```
# Import Python Libraries
import numpy as np

# Take two sets of patterns:
# Set A: Input Pattern
x1 = np.array([1, 1, 1, 1, 1, 1]).reshape(6, 1)
x2 = np.array([-1, -1, -1, -1, -1, -1]).reshape(6, 1)
x3 = np.array([1, 1, -1, -1, 1, 1]).reshape(6, 1)
x4 = np.array([-1, -1, 1, 1, -1, -1]).reshape(6, 1)

# Set B: Target Pattern
y1 = np.array([1, 1, 1]).reshape(3, 1)
y2 = np.array([-1, -1, -1]).reshape(3, 1)
y3 = np.array([1, -1, 1]).reshape(3, 1)
y4 = np.array([-1, 1, -1]).reshape(3, 1)

'''
print("Set A: Input Pattern, Set B: Target Pattern")
print("\nThe input for pattern 1 is")
print(x1)
print("\nThe target for pattern 1 is")
print(y1)
print("\nThe input for pattern 2 is")
print(x2)
print("\nThe target for pattern 2 is")
print(y2)
print("\nThe input for pattern 3 is")
print(x3)
print("\nThe target for pattern 3 is")
print(y3)
print("\nThe input for pattern 4 is")
```

```
print(x4)
print("\nThe target for pattern 4 is")
print(y4)

print("\n------------------------------")
'''
# Calculate weight Matrix: W
inputSet = np.concatenate((x1, x2, x3, x4), axis = 1)
targetSet = np.concatenate((y1.T, y2.T, y3.T, y4.T), axis = 0)
print("\nWeight matrix:")
weight = np.dot(inputSet, targetSet)
print(weight)

print("\n------------------------------")

# Testing Phase
# Test for Input Patterns: Set A
print("\nTesting for input patterns: Set A")
def testInputs(x, weight):
  # Multiply the input pattern with the weight matrix
  # (weight.T X x)
  y = np.dot(weight.T, x)
  y[y < 0] = -1
  y[y >= 0] = 1
  return np.array(y)

print("\nOutput of input pattern 1")
print(testInputs(x1, weight))
print("\nOutput of input pattern 2")
print(testInputs(x2, weight))
print("\nOutput of input pattern 3")
```

```
print(testInputs(x3, weight))
print("\nOutput of input pattern 4")
print(testInputs(x4, weight))

# Test for Target Patterns: Set B
print("\nTesting for target patterns: Set B")
def testTargets(y, weight):
  # Multiply the target pattern with the weight matrix
  # (weight X y)
  x = np.dot(weight, y)
  x[x <= 0] = -1
  x[x > 0] = 1
  return np.array(x)

print("\nOutput of target pattern 1")
print(testTargets(y1, weight))
print("\nOutput of target pattern 2")
print(testTargets(y2, weight))
print("\nOutput of target pattern 3")
print(testTargets(y3, weight))
print("\nOutput of target pattern 4")
print(testTargets(y4, weight))
```

==========output==========

Weight matrix:

```
[[4 0 4]
 [4 0 4]
 [0 4 0]
```

[0 4 0]

[4 0 4]

[4 0 4]]

Testing for input patterns: Set A

Output of input pattern 1

[[1]

[1]

[1]]

Output of input pattern 2

[[-1]

[-1]

[-1]]

Output of input pattern 3

[[1]

[-1]

[1]]

Output of input pattern 4

[[-1]

[1]

[-1]]

Testing for target patterns: Set B

Output of target pattern 1

[[1]

[1]

[1]

[1]

[1]

[1]]

Output of target pattern 2

[[-1]

[-1]

[-1]

[-1]

[-1]

[-1]]

Output of target pattern 3

[[1]

[1]

[-1]

[-1]

[1]

[1]]

Output of target pattern 4

[[-1]

[-1]

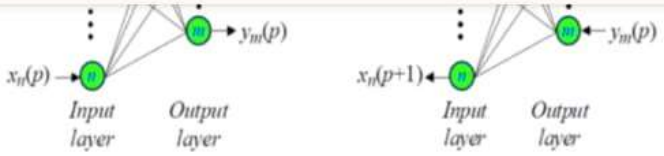
[1]

[1]

[-1]

[-1]]

=== Code Execution Successful ===



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

Algorithm:

Step 0: Initialize the weights to store p vectors. Also initialize all the activations to zero.
Step 1: Perform Steps 2-6 for each testing input.
Step 2: Ser the activations of X layer to current input pattern, i.e., presenting the
layer similarly presenting the input pattern y to Y layer. Even though it is
at one time step, signals can be sent from only one layer. So, either of the input patterns may
the zero vector
Step 3: Perform Steps 4-6 when the activations are not converged.
Step 4: Update the activations of units in the Y layer. Calculate the net input,

$$y_{inj} = \sum_{i=1}^{n} x_i w_{ij}$$

Applying activations, we obtain

$$y_j = f(y_{inj})$$

Send this signal to the X layer.

Step 5: Update the activations of units in X layer. Calculate the net input,

$$x_{ini} = \sum_{j=1}^{m} y_j w_{ij}$$

Applying activations, we obtain

$$x_i = f(x_{ini})$$

Send this signal to the Y layer.

Step 6: Test for convergence of the net. The convergence occurs if the activation vectors x and y reach equilibrium. If this occurs then stop, Otherwise, continue.

Conclusion:

We have successfully implemented python Program for Bidirectional A
with two pairs of vectors.

Assignment – 5

Title:

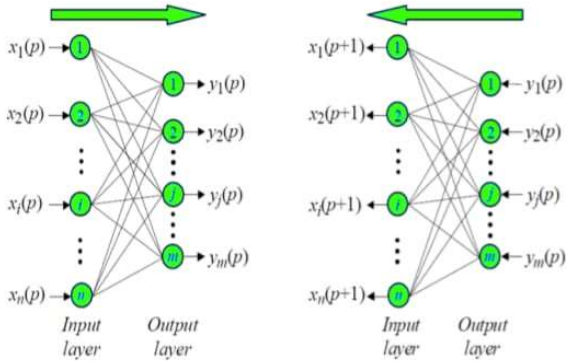
Bidirectional Associative Memory with two pairs of vectors.

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

Objective: To learn about Bidirectional Associative Memory with two pairs of vectors.

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.



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

D. Y. PATIL COLLEGE OF ENGINEERING, AKURDI PUNE 411044
Department of Artificial Intelligence & Data Science

ANN Lab Manual

gorithm:

ep 0: Initialize the weights to store p vectors. Also initialize all the activations to zero.
ep 1: Perform Steps 2-6 for each testing input.
ep 2: Ser the activations of X layer to current input pattern, i.e., presenting the input pattern x to X layer similarly presenting the input pattern y to Y layer. Even though it is bidirectional memory, at one time step, signals can be sent from only one layer. So, either of the input patterns may be the zero vector
ep 3: Perform Steps 4-6 when the activations are not converged.
ep 4: Update the activations of units in the Y layer. Calculate the net input,

$$y_{inj} = \sum_{i=1}^{n} x_i w_{ij}$$