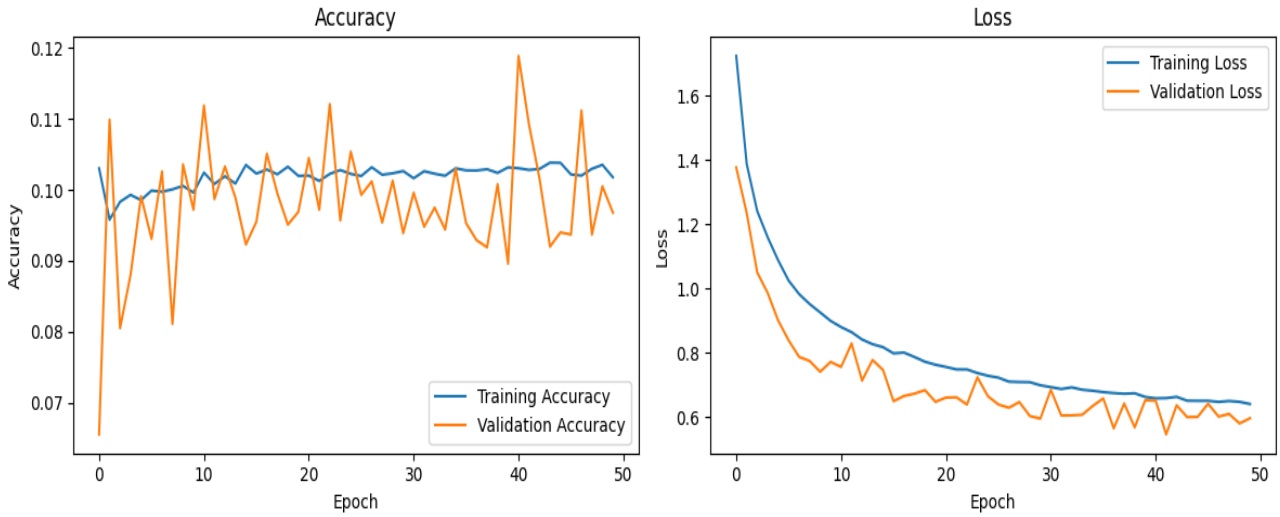
391/391 [==============================] - 88s 226ms/step - loss: 0.6400 - accuracy: 0.1018 - val_loss: 0.5958 - val_accuracy: 0.0968

313/313 - 4s - loss: 0.5958 - accuracy: 0.0968 - 4s/epoch - 12ms/step

Test Loss: 0.5958

Test Accuracy: 0.0968

Graphs of comparison for Accuracy Vs Epochs and Loss vs Epochs



Aim: MNIST Handwritten character detection using PyTorch, Keras and Tensorflow.

Software requirement for Python: Jupyter Notebook

Theory:

1. The steps for MNIST Handwritten character detection using PyTorch, Keras, and Tensorflow are similar and involve the following:
2. Load the dataset: The first step is to load the MNIST dataset, which contains 60,000 training images and 10,000 test images of handwritten digits.
3. Prepare the data: In this step, you will normalize the data and convert the labels to one-hot vectors.
4. Build the model: You will build a neural network model with input layer, hidden layers, and output layer.
5. Compile the model: You will compile the model by specifying the loss function, optimizer, and metrics.
6. Train the model: In this step, you will train the model on the training dataset using the fit method.
7. Evaluate the model: After training, you will evaluate the performance of the model on the test dataset using the evaluate method.
8. Make predictions: Finally, you will make predictions on new data using the predict method.

Procedure:

1. This code implements a neural network model using the Keras API in TensorFlow for the MNIST handwritten digit recognition task. The code first loads and preprocesses the MNIST dataset by scaling the pixel values to the range [0, 1].
2. Then, a simple feedforward neural network with one hidden layer is defined using the Sequential model. The input layer is a Flatten layer that converts the 28x28 input images to a 784-dimensional vector. The hidden layer has 128 units with the ReLU activation function. The output layer has 10 units with the softmax activation function that outputs the probability distribution over 10 classes.
3. The model is compiled using the Adam optimizer with a learning rate of 0.001 and the sparse categorical cross-entropy loss function. The accuracy metric is also specified for evaluation.
4. The model is then trained on the training set using the fit() method with a batch size of 128 and for 10 epochs.
5. Finally, the model is evaluated on the test set using the evaluate() method, and the test loss and accuracy are printed.

Conclusion: Thus, we have studied MNIST Handwritten character detection using PyTorch, Keras and Tensorflow.

Program Code and Output

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import Adam
# Load and preprocess the MNIST dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train / 255.0
x_test = x_test / 255.0

# Define the neural network architecture
model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(128, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer=Adam(learning_rate=0.001),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
# Train the model
model.fit(x_train, y_train, batch_size=128, epochs=10, verbose=1)

# Evaluate the model on the test set
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print(f'Test Loss: {test_loss:.4f}')
print(f'Test Accuracy: {test_acc:.4f}')
```

OUTPUT OF THE CODE

```
Epoch 1/10
469/469 [==============================] - 2s 3ms/step - loss: 0.3584 - accuracy: 0.9014
Epoch 2/10
469/469 [==============================] - 1s 3ms/step - loss: 0.1667 - accuracy: 0.9522
Epoch 3/10
469/469 [==============================] - 1s 3ms/step - loss: 0.1191 - accuracy: 0.9651
Epoch 4/10
469/469 [==============================] - 1s 3ms/step - loss: 0.0934 - accuracy: 0.9728
Epoch 5/10
469/469 [==============================] - 1s 3ms/step - loss: 0.0757 - accuracy: 0.9784
Epoch 6/10
469/469 [==============================] - 1s 3ms/step - loss: 0.0632 - accuracy: 0.9817
Epoch 7/10
469/469 [==============================] - 1s 3ms/step - loss: 0.0526 - accuracy: 0.9850
Epoch 8/10
469/469 [==============================] - 1s 3ms/step - loss: 0.0453 - accuracy: 0.9872
Epoch 9/10
469/469 [==============================] - 1s 3ms/step - loss: 0.0378 - accuracy: 0.9896
Epoch 10/10
469/469 [==============================] - 1s 3ms/step - loss: 0.0323 - accuracy: 0.9913
313/313 - 1s - loss: 0.0716 - accuracy: 0.9782 - 613ms/epoch - 2ms/step
Test Loss: 0.0716
Test Accuracy: 0.9782
```