UNIVERSITY OF TEHRAN

COLLEGE OF ENGINEERING

DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

NEURAL NETWORK & DEEP LEARNING

ASSIGNMENT#3

MOHAMMAD HEYDARI

810197494

UNDER SUPERVISION OF:

DR. AHMAD KALHOR

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

UNIVERSITY OF TEHRAN

May. 2022

1 CONTENTS

2 QUESTION #1: PATTERN ASSOCIATION (HEBBIAN RULE)

In this part we intend to implement Hebbian Learning Rule in case of Pattern Association.

Further, we are going to report the results of this section one by one:

2.1 HEBBIAN LEARNING RULE DESCRIPTION

Hebbian Learning Rule, also known as Hebb Learning Rule, was proposed by Donald O Hebb. It is one of the first and also easiest learning rules in the neural network.

It is used for pattern classification. It is a single layer neural network, i.e., it has one input layer and one output layer. The input layer can have many units, say n. The output layer only has one unit. Hebbian rule works by updating the weights between neurons in the neural network for each training sample.

Hebbian Learning Rule Algorithm :

Step1: Set all weights to zero, wi = 0 for i=1 to n, and bias to zero.

Step2: For each input vector, S(input vector) : t(target output pair), repeat steps 3-5.

Step3: Set activations for input units with the input vector Xi = Si for i = 1 to n.

Step4: Set the corresponding output value to the output neuron, i.e., y = t.

Step5: Update weight and bias by applying Hebb rule for all i = 1 to n:

$$w_i\,(\text{new}) = w_i\,(\text{old}) + x_i\,y$$
$$b\,(\text{new}) = b\,(\text{old}) + y$$

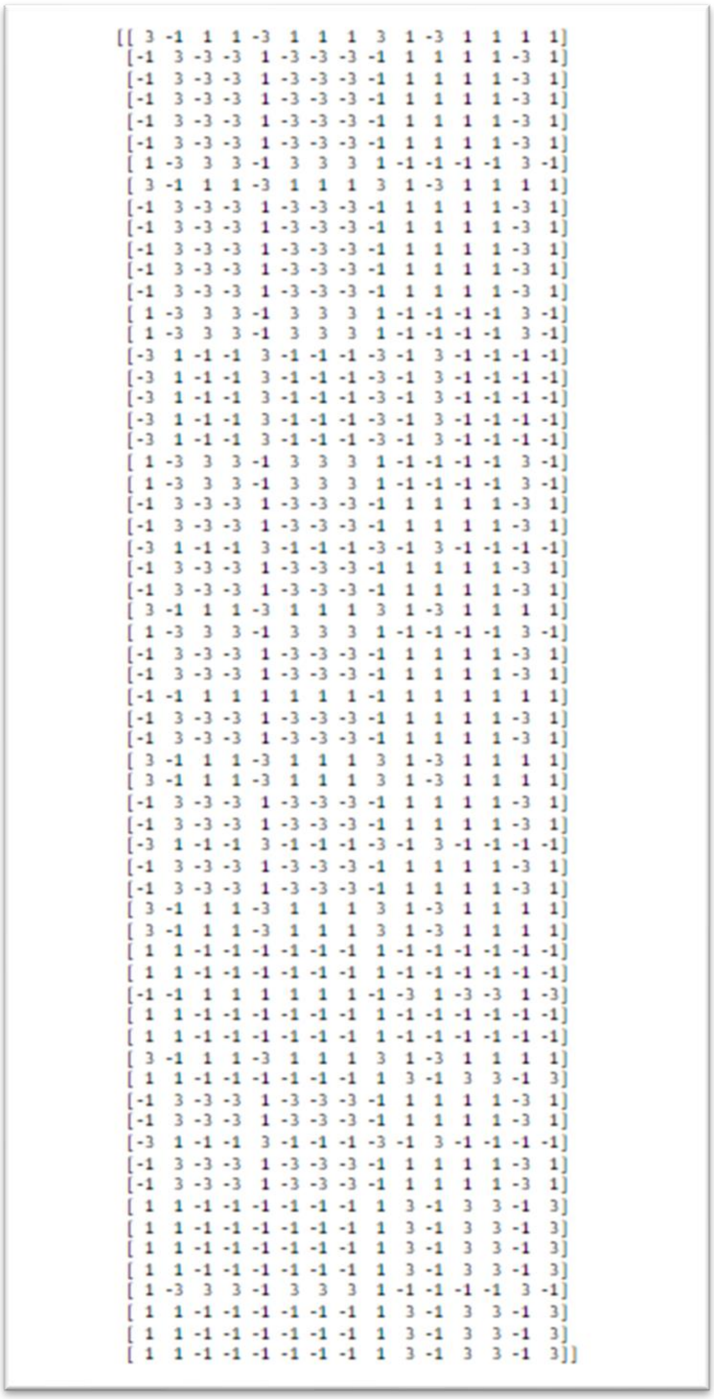
2.2 UPDATING NETWORK WEIGHTS

In this part we have used Hebb rule to update the network weights.

The input patterns have a dimension of **(9*7)** and their correspond outputs have a dimension of **(5*3).**

Note that in this section I have used a list of **63 elements** as of my input patterns and also a list of **15 elements** as of my output patterns furthermore, the weight matrix has a dimension of **(63,15)** in this case.

The last-edition of weight matrix which produce using Hebb Rule:



```
[[ 3 -1  1  1 -3  1  1  1  3  1 -3  1  1  1  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [ 1 -3  3  3 -1  3  3  3  1 -1 -1 -1 -1  3 -1]
 [ 3 -1  1  1 -3  1  1  1  3  1 -3  1  1  1  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [ 1 -3  3  3 -1  3  3  3  1 -1 -1 -1 -1  3 -1]
 [ 1 -3  3  3 -1  3  3  3  1 -1 -1 -1 -1  3 -1]
 [-3  1 -1 -1  3 -1 -1 -1 -3 -1  3 -1 -1 -1 -1]
 [-3  1 -1 -1  3 -1 -1 -1 -3 -1  3 -1 -1 -1 -1]
 [-3  1 -1 -1  3 -1 -1 -1 -3 -1  3 -1 -1 -1 -1]
 [-3  1 -1 -1  3 -1 -1 -1 -3 -1  3 -1 -1 -1 -1]
 [-3  1 -1 -1  3 -1 -1 -1 -3 -1  3 -1 -1 -1 -1]
 [ 1 -3  3  3 -1  3  3  3  1 -1 -1 -1 -1  3 -1]
 [ 1 -3  3  3 -1  3  3  3  1 -1 -1 -1 -1  3 -1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-3  1 -1 -1  3 -1 -1 -1 -3 -1  3 -1 -1 -1 -1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [ 3 -1  1  1 -3  1  1  1  3  1 -3  1  1  1  1]
 [ 1 -3  3  3 -1  3  3  3  1 -1 -1 -1 -1  3 -1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-1 -1  1  1  1  1  1  1 -1  1  1  1  1  1  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [ 3 -1  1  1 -3  1  1  1  3  1 -3  1  1  1  1]
 [ 3 -1  1  1 -3  1  1  1  3  1 -3  1  1  1  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-3  1 -1 -1  3 -1 -1 -1 -3 -1  3 -1 -1 -1 -1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [ 3 -1  1  1 -3  1  1  1  3  1 -3  1  1  1  1]
 [ 3 -1  1  1 -3  1  1  1  3  1 -3  1  1  1  1]
 [ 1  1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1]
 [ 1  1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1]
 [-1 -1  1  1  1  1  1  1 -1 -3  1 -3 -3  1 -3]
 [ 1  1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1]
 [ 1  1 -1 -1 -1 -1 -1 -1  1 -1 -1 -1 -1 -1 -1]
 [ 3 -1  1  1 -3  1  1  1  3  1 -3  1  1  1  1]
 [ 1  1 -1 -1 -1 -1 -1 -1  1  3 -1  3  3 -1  3]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-3  1 -1 -1  3 -1 -1 -1 -3 -1  3 -1 -1 -1 -1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [-1  3 -3 -3  1 -3 -3 -3 -1  1  1  1  1 -3  1]
 [ 1  1 -1 -1 -1 -1 -1 -1  1  3 -1  3  3 -1  3]
 [ 1  1 -1 -1 -1 -1 -1 -1  1  3 -1  3  3 -1  3]
 [ 1  1 -1 -1 -1 -1 -1 -1  1  3 -1  3  3 -1  3]
 [ 1  1 -1 -1 -1 -1 -1 -1  1  3 -1  3  3 -1  3]
 [ 1 -3  3  3 -1  3  3  3  1 -1 -1 -1 -1  3 -1]
 [ 1  1 -1 -1 -1 -1 -1 -1  1  3 -1  3  3 -1  3]
 [ 1  1 -1 -1 -1 -1 -1 -1  1  3 -1  3  3 -1  3]
 [ 1  1 -1 -1 -1 -1 -1 -1  1  3 -1  3  3 -1  3]]
```

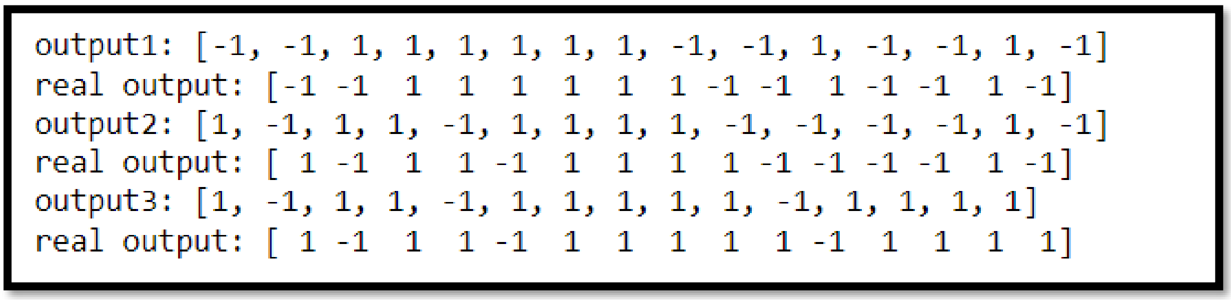
Figure1. Weight Matrix using Hebb Rule

As we expect the matrix has a dimension of **(63,15).**

2.3 Predicting the Inputs Using Weight Matrix

In this section, I have written a function which get us the prediction of input using Hebb rule.

Results of prediction:



```
output1: [-1, -1, 1, 1, 1, 1, 1, 1, -1, -1, 1, -1, -1, 1, -1]
real output: [-1 -1  1  1  1  1  1  1 -1 -1  1 -1 -1  1 -1]
output2: [1, -1, 1, 1, -1, 1, 1, 1, 1, -1, -1, -1, -1, 1, -1]
real output: [ 1 -1  1  1 -1  1  1  1  1 -1 -1 -1 -1  1 -1]
output3: [1, -1, 1, 1, -1, 1, 1, 1, 1, 1, -1, 1, 1, 1, 1]
real output: [ 1 -1  1  1 -1  1  1  1  1  1 -1  1  1  1  1]
```

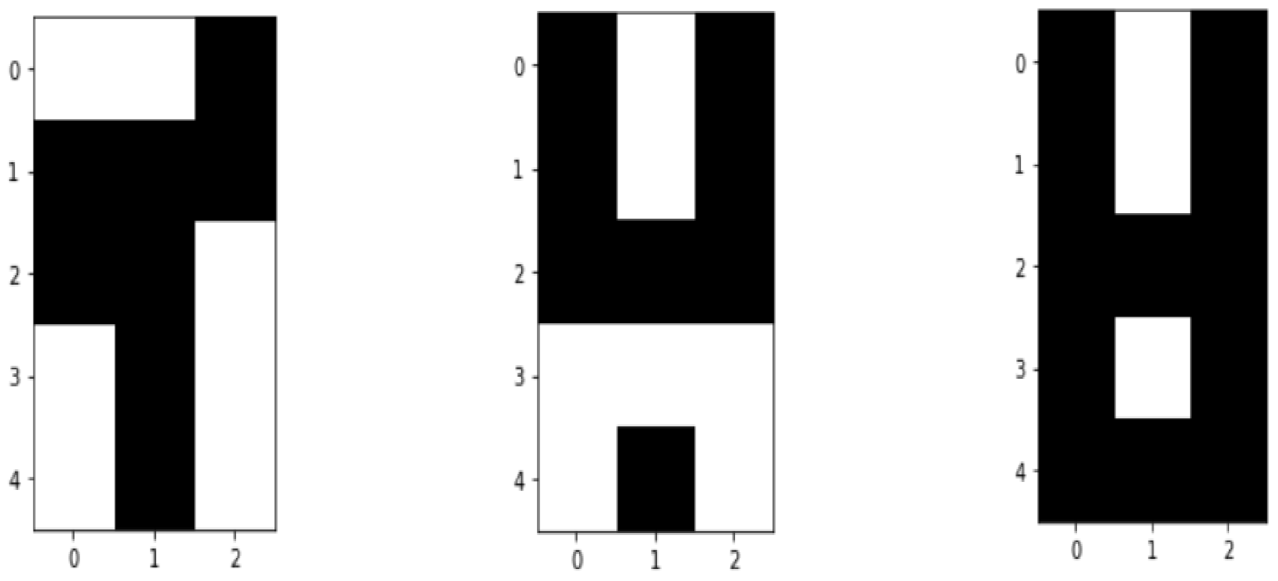


Figure2. Prediction Results

2.4 THE MINIMUM SIZE OF NETWORK TO PREDICT PROPERLY

We have two bipolar bits: **{-1,1}**

and we are going to **find the minimum dimension** for the output that could enable us to have **three separated pattern.**

As best of our knowledge, we know that using three bits we will have $2^3 = 8$ separated patterns and using the same concept, we will have $2^2 = 4$ separated pattern when **we are dealing with 2 bits!**

So, we must use our 2 bits in a below form to make the minimum possible dimension for the output:

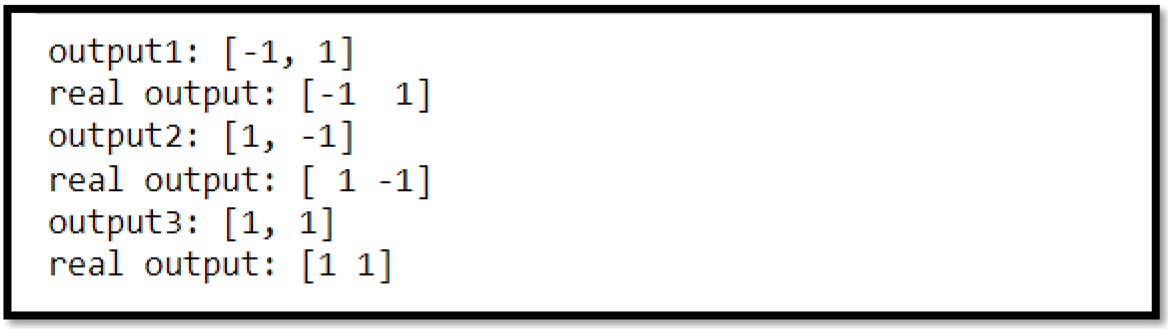
Output_1={-1,1}

Output_2={1,-1}

Output_3={1,1}

Now we have **(1*2) dimension** for the output which can lead to a weight matrix of **(63,2)** dimension.

Further you can see the predicted results which abbreviated below:



```
output1: [-1, 1]
real output: [-1  1]
output2: [1, -1]
real output: [ 1 -1]
output3: [1, 1]
real output: [1 1]
```

Figure3. Prediction Results

2.5 ADDING NOSIE

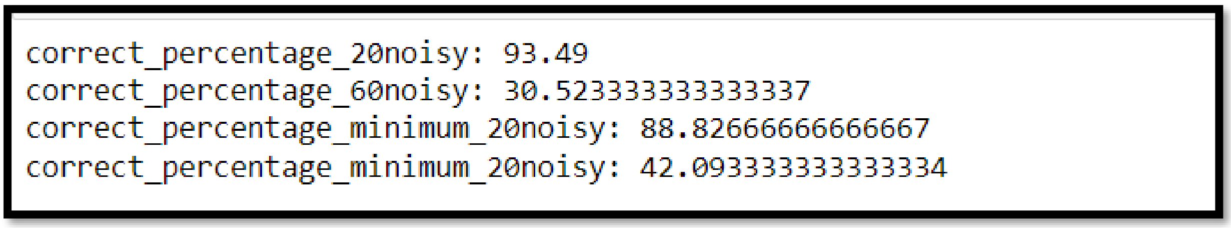
In this section, we are going to add noise to the original input patterns and investigate whether prediction is correct or not.

Procedure:

I have written a function which add noise to the input vectors and then we can predict output again and see what percent of times we capture correct output for both **20 & 60 percent noisy vector.**

Please Note that in this section I have implemented another weight matrix called w_minimum which pointed to the weight matrix of the minimum possible dimension which discussed in the previous part and then I have predicted the output patterns for both states and calculate the accuracy for each of them.

The Accuracy results:



```
correct_percentage_20noisy: 93.49
correct_percentage_60noisy: 30.523333333333337
correct_percentage_minimum_20noisy: 88.82666666666667
correct_percentage_minimum_20noisy: 42.093333333333334
```

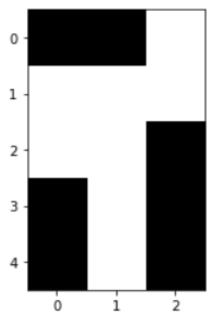
Procedure:

I have added noise to the vectors for 10000 times and then using if statement I check how many times the predicted output is equal to original-output.(for **minimum-dimension state** and **63*15 dimension state**)

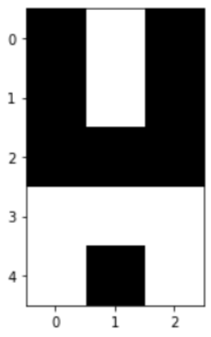
Further I have attached results of some output which could be correct or not!

weight-matrix (63*15):

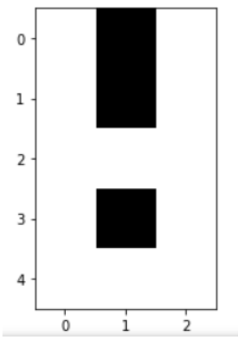
a) 20 percent noise:



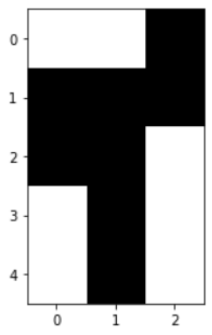
un-correct one

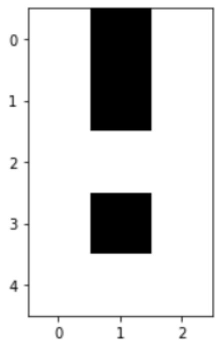


correct one



un-correct one


correct one


un-correct one

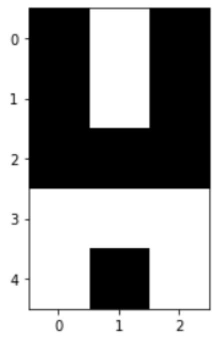
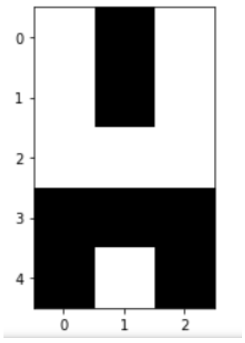
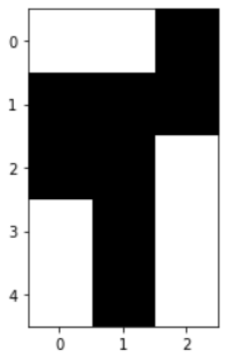
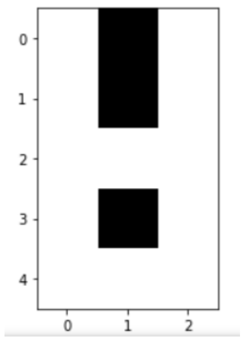

correct one

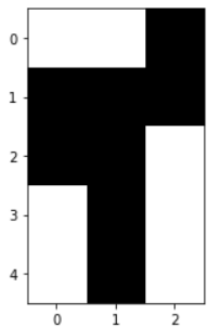
Figure4. Prediction Results for 20% noisy vector

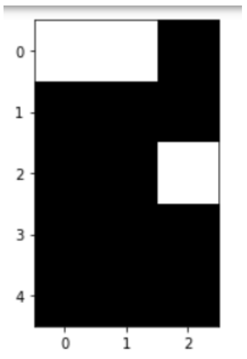
b) 60 percent noise:


un-correct one


correct one


un-correct one


correct one


un-correct one

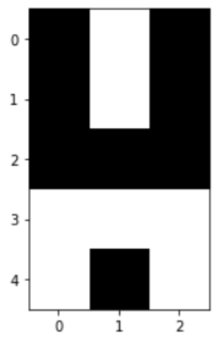

correct one

Figure5. Prediction Results for 60% noisy vector

2.6 Information Loss

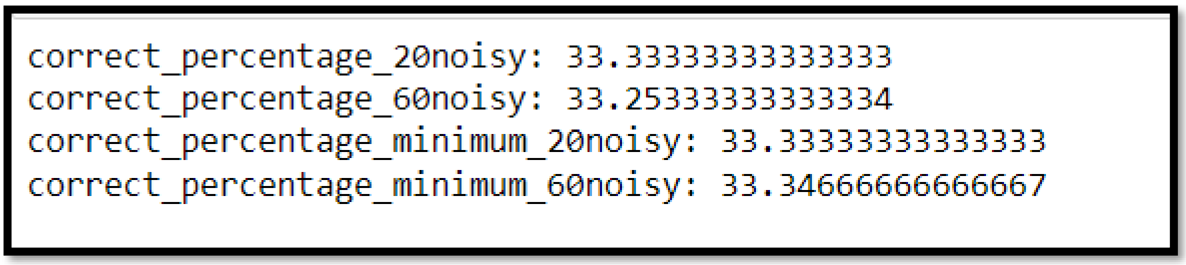
In this section, we are going to add information loss(zero padding) to the original input patterns and investigate whether prediction is correct or not.

Procedure:

I have written a function which add information loss(zero padding) to the input vectors and then we can predict output again and see what percent of times we capture correct output for both **20 & 60 percent noisy vector.**

Please Note that in this section I have implemented another weight matrix called w_minimum which pointed to the weight matrix of the minimum possible dimension which discussed in the previous part and then I have predicted the output patterns for both states and calculate the accuracy for each of them.

The Accuracy results:



```
correct_percentage_20noisy: 33.333333333333
correct_percentage_60noisy: 33.25333333333334
correct_percentage_minimum_20noisy: 33.333333333333333
correct_percentage_minimum_60noisy: 33.3466666666667
```

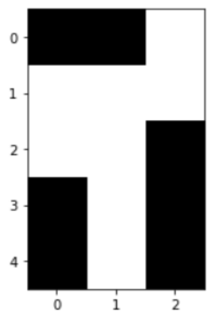
Procedure:

I have added loss to the vectors for 10000 times and then using if statement I check how many times the predicted output is equal to original-output.(for **minimum-dimension state** and **63*15 dimension state**)

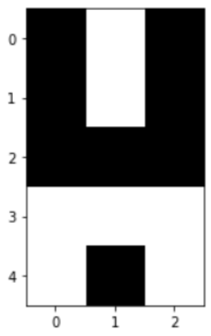
Further I have attached results of some output which could be correct or not!

weight-matrix (63*15):

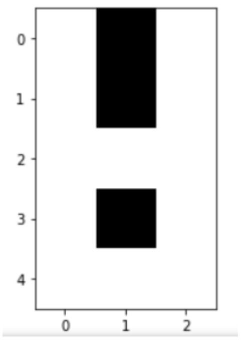
a) 20 percent information loss:



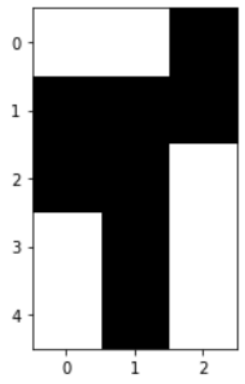
un-correct one

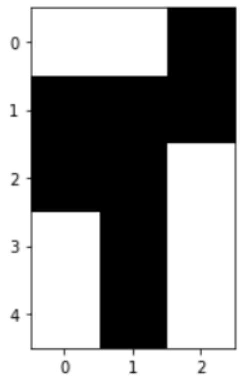


correct one



un-correct one


un-correct one


un-correct one

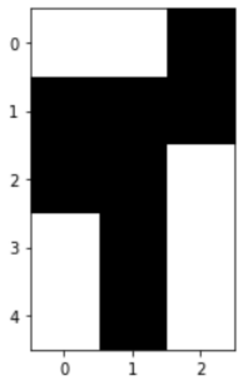
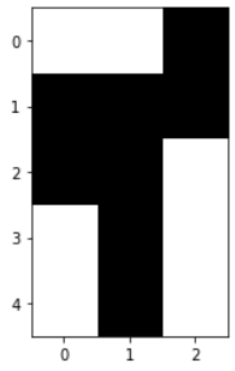

un-correct one

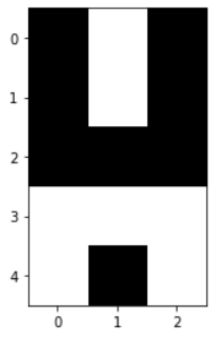
Figure6. Prediction Results for 20% information loss in vector

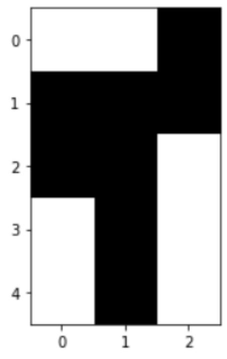
please note that above pattern predicted wrong because our predictor generates “آ " pattern instead of generating “ب” or “ن”.

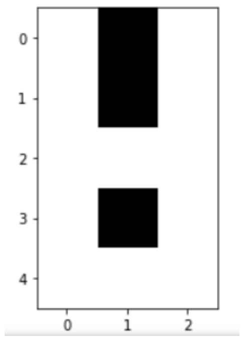
In fact our metrics is completely same pattern to count correct rate.

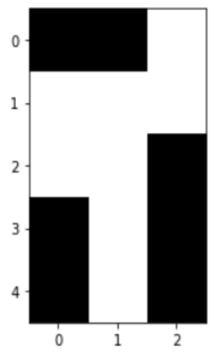
a) 60 percent information loss:


un-correct one


correct one


un-correct one


un-correct one


un-correct one

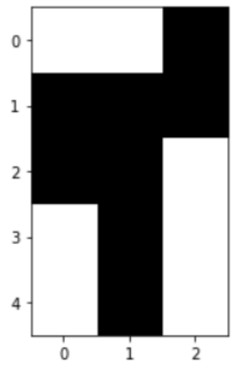

un-correct one

Figure7. Prediction Results for 60% information loss in vector

2.7 Information Loss vs Adding Noise in case of Network Stability

Conclusion:

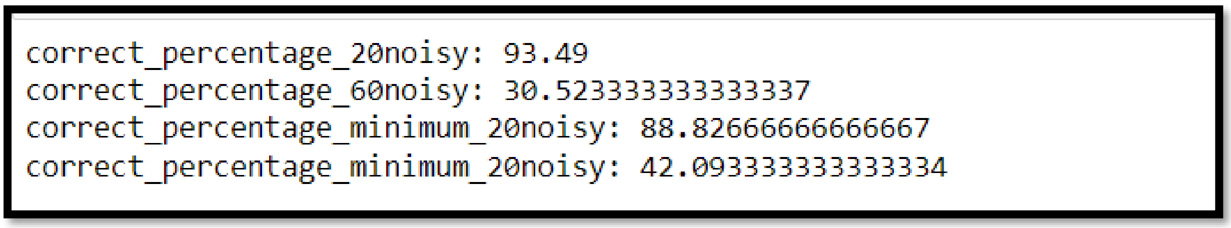
Effect of stability on the original wright matrix:

As we can see the accuracy in **case of noise adding was pretty much better** when we were dealing with predicting the results.

And we can state that **network stability in case of noise adding was higher in comparison with adding loss information.**

In fact in the same condition we get 93 percent accuracy vs 33 percent accuracy in 20 percent noisy vector and also we get 40 percent accuracy vs 33 percent accuracy in 60 percent noisy vector which show **that we get a better result when we are dealing with 20 percent noisy vector in both cases.**

Effect of stability on the minimum wright matrix(lower dimension):



```
correct_percentage_20noisy: 93.49
correct_percentage_60noisy: 30.523333333333337
correct_percentage_minimum_20noisy: 88.82666666666667
correct_percentage_minimum_20noisy: 42.093333333333334
```

If we look at the above table again we found that the accuracy for the minimum possible dimension state is lower that the original dimension which was 63*15 and then **we can conclude that we should use higher than dimension to get a better accuracy in average.**

Please note that the results conclude based on our metrics which **was equality of the output vector with the predicted vector and it's obvious that if we choose another metrics the results may tend to be different!**

3 QUESTION #2: AUTO-ASSOCIATIVE NET

In this part we intend to implement Hebbian Learning Rule in case of Pattern Auto-Association. Further, we are going to report the results of this section one by one.

3.1 WEIGHT MATRIX USING BOTH HEBBIAN RULE & MODIFIED HEBBIAN RULE

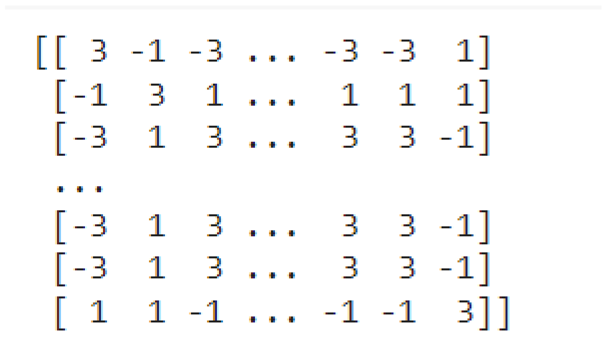
in this part we should find the weight matrix based on both approaches:

as we know the modified Hebbian is all same with the Hebbian rule with a small difference which abbreviated below:

```
W_modified_Hebbian_Rule = W_Hebbian_Rule - P*np.identity(35)
```

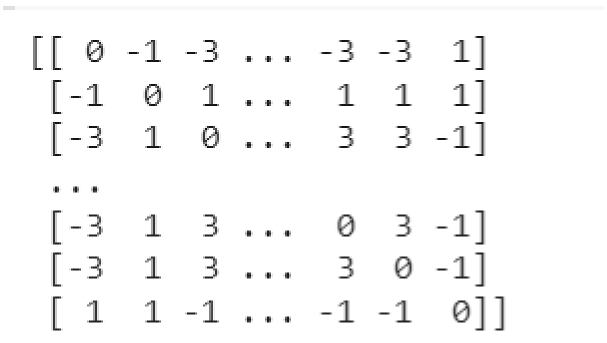
In fact, in we must subtract the Hebbian weight matrix from P * Identity matrix to obtain the Modified Hebbian Rule.

Hebbian weight matrix:



```
[[ 3 -1 -3 ... -3 -3  1]
 [-1  3  1 ...  1  1  1]
 [-3  1  3 ...  3  3 -1]
 ...
 [-3  1  3 ...  3  3 -1]
 [-3  1  3 ...  3  3 -1]
 [ 1  1 -1 ... -1 -1  3]]
```

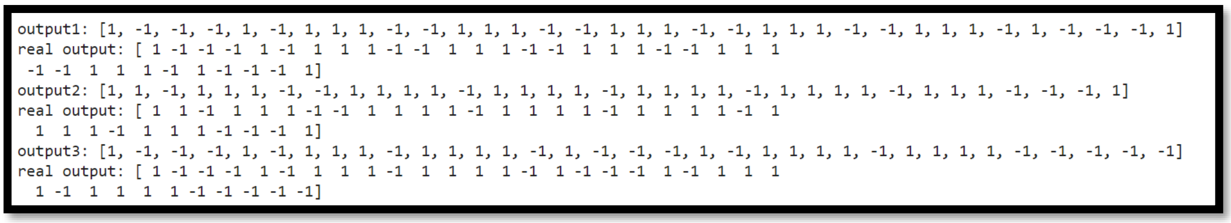
Modified Hebbian weight matrix:



```
[[ 0 -1 -3 ... -3 -3  1]
 [-1  0  1 ...  1  1  1]
 [-3  1  0 ...  3  3 -1]
 ...
 [-3  1  3 ...  0  3 -1]
 [-3  1  3 ...  3  0 -1]
 [ 1  1 -1 ... -1 -1  0]]
```

3.2 EVALUATION OF SECTION USING HEBBIAN RULE

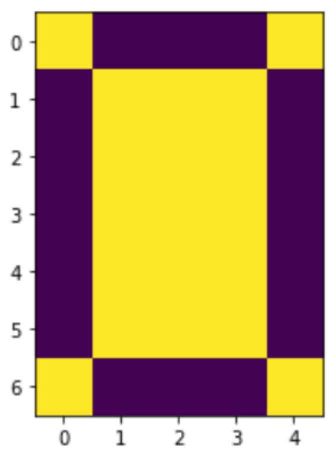
In this section, we predict the model using the test images provided in question directory and the results have been attached below:

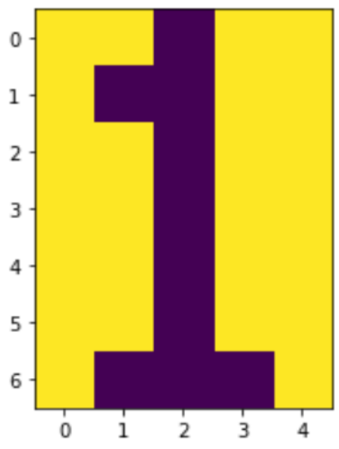


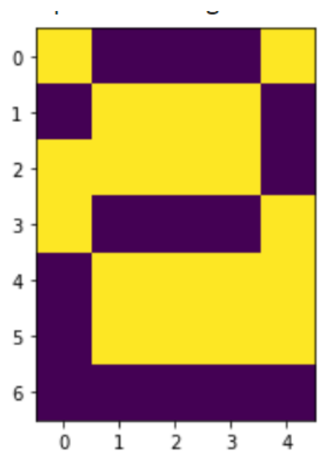
```
output1: [1, -1, -1, -1, 1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1, 1, -1, -1, -1, 1]
real output: [ 1 -1 -1 -1  1 -1  1  1  1 -1 -1  1  1  1 -1 -1  1  1  1 -1 -1  1  1  1
 -1 -1  1  1  1 -1  1 -1 -1 -1  1]
output2: [1, 1, -1, 1, 1, 1, -1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, -1, -1, -1, 1]
real output: [ 1  1 -1  1  1  1 -1 -1  1  1  1  1 -1  1  1  1  1 -1  1  1  1  1 -1  1
  1  1  1 -1  1  1  1 -1 -1 -1  1]
output3: [1, -1, -1, -1, 1, -1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, -1, -1, -1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, -1, -1, -1, -1]
real output: [ 1 -1 -1 -1  1 -1  1  1  1 -1  1  1  1  1 -1  1 -1 -1 -1  1 -1  1  1  1
  1 -1  1  1  1  1 -1 -1 -1 -1 -1]
```

Visualizing the input and output images:

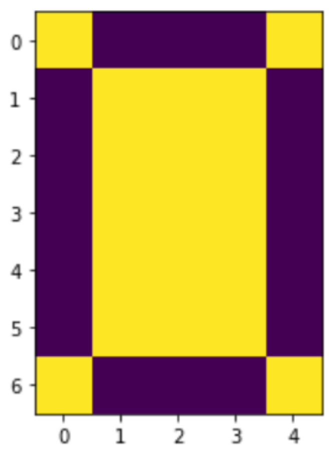
Inputs:

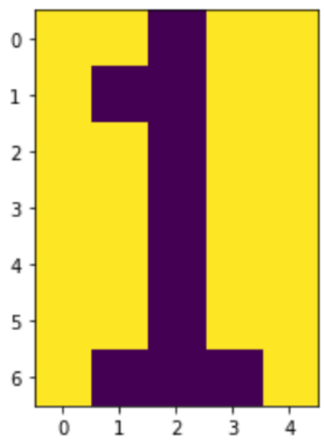






Outputs:





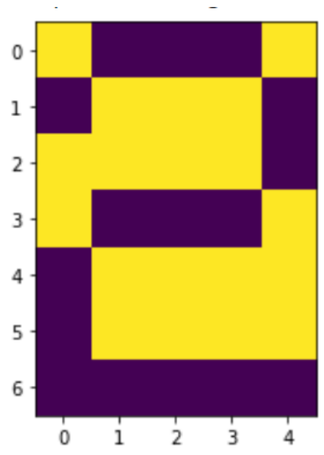


Figure8. Prediction Results

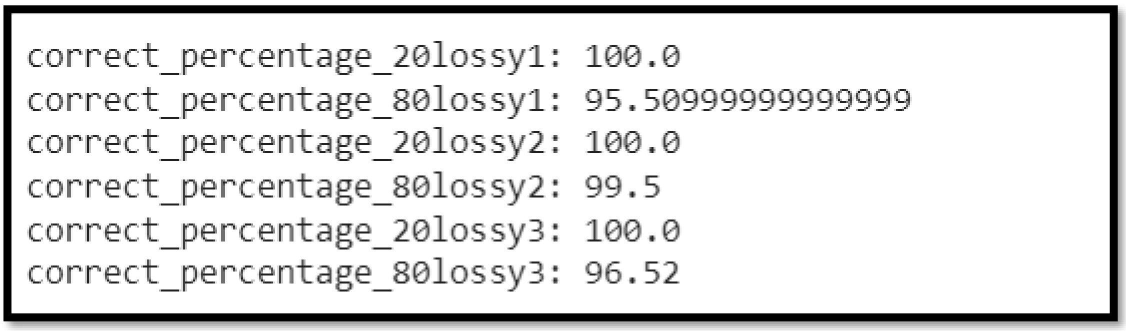
3.3 EVALUATION OF SECTION USING ADDING NOISE

In this section, we are going to add noise to the original input patterns and investigate whether prediction is correct or not.

Procedure:

I have written a function which add noise to the input vectors and then we can predict output again and see what percent of times we capture correct output for both **20 & 80 percent noisy vector.**

The Accuracy results:

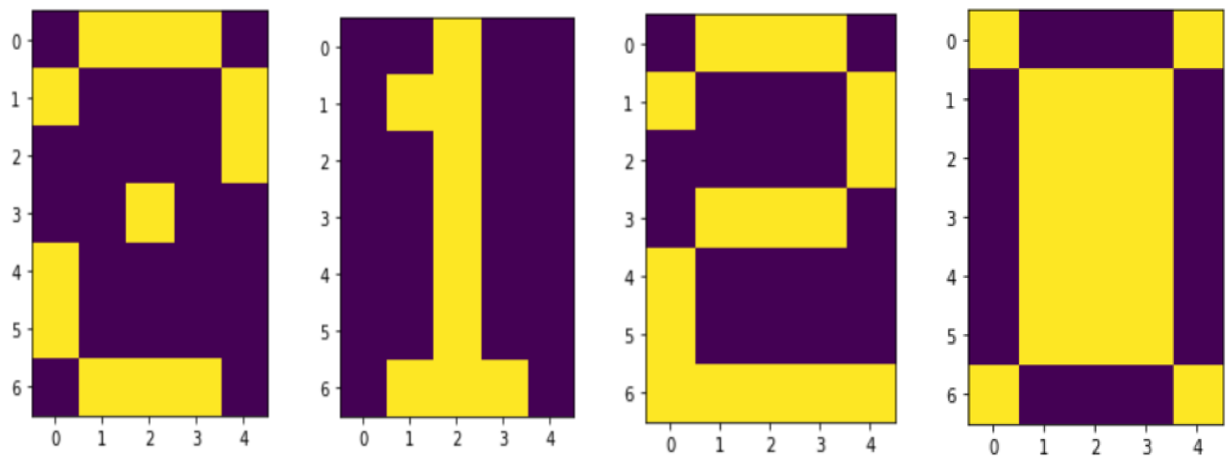


```
correct_percentage_20lossy1: 100.0
correct_percentage_80lossy1: 95.50999999999999
correct_percentage_20lossy2: 100.0
correct_percentage_80lossy2: 99.5
correct_percentage_20lossy3: 100.0
correct_percentage_80lossy3: 96.52
```

Procedure:

I have added loss to the vectors for 10000 times and then using if statement I check how many times the predicted output is equal to original-output.(for **each of input-pattern**)

Further I have attached results of some output which could be correct or not!



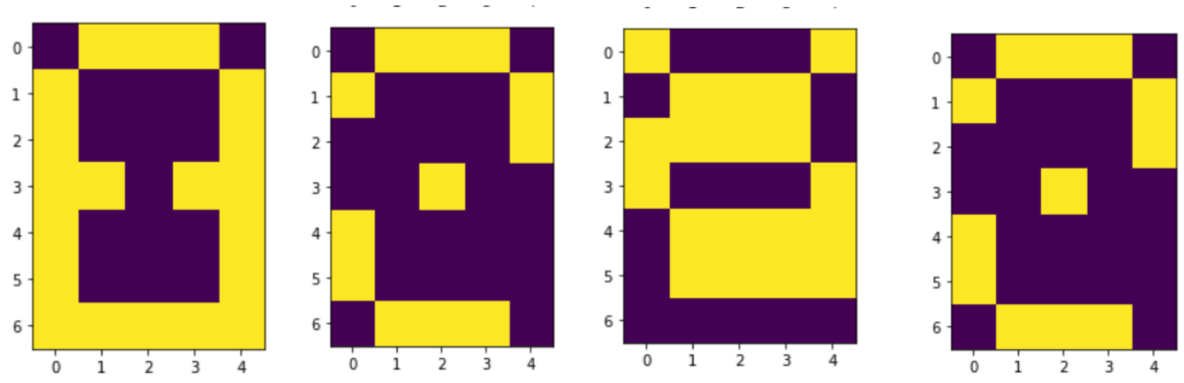
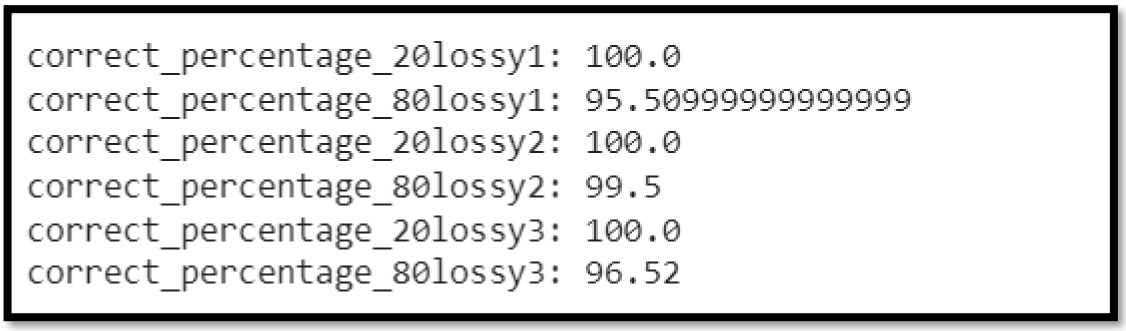


Figure8. some of Prediction Results for both 20% and 80% noisy vector

What is the most sensitive pattern-number to the noise?



```
correct_percentage_20lossy1: 100.0
correct_percentage_80lossy1: 95.50999999999999
correct_percentage_20lossy2: 100.0
correct_percentage_80lossy2: 99.5
correct_percentage_20lossy3: 100.0
correct_percentage_80lossy3: 96.52
```

The most sensitive pattern-number to the noise is the pattern which has minimum accuracy value when we add noise to the input-patterns.

And Then the most sensitive pattern to the noise is "2" pattern which has the minimum accuracy-value.

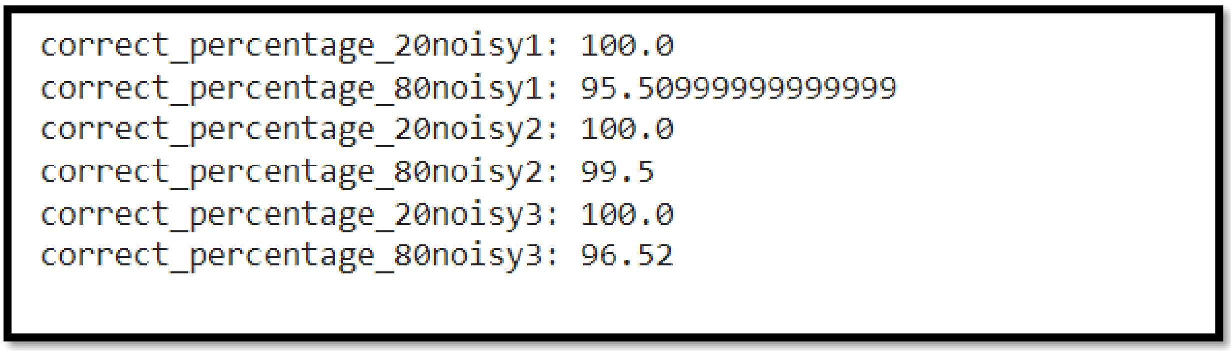
3.4 EVALUATION OF SECTION USING ADDING INFORMATION LOSS

In this section, we are going to add loss to the original input patterns and investigate whether prediction is correct or not.

Procedure:

I have written a function which add loss to the input vectors and then we can predict output again and see what percent of times we capture correct output for both **20 & 80 percent lossy vector.**

The Accuracy results:

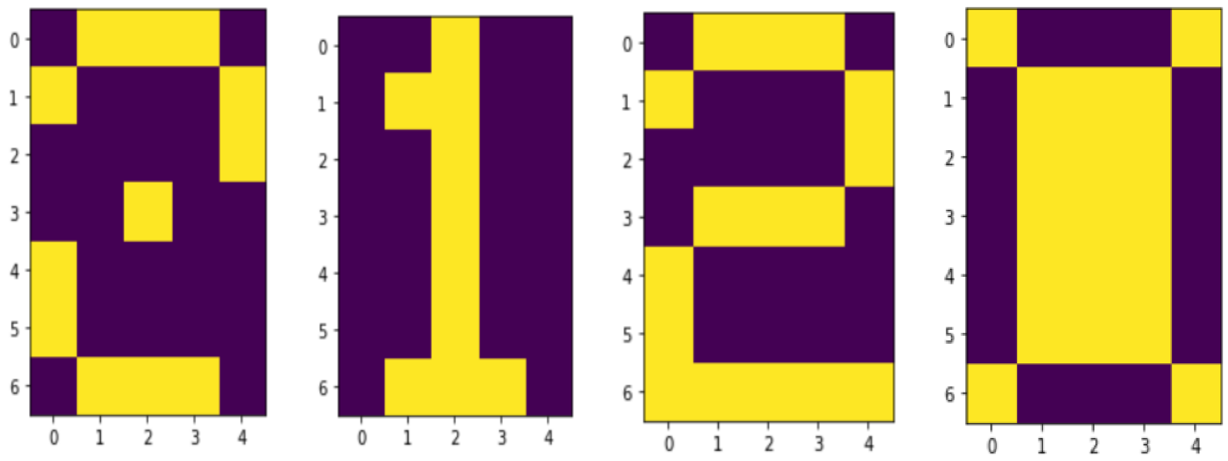


```
correct_percentage_20noisy1: 100.0
correct_percentage_80noisy1: 95.50999999999999
correct_percentage_20noisy2: 100.0
correct_percentage_80noisy2: 99.5
correct_percentage_20noisy3: 100.0
correct_percentage_80noisy3: 96.52
```

Procedure:

I have added loss to the vectors for 10000 times and then using if statement I check how many times the predicted output is equal to original-output.(for **each of input-pattern**)

Further I have attached results of some output which could be correct or not!



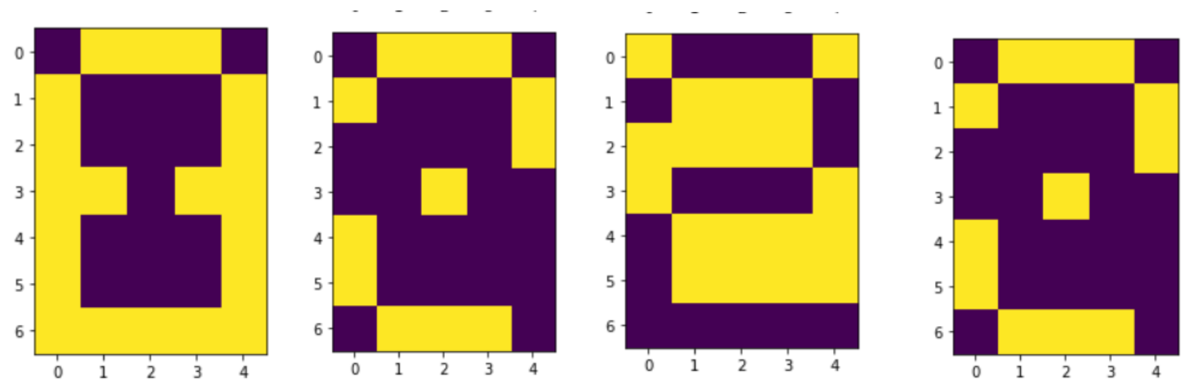


Figure9. some of Prediction Results for both 20% and 80% lossy vector

What is the most sensitive pattern-number to the loss?

```
correct_percentage_20noisy1: 100.0
correct_percentage_80noisy1: 95.50999999999999
correct_percentage_20noisy2: 100.0
correct_percentage_80noisy2: 99.5
correct_percentage_20noisy3: 100.0
correct_percentage_80noisy3: 96.52
```

The most sensitive pattern-number to the loss is the pattern which has minimum accuracy value when we add loss to the input-patterns.

And Then as you can see we are dealing with even state and on-one has a priority to the another-one and then in this case we have no most-sensitive bit in comparison to others!

Please note that the results conclude based on our metrics which **was equality of the output vector with the predicted vector and it's obvious that if we choose another metrics the results may tend to be different!**

3.5 BONUS SECTION

In linear algebra pseudoinverse (A^{+}) of a matrix A is a generalization of the inverse matrix. The most common use of pseudoinverse is to compute the best fit solution to a system of linear equations which lacks a unique solution. Moore – Penrose inverse is the most widely known type of matrix pseudoinverse. The term generalized inverse is sometimes used as a synonym of pseudoinverse.

Let the system is given as:

$\overrightarrow{y} = A\overrightarrow{x}$

We know **A** and $\overrightarrow{y}$, and we want to find $\overrightarrow{x}$.
Where:
$\overrightarrow{x}$ and $\overrightarrow{y}$ are vectors,

A is a matrix If A is a square matrix, we proceed as below:

$$\overrightarrow{y} = A\overrightarrow{x}$$

$$A^{-1}\overrightarrow{y} = A^{-1}A\overrightarrow{x}$$

$$A^{-1}\overrightarrow{y} = I\overrightarrow{x}$$

$$A^{-1}\overrightarrow{y} = \overrightarrow{x}$$

But if A is not a square matrix, we cannot compute the usual A^{-1}. Nevertheless, we can form the pseudoinverse.

$A^{+} \equiv pseudoinverse$

If we go in detail at the system i.e. $\overrightarrow{y} = A\overrightarrow{x}$, then it represents the following set of equations:

$a_{1,1}x_1 + a_{1,2}x_2 + ... + a_{1,n}x_n = y_1$

$a_{2,1}x_1 + a_{2,2}x_2 + ... + a_{2,n}x_n = y_2$

. .

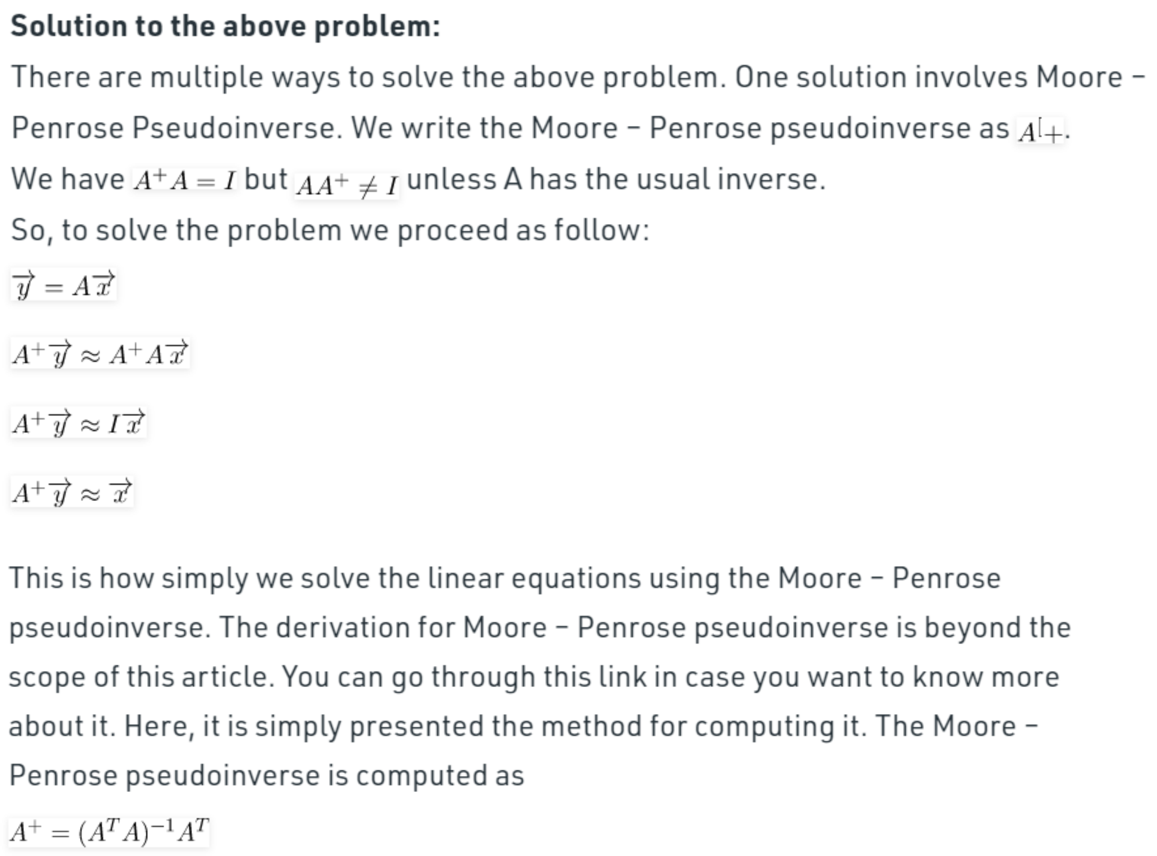
. .

$a_{m,1}x_1 + a_{m,2}x_2 + ... + a_{m,n}x_n = y_m$

which can also be written in matrix form as below:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & . & . & . & a_{1,n} \\ a_{2,1} & a_{2,2} & . & . & . & a_{2,n} \\ . & & & & & . \\ . & & & & & . \\ . & & & & & . \\ a_{m,1} & a_{m,2} & . & . & . & a_{m,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ . \\ y_n \end{bmatrix}$$

Where m > n which means the number of rows is greater than the number of columns or the number of rows is greater than the number of variables.



Solution to the above problem:

There are multiple ways to solve the above problem. One solution involves Moore – Penrose Pseudoinverse. We write the Moore – Penrose pseudoinverse as A^{+}.
We have $A^+A = I$ but $AA^+ \neq I$ unless A has the usual inverse.
So, to solve the problem we proceed as follow:

$$\vec{y} = A\vec{x}$$

$$A^+\vec{y} \approx A^+A\vec{x}$$

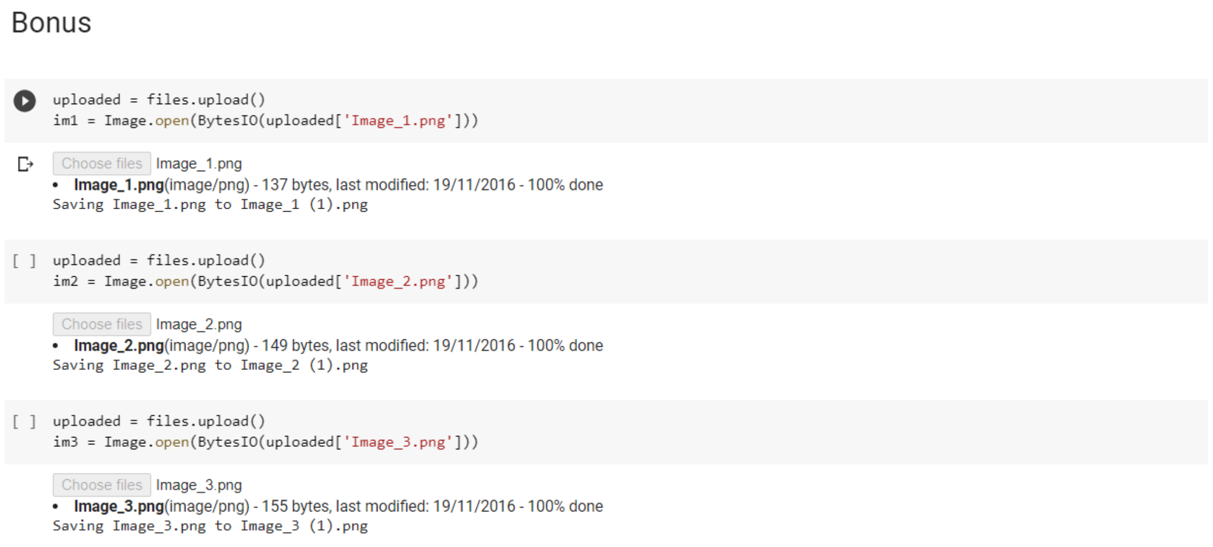
$$A^+\vec{y} \approx I\vec{x}$$

$$A^+\vec{y} \approx \vec{x}$$

This is how simply we solve the linear equations using the Moore – Penrose pseudoinverse. The derivation for Moore – Penrose pseudoinverse is beyond the scope of this article. You can go through this link in case you want to know more about it. Here, it is simply presented the method for computing it. The Moore – Penrose pseudoinverse is computed as

$$A^+ = (A^TA)^{-1}A^T$$

Further the codes of this part have been attached :



Bonus

```
uploaded = files.upload()
im1 = Image.open(BytesIO(uploaded['Image_1.png']))
```

Choose files Image_1.png
- **Image_1.png**(image/png) - 137 bytes, last modified: 19/11/2016 - 100% done

```
Saving Image_1.png to Image_1 (1).png
```

```
[ ] uploaded = files.upload()
im2 = Image.open(BytesIO(uploaded['Image_2.png']))
```

Choose files Image_2.png
- **Image_2.png**(image/png) - 149 bytes, last modified: 19/11/2016 - 100% done

```
Saving Image_2.png to Image_2 (1).png
```

```
[ ] uploaded = files.upload()
im3 = Image.open(BytesIO(uploaded['Image_3.png']))
```

Choose files Image_3.png
- **Image_3.png**(image/png) - 155 bytes, last modified: 19/11/2016 - 100% done

```
Saving Image_3.png to Image_3 (1).png
```

```
uploaded = files.upload()
im4 = Image.open(BytesIO(uploaded['Image_4.png']))
```

Choose files Image_4.png
- **Image_4.png**(image/png) - 157 bytes, last modified: 19/11/2016 - 100% done

```
Saving Image_4.png to Image_4.png
```

```
uploaded = files.upload()
im5 = Image.open(BytesIO(uploaded['Image_5.png']))
```

Choose files Image_5.png
- **Image_5.png**(image/png) - 158 bytes, last modified: 19/11/2016 - 100% done

```
Saving Image_5.png to Image_5.png
```

```
uploaded = files.upload()
im6 = Image.open(BytesIO(uploaded['Image_6.png']))
```

Choose files Image_6.png
- **Image_6.png**(image/png) - 158 bytes, last modified: 19/11/2016 - 100% done

```
Saving Image_6.png to Image_6.png
```

```
uploaded = files.upload()
im7 = Image.open(BytesIO(uploaded['Image_7.png']))
```

Choose files Image_7.png
- **Image_7.png**(image/png) - 145 bytes, last modified: 19/11/2016 - 100% done

```
Saving Image_7.png to Image_7.png
```

```
uploaded = files.upload()
im8 = Image.open(BytesIO(uploaded['Image_8.png']))
```

Choose files Image_8.png
- **Image_8.png**(image/png) - 160 bytes, last modified: 19/11/2016 - 100% done

```
Saving Image_8.png to Image_8.png
```

```
uploaded = files.upload()
im9 = Image.open(BytesIO(uploaded['Image_9.png']))
```

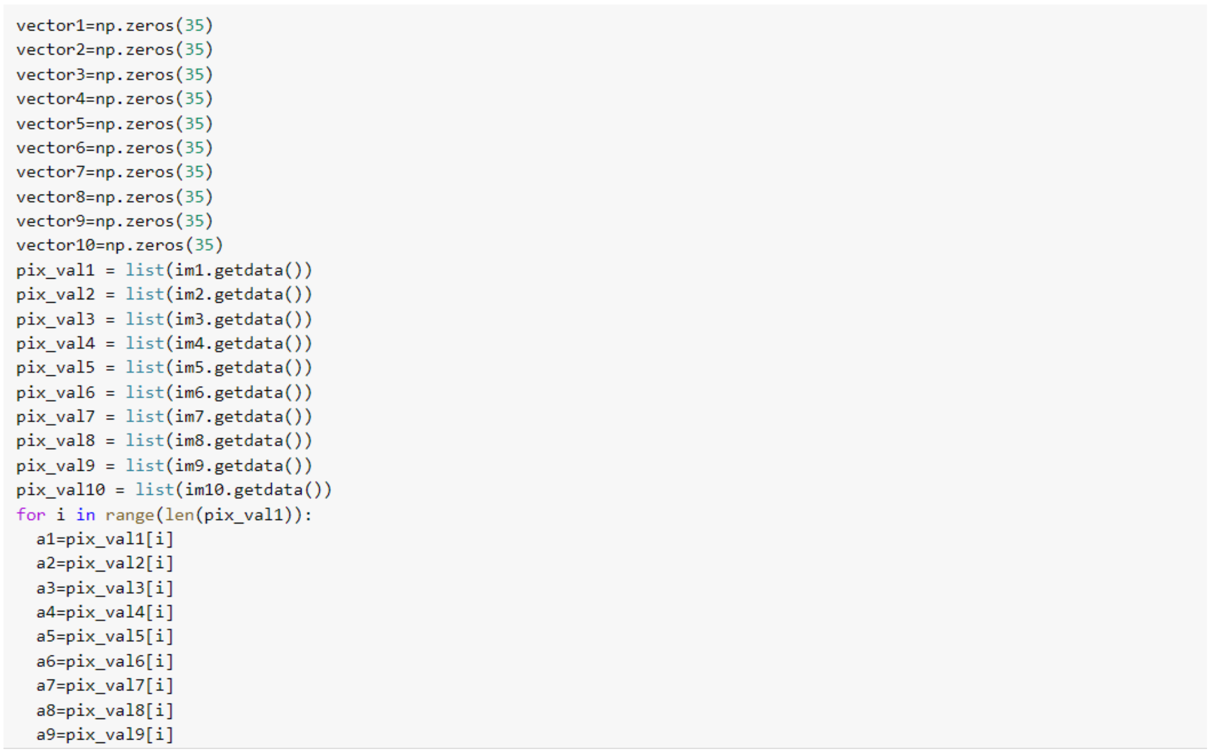
Choose files Image_9.png
- **Image_9.png**(image/png) - 140 bytes, last modified: 19/11/2016 - 100% done

```
Saving Image_9.png to Image_9.png
```

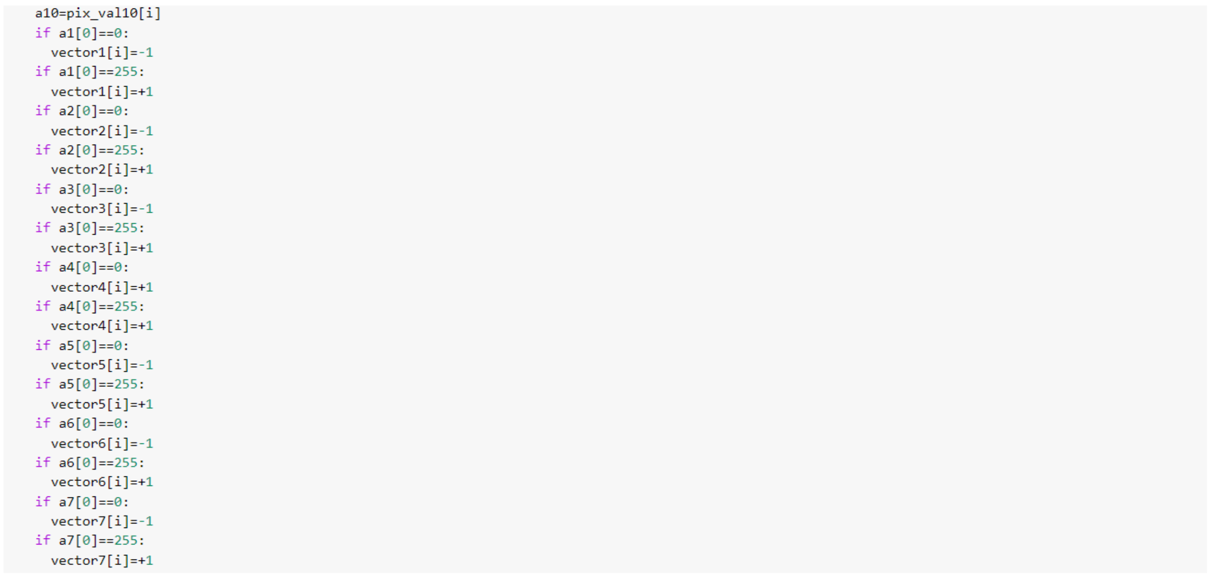
```
uploaded = files.upload()
im10 = Image.open(BytesIO(uploaded['Image_10.png']))
```

Choose files Image_10.png
- **Image_10.png**(image/png) - 155 bytes, last modified: 19/11/2016 - 100% done

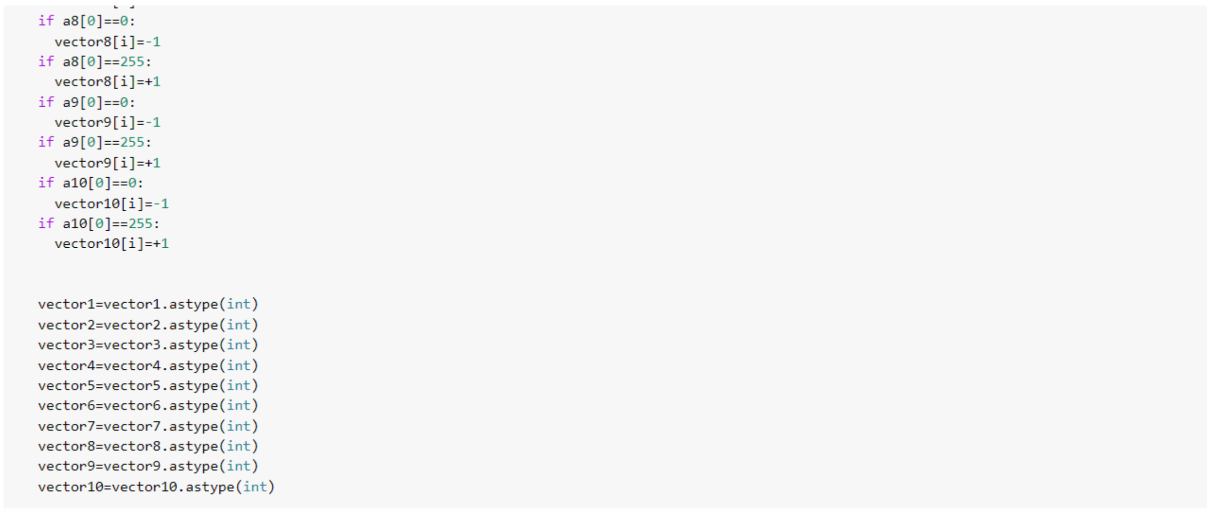
```
Saving Image_10.png to Image_10.png
```



```
vector1=np.zeros(35)
vector2=np.zeros(35)
vector3=np.zeros(35)
vector4=np.zeros(35)
vector5=np.zeros(35)
vector6=np.zeros(35)
vector7=np.zeros(35)
vector8=np.zeros(35)
vector9=np.zeros(35)
vector10=np.zeros(35)
pix_val1 = list(im1.getdata())
pix_val2 = list(im2.getdata())
pix_val3 = list(im3.getdata())
pix_val4 = list(im4.getdata())
pix_val5 = list(im5.getdata())
pix_val6 = list(im6.getdata())
pix_val7 = list(im7.getdata())
pix_val8 = list(im8.getdata())
pix_val9 = list(im9.getdata())
pix_val10 = list(im10.getdata())
for i in range(len(pix_val1)):
  a1=pix_val1[i]
  a2=pix_val2[i]
  a3=pix_val3[i]
  a4=pix_val4[i]
  a5=pix_val5[i]
  a6=pix_val6[i]
  a7=pix_val7[i]
  a8=pix_val8[i]
  a9=pix_val9[i]
```

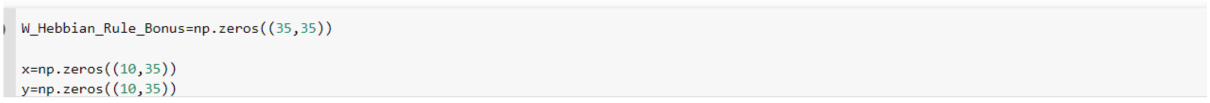


```
  a10=pix_val10[i]
  if a1[0]==0:
    vector1[i]=-1
  if a1[0]==255:
    vector1[i]=+1
  if a2[0]==0:
    vector2[i]=-1
  if a2[0]==255:
    vector2[i]=+1
  if a3[0]==0:
    vector3[i]=-1
  if a3[0]==255:
    vector3[i]=+1
  if a4[0]==0:
    vector4[i]=+1
  if a4[0]==255:
    vector4[i]=+1
  if a5[0]==0:
    vector5[i]=-1
  if a5[0]==255:
    vector5[i]=+1
  if a6[0]==0:
    vector6[i]=-1
  if a6[0]==255:
    vector6[i]=+1
  if a7[0]==0:
    vector7[i]=-1
  if a7[0]==255:
    vector7[i]=+1
```



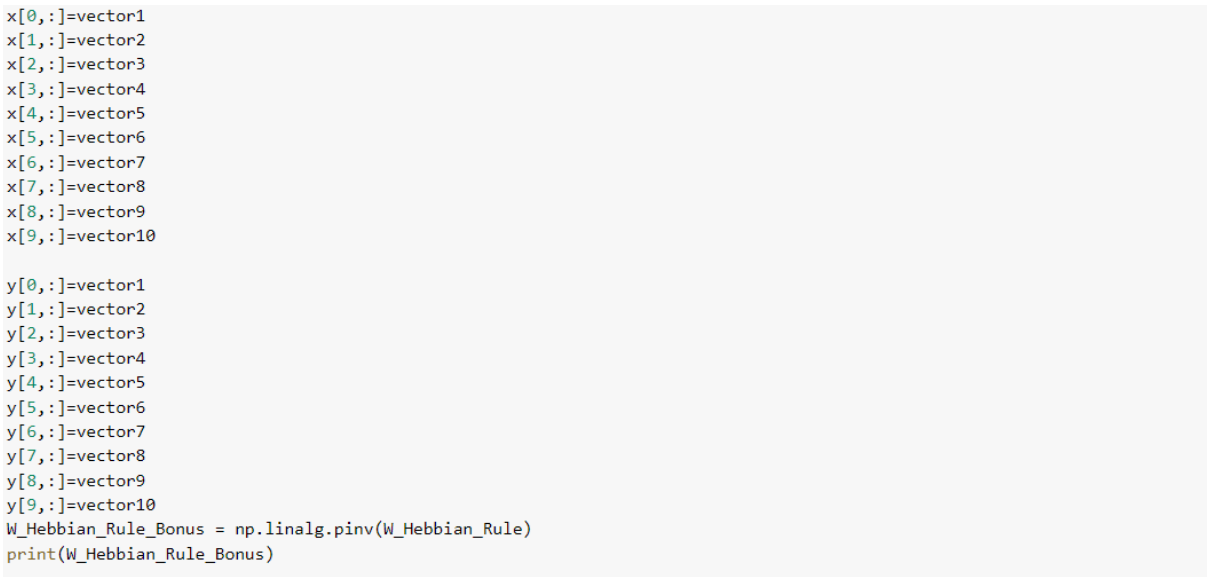
```
    if a8[0]==0:
      vector8[i]=-1
    if a8[0]==255:
      vector8[i]=+1
    if a9[0]==0:
      vector9[i]=-1
    if a9[0]==255:
      vector9[i]=+1
    if a10[0]==0:
      vector10[i]=-1
    if a10[0]==255:
      vector10[i]=+1


    vector1=vector1.astype(int)
    vector2=vector2.astype(int)
    vector3=vector3.astype(int)
    vector4=vector4.astype(int)
    vector5=vector5.astype(int)
    vector6=vector6.astype(int)
    vector7=vector7.astype(int)
    vector8=vector8.astype(int)
    vector9=vector9.astype(int)
    vector10=vector10.astype(int)
```



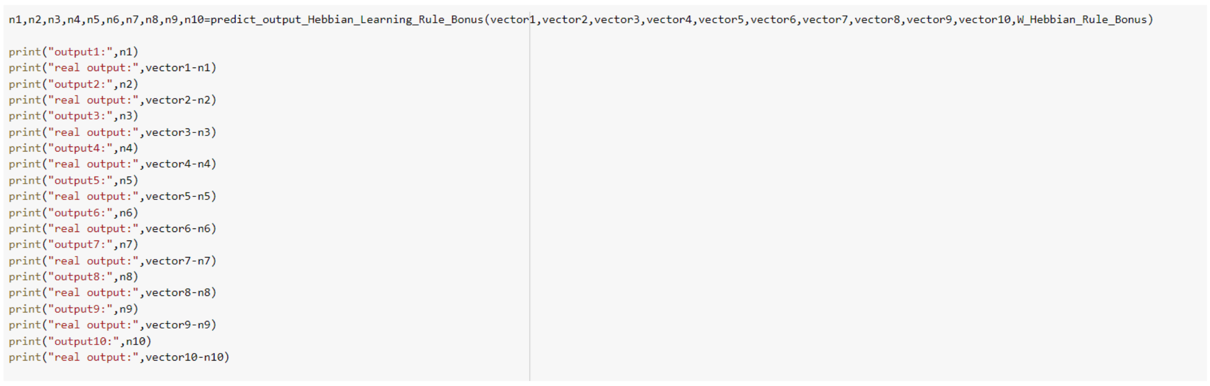
```
W_Hebbian_Rule_Bonus=np.zeros((35,35))

x=np.zeros((10,35))
y=np.zeros((10,35))
```



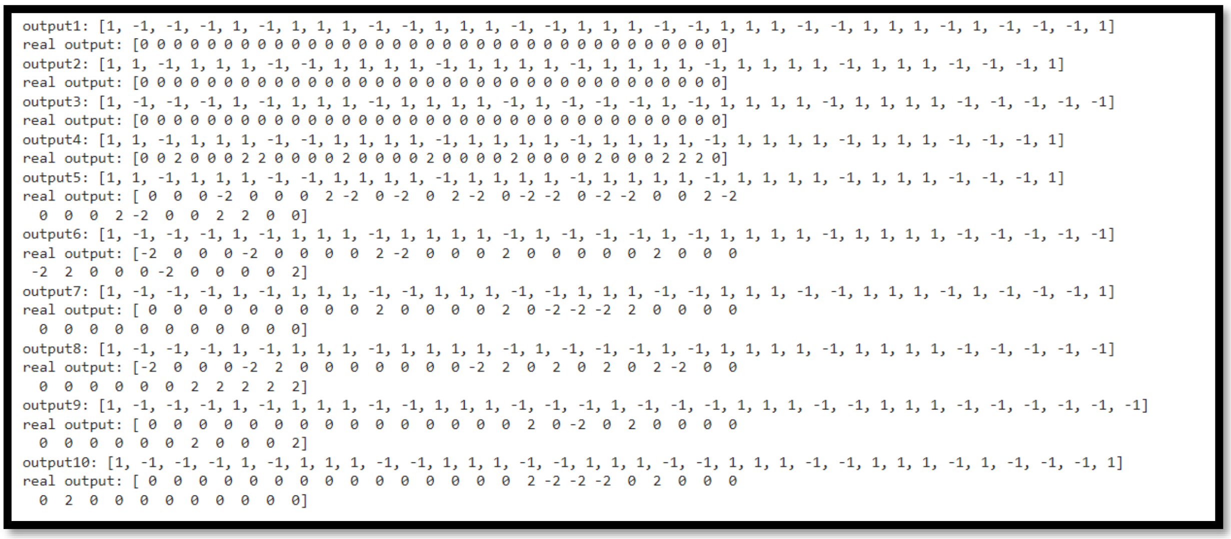
```
x[0,:]=vector1
x[1,:]=vector2
x[2,:]=vector3
x[3,:]=vector4
x[4,:]=vector5
x[5,:]=vector6
x[6,:]=vector7
x[7,:]=vector8
x[8,:]=vector9
x[9,:]=vector10

y[0,:]=vector1
y[1,:]=vector2
y[2,:]=vector3
y[3,:]=vector4
y[4,:]=vector5
y[5,:]=vector6
y[6,:]=vector7
y[7,:]=vector8
y[8,:]=vector9
y[9,:]=vector10
W_Hebbian_Rule_Bonus = np.linalg.pinv(W_Hebbian_Rule)
print(W_Hebbian_Rule_Bonus)
```



```
n1,n2,n3,n4,n5,n6,n7,n8,n9,n10=predict_output_Hebbian_Learning_Rule_Bonus(vector1,vector2,vector3,vector4,vector5,vector6,vector7,vector8,vector9,vector10,W_Hebbian_Rule_Bonus)

print("output1:",n1)
print("real output:",vector1-n1)
print("output2:",n2)
print("real output:",vector2-n2)
print("output3:",n3)
print("real output:",vector3-n3)
print("output4:",n4)
print("real output:",vector4-n4)
print("output5:",n5)
print("real output:",vector5-n5)
print("output6:",n6)
print("real output:",vector6-n6)
print("output7:",n7)
print("real output:",vector7-n7)
print("output8:",n8)
print("real output:",vector8-n8)
print("output9:",n9)
print("real output:",vector9-n9)
print("output10:",n10)
print("real output:",vector10-n10)
```



```
output1: [1, -1, -1, -1, 1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1, 1, -1, -1, -1, 1]
real output: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
output2: [1, 1, -1, 1, 1, 1, -1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, -1, -1, -1, 1]
real output: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
output3: [1, -1, -1, -1, 1, -1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, -1, -1, -1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, -1, -1, -1, -1]
real output: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
output4: [1, 1, -1, 1, 1, 1, -1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, -1, -1, -1, 1]
real output: [0 0 2 0 0 0 2 2 0 0 0 0 2 0 0 0 0 2 0 0 0 0 2 0 0 0 0 2 0 0 0 2 2 2 0]
output5: [1, 1, -1, 1, 1, 1, -1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, -1, -1, -1, 1]
real output: [ 0  0  0 -2  0  0  0  2 -2  0 -2  0  2 -2  0 -2 -2  0 -2 -2  0  0  2 -2
  0  0  0  2 -2  0  0  2  2  0  0]
output6: [1, -1, -1, -1, 1, -1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, -1, -1, -1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, -1, -1, -1, -1]
real output: [-2  0  0  0 -2  0  0  0  0  2 -2  0  0  0  2  0  0  0  0  0  2  0  0  0
 -2  2  0  0  0 -2  0  0  0  0  2]
output7: [1, -1, -1, -1, 1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1, 1, -1, -1, -1, 1]
real output: [ 0  0  0  0  0  0  0  0  0  2  0  0  0  0  2  0 -2 -2 -2  2  0  0  0  0
  0  0  0  0  0  0  0  0  0  0  0]
output8: [1, -1, -1, -1, 1, -1, 1, 1, 1, -1, 1, 1, 1, 1, -1, 1, -1, -1, -1, 1, -1, 1, 1, 1, 1, -1, 1, 1, 1, 1, -1, -1, -1, -1, -1]
real output: [-2  0  0  0 -2  2  0  0  0  0  0  0  0 -2  2  0  2  0  2  0  2 -2  0  0
  0  0  0  0  0  0  2  2  2  2  2]
output9: [1, -1, -1, -1, 1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1, -1, -1, 1, -1, -1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1, -1, -1, -1, -1, -1]
real output: [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  2  0 -2  0  2  0  0  0  0
  0  0  0  0  0  0  2  0  0  0  2]
output10: [1, -1, -1, -1, 1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1, -1, 1, 1, 1, -1, 1, -1, -1, -1, 1]
real output: [ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  2 -2 -2 -2  0  2  0  0  0
  0  2  0  0  0  0  0  0  0  0  0]
```

Figure9. The original-output vs predicted-output

4 QUESTION #3: DISCRETE HOPFIELD NETWORK

In this part we intend to implement Discrete Hopfield Network in case of Pattern Association.

Further, we are going to report the results of this section one by one:

4.1 DISCRETE HOPFIELD NETWORK DESCRIPTION

It is a fully interconnected neural network where each unit is connected to every other unit. It behaves in a discrete manner, i.e., it gives finite distinct output, generally of two types:

Binary (0/1)

Bipolar (-1/1)

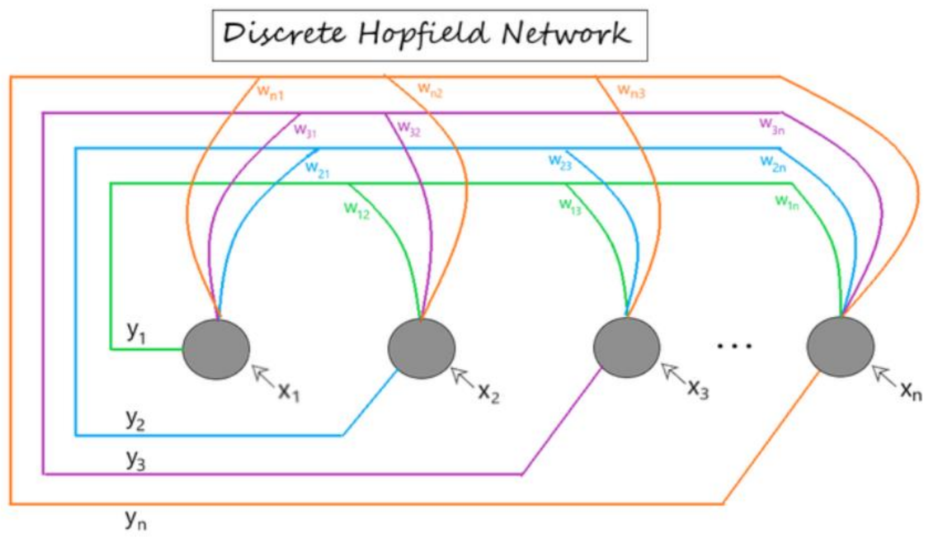
The weights associated with this network is symmetric in nature and has the following properties.

$$1.\ w_{ij} = w_{ji}$$
$$2.\ w_{ii} = 0$$

Structure & Architecture

- Each neuron has an inverting and a non-inverting output.
- Being fully connected, the output of each neuron is an input to all other neurons but not self.

Fig ? shows a sample representation of a Discrete Hopfield Neural Network architecture having the following elements.



Figure?. Sample Representation of a Discrete Hopfield

Training Algorithm:

Step 1 - Initialize weights $\mathbf{w}_{ij}$ to store patterns **(using training algorithm)**.

Step 2 - For each input vector y_i, perform **steps 3-7**.

Step 3 - Make initial activators of the network equal to the external input vector x.

$$y_i = x_i : (for\ i = 1\ to\ n)$$

Step 4 - For each vector y_i, perform **steps 5-7**.

Step 5 - Calculate the total input of the network $\mathbf{y}_{in}$ using the equation given below.

$$y_{in_i} = x_i + \sum_j [y_j w_{ji}]$$

Step 6 - Apply activation over the total input to calculate the output as per the equation given below:

$$y_i = \begin{cases} 1 \text{ if } y_{in} > \theta_i \\ y_i \text{ if } y_{in} = \theta_i \\ 0 \text{ if } y_{in} < \theta_i \end{cases}$$

(where $\boldsymbol{\theta}_i$ (threshold) and is normally taken as 0)

Step 7 - Now feedback the obtained output y_i to all other units. Thus, the activation vectors are updated.

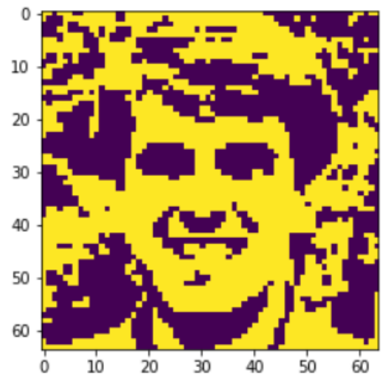
Step 8 - Test the network for convergence.

4.2 Pre-processing Stage

After the pre-processing and converting to the bi-polar the train and test images have been attached:

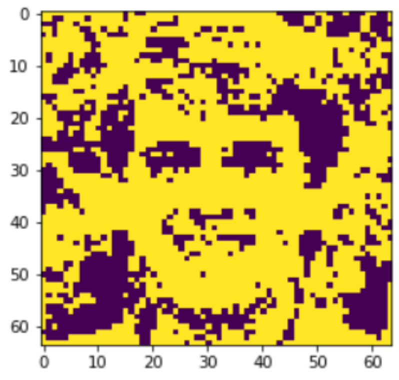
Please note that in this part I have used different threshold and at the final I have used the **average value** of the grayscale train picture.

```
<matplotlib.image.AxesImage at 0x7fd1b7e50ad0>
```

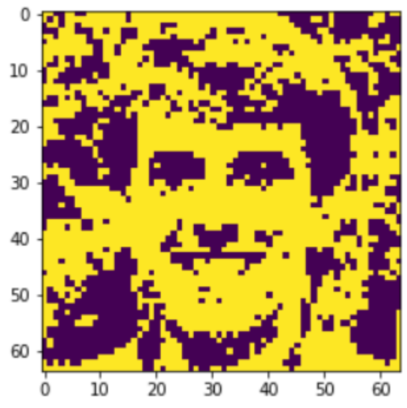


train_image

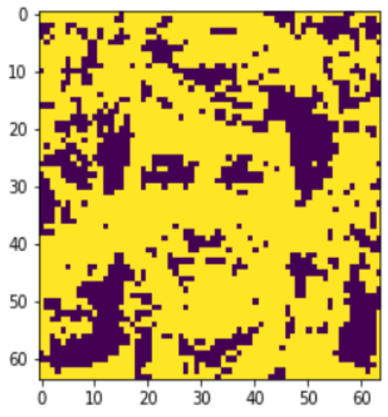
```
<matplotlib.image.AxesImage at 0x7fa9b9d71d90>
```



test1_image



test2_image



test3_image

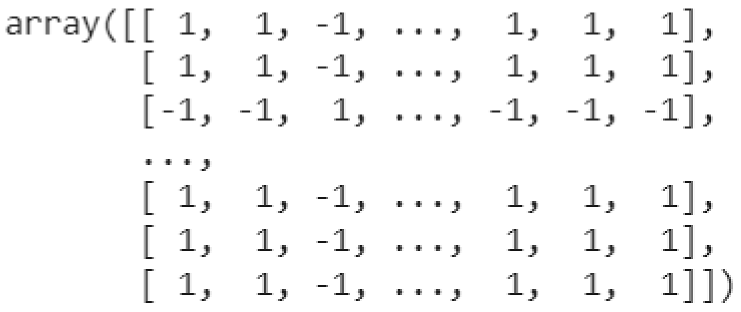
4.3 MAKING WEIGHT MATRIC

In this section we are going to make weight matrix in the Discrete Hopfield Network algorithm:

According to algorithm steps we should do :

```
W_Discrete_Hopfield_Network=product(x_train[0],x_train[0])
```

So, we obtain the weight matrix which is a (4096,4096) dimension matrix:

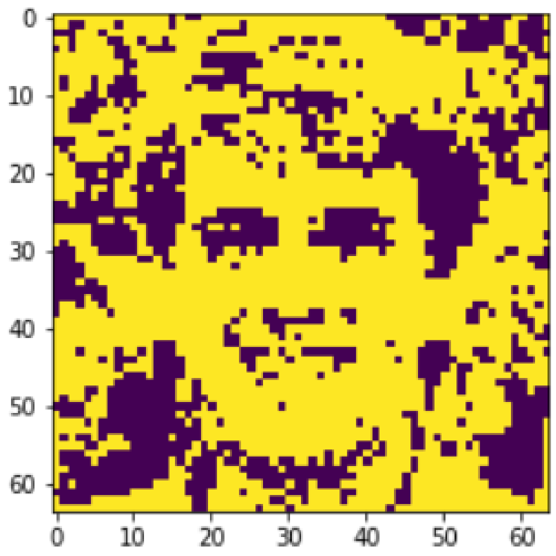


```
array([[ 1,  1, -1, ...,  1,  1,  1],
       [ 1,  1, -1, ...,  1,  1,  1],
       [-1, -1,  1, ..., -1, -1, -1],
       ...,
       [ 1,  1, -1, ...,  1,  1,  1],
       [ 1,  1, -1, ...,  1,  1,  1],
       [ 1,  1, -1, ...,  1,  1,  1]])
```

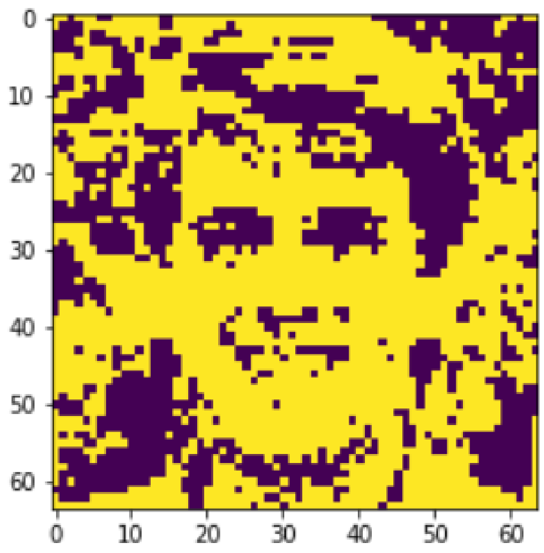
4.4 EVALUATION OF ALGORITHM USING TEST IMAGES

In this section we must eliminate the noise from our test images to found out that our algorithm works properly.

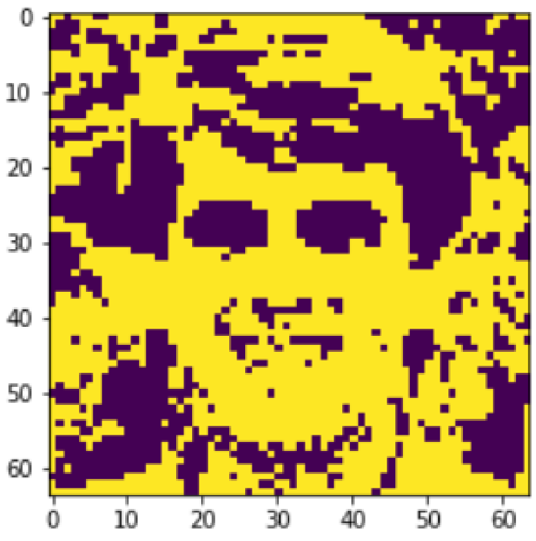
Test-image1:

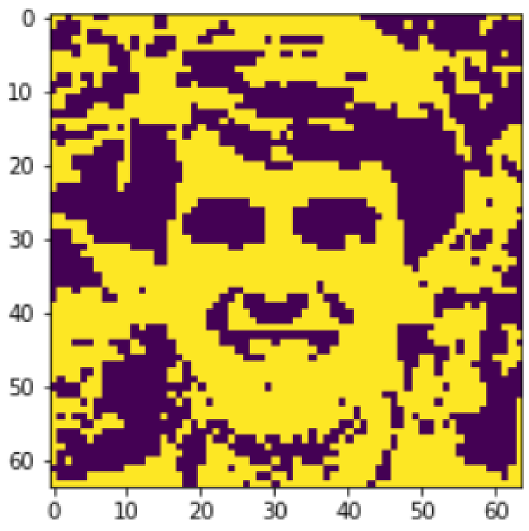


1.



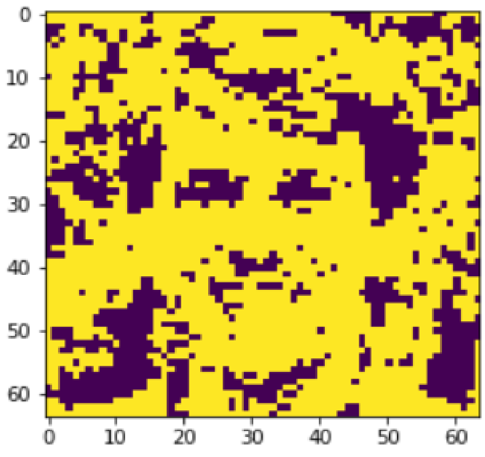
2.



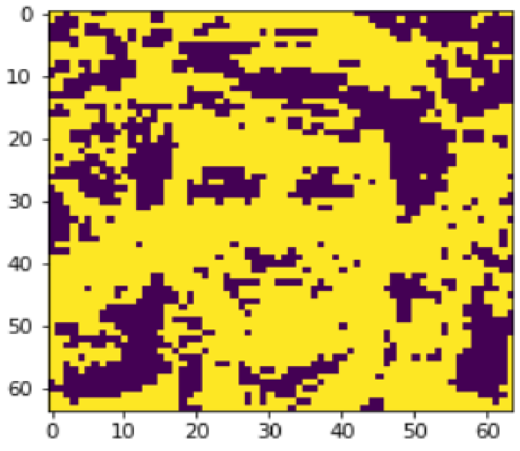


3. **4.**

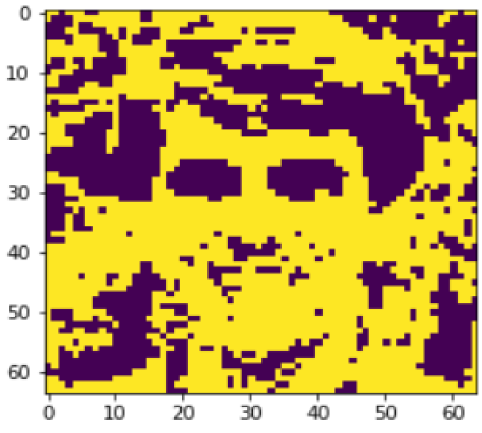
Test2_image:



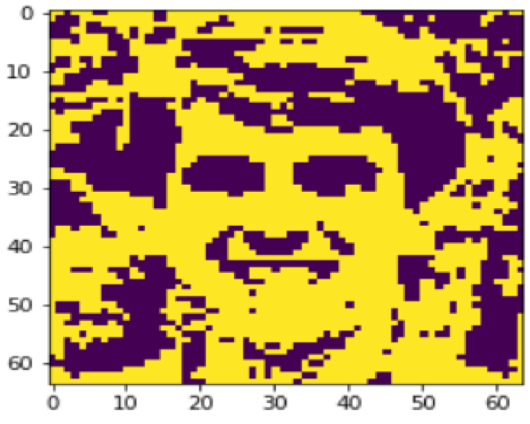
1.



2.

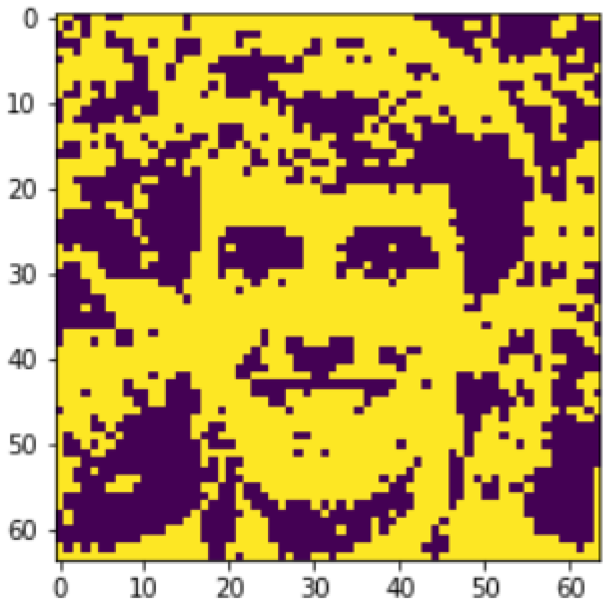


3.

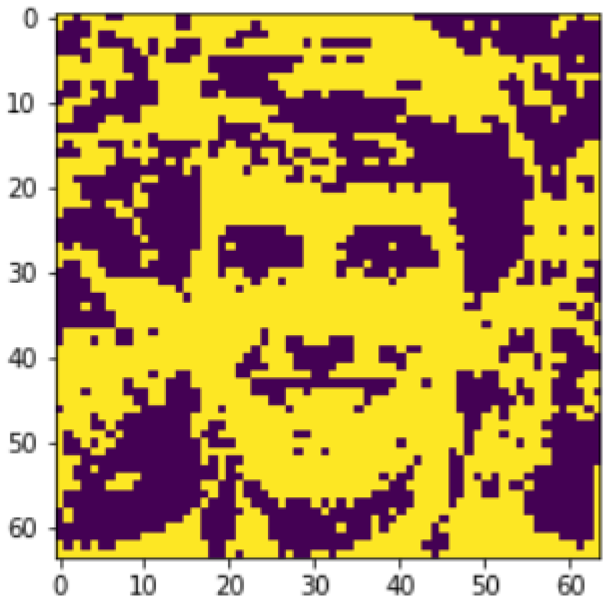


4.

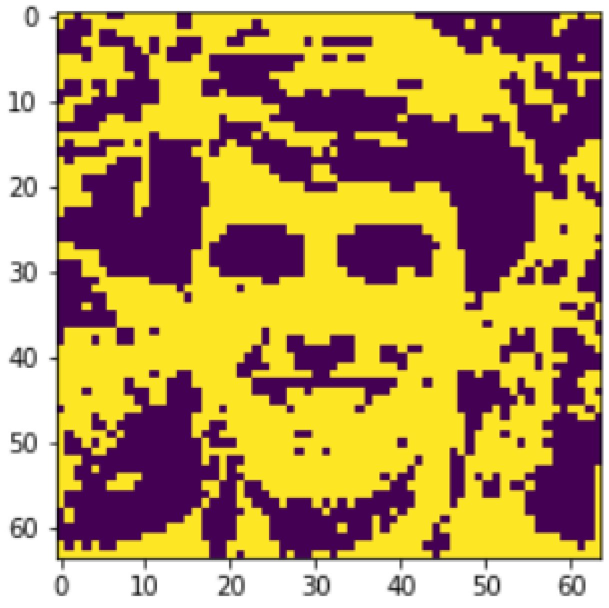
Test3_image:



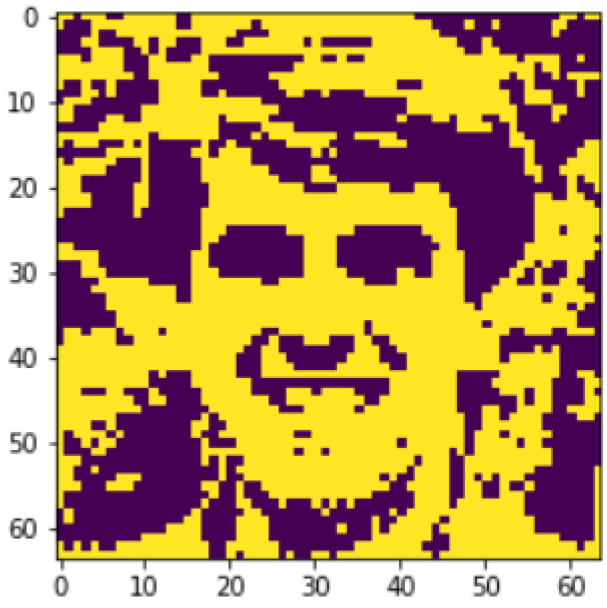
1.



2.

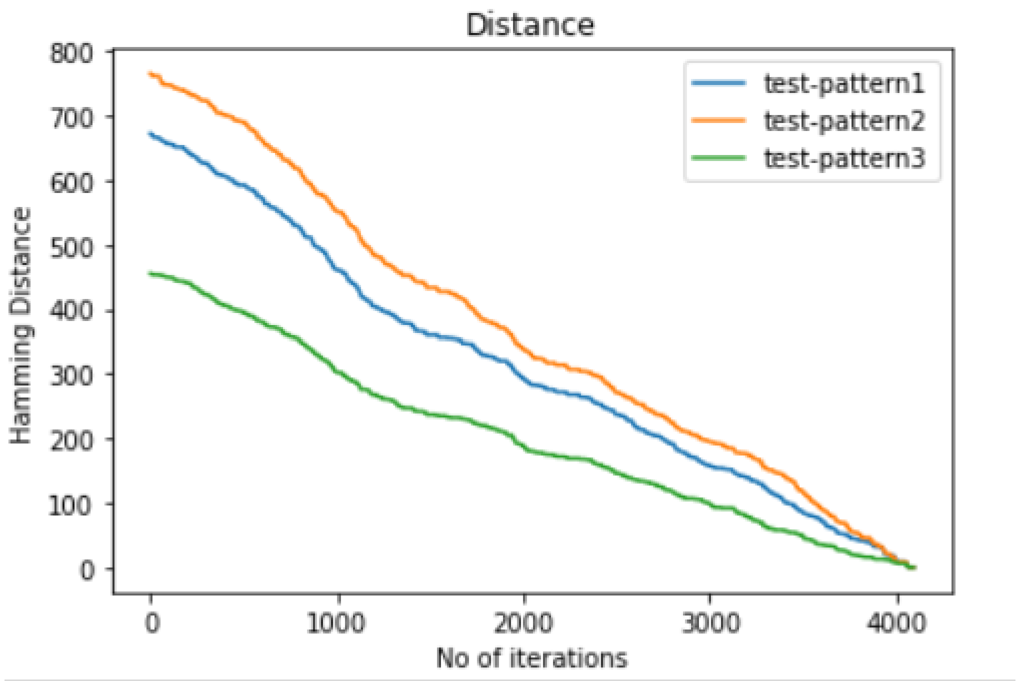


3.



4.

4.5 HAMMING DISTANCE PER ITERATIONS



Figure?. Sample Representation of a Hamming Distance

Further, you can see the Hamming Distance for the different Images.

5 QUESTION #4: BIDIRECTIONAL ASSOCIATIVE MEMORY

In this part we intend to implement BIDIRECTIONAL ASSOCIATIVE MEMORY in case of Pattern Association.

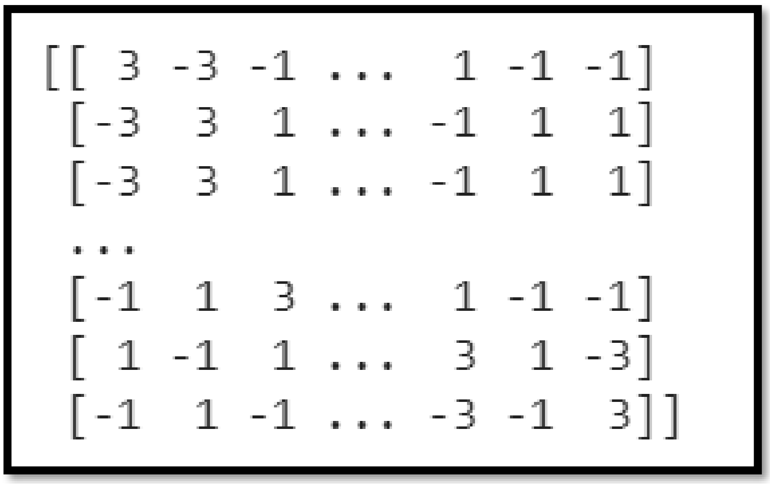
Further, we are going to report the results of this section one by one:

5.1 WEIGHT MATRIX

In this section we are dealing with a Hebbian Rule algorithm which is used in both direction and make a PAM network.

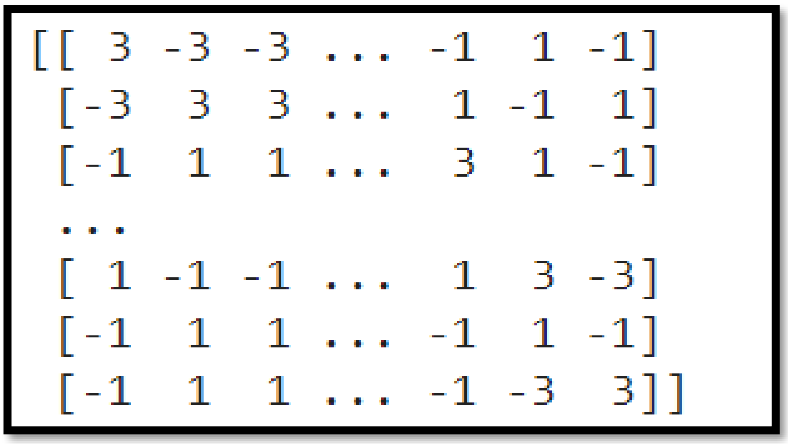
In this part we use as same as the approaches we have used in the question 2 in case of Hebbian rule.

Further you can see the bi-directional weight matrix which generate y as of its output:



```
[[ 3 -3 -1 ...  1 -1 -1]
 [-3  3  1 ... -1  1  1]
 [-3  3  1 ... -1  1  1]
 ...
 [-1  1  3 ...  1 -1 -1]
 [ 1 -1  1 ...  3  1 -3]
 [-1  1 -1 ... -3 -1  3]]
```

Further you can see the bi-directional weight matrix which generate x as of its output:



```
[[ 3 -3 -3 ... -1  1 -1]
 [-3  3  3 ...  1 -1  1]
 [-1  1  1 ...  3  1 -1]
 ...
 [ 1 -1 -1 ...  1  3 -3]
 [-1  1  1 ... -1  1 -1]
 [-1  1  1 ... -1 -3  3]]
```

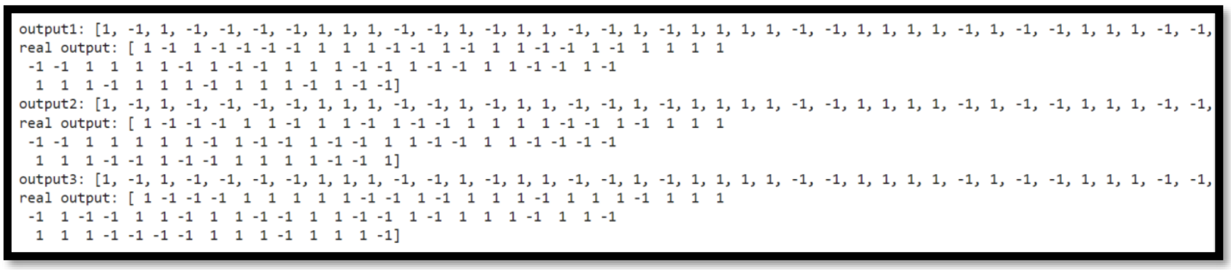
Figure?. Weight matrix representation

5.2 Evaluation of BAM Algorithm in both Direction

In this section we are going to check the performance of our algorithm in both direction.

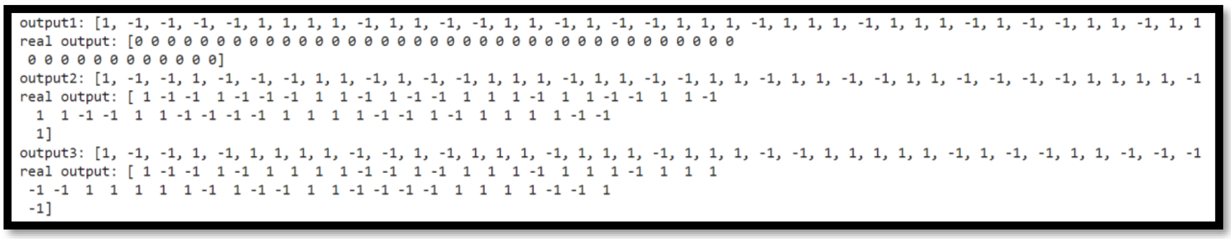
If we pass the x as of our input we must get a y vector as of our output-vector and if we pass the y as of our input we must get a x vector as of output-vector.

Further, you can see the original-value VS predicted-value and as you can see all are the same and it's a good reason which we are walking on a right path.



```
output1: [1, -1, 1, -1, -1, -1, -1, 1, 1, 1, -1, -1, 1, -1, 1, 1, -1, -1, 1, -1, 1, 1, 1, 1, -1, -1, 1, 1, 1, 1, -1, 1, -1, -1, 1, 1, 1, -1, -1,
real output: [ 1 -1  1 -1 -1 -1 -1  1  1  1 -1 -1  1 -1  1  1 -1 -1  1 -1  1  1  1  1
 -1 -1  1  1  1  1 -1  1 -1 -1  1  1  1 -1 -1  1 -1 -1  1  1 -1 -1  1 -1
  1  1  1 -1  1  1  1 -1  1  1  1 -1  1 -1 -1]
output2: [1, -1, 1, -1, -1, -1, -1, 1, 1, 1, -1, -1, 1, -1, 1, 1, -1, -1, 1, -1, 1, 1, 1, 1, -1, -1, 1, 1, 1, 1, -1, 1, -1, -1, 1, 1, 1, -1, -1,
real output: [ 1 -1 -1 -1  1  1 -1  1  1 -1  1 -1 -1  1  1  1  1 -1 -1  1 -1  1  1  1
 -1 -1  1  1  1  1  1 -1  1 -1 -1  1 -1 -1  1  1 -1 -1  1  1 -1 -1 -1 -1
  1  1  1 -1 -1  1 -1 -1  1  1  1  1 -1 -1  1]
output3: [1, -1, 1, -1, -1, -1, -1, 1, 1, 1, -1, -1, 1, -1, 1, 1, -1, -1, 1, -1, 1, 1, 1, 1, -1, -1, 1, 1, 1, 1, -1, 1, -1, -1, 1, 1, 1, -1, -1,
real output: [ 1 -1 -1 -1  1  1  1  1  1 -1 -1  1 -1  1  1  1 -1  1  1  1 -1  1  1  1
 -1  1 -1 -1  1  1 -1  1  1 -1 -1  1  1 -1 -1  1 -1  1  1  1 -1  1  1 -1
  1  1  1 -1 -1 -1 -1  1  1  1 -1  1  1  1 -1]
```

Figure?. The first direction original-output vs predicted-output



```
output1: [1, -1, -1, -1, -1, 1, 1, 1, 1, -1, 1, 1, -1, -1, 1, 1, -1, 1, -1, -1, 1, 1, 1, -1, 1, 1, 1, -1, 1, 1, 1, -1, 1, -1, -1, 1, 1, -1, 1, 1
real output: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0]
output2: [1, -1, -1, 1, -1, -1, -1, 1, 1, -1, 1, -1, -1, 1, 1, 1, -1, 1, 1, -1, -1, 1, 1, -1, 1, 1, -1, -1, 1, 1, -1, -1, -1, -1, 1, 1, 1, 1, -1
real output: [ 1 -1 -1  1 -1 -1 -1  1  1 -1  1 -1 -1  1  1  1 -1  1  1 -1 -1  1  1 -1
  1  1 -1 -1  1  1 -1 -1 -1 -1  1  1  1  1 -1 -1  1 -1  1  1  1  1 -1 -1
  1]
output3: [1, -1, -1, 1, -1, 1, 1, 1, 1, -1, -1, 1, -1, 1, 1, 1, -1, 1, 1, 1, -1, 1, 1, 1, -1, -1, 1, 1, 1, 1, 1, -1, 1, -1, -1, 1, 1, -1, -1, -1
real output: [ 1 -1 -1  1 -1  1  1  1  1 -1 -1  1 -1  1  1  1 -1  1  1  1 -1  1  1  1
 -1 -1  1  1  1  1  1 -1  1 -1 -1  1  1 -1 -1 -1 -1  1  1  1  1 -1 -1  1
 -1]
```

Figure?. The second direction original-output vs predicted-output

as you can see the results are totally correct.

5.3 EVALUATION OF BAM ALGORITHM IN CASE OF NOISE ADDING

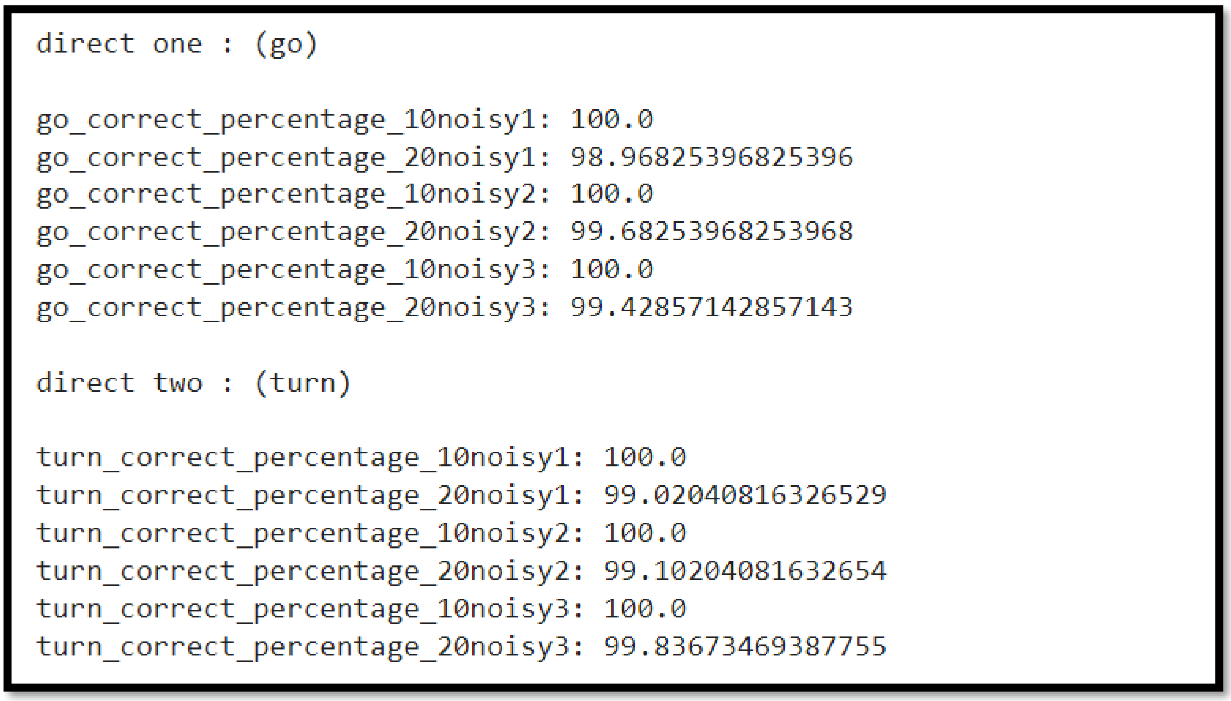
In this section, according to the question description I have used the previous function to add noise to the inputs and I repeat that for both direction one-by-one.

And also, I have used two percent noise value abbreviated below:

1- 10% noise

2- 20% noise

Further I have reported the result of this section :



```
direct one : (go)

go_correct_percentage_10noisy1: 100.0
go_correct_percentage_20noisy1: 98.96825396825396
go_correct_percentage_10noisy2: 100.0
go_correct_percentage_20noisy2: 99.68253968253968
go_correct_percentage_10noisy3: 100.0
go_correct_percentage_20noisy3: 99.42857142857143

direct two : (turn)

turn_correct_percentage_10noisy1: 100.0
turn_correct_percentage_20noisy1: 99.02040816326529
turn_correct_percentage_10noisy2: 100.0
turn_correct_percentage_20noisy2: 99.10204081632654
turn_correct_percentage_10noisy3: 100.0
turn_correct_percentage_20noisy3: 99.83673469387755
```

Figure?. Correct-percentage of bidirectional associative Memory

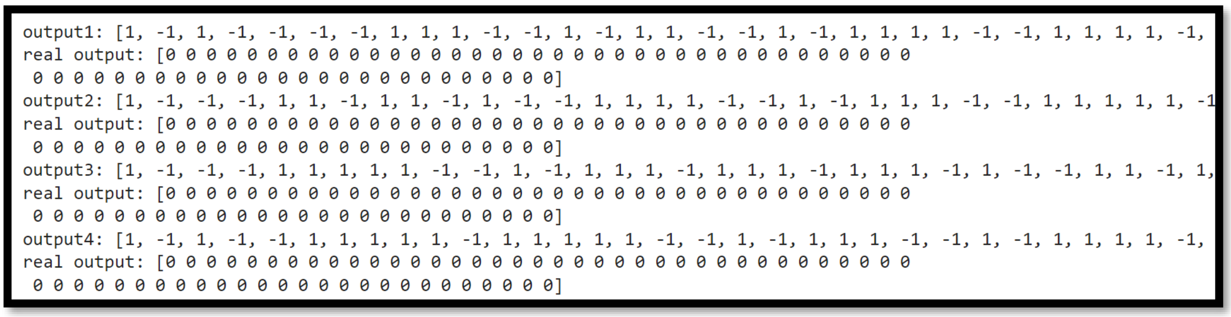
Further I have reported all of above value in a table:

	Clinton(path1)	Hillary(path1)	Kenstar(path1)	Clinton(path2)	Hillary(path2)	Kenstar(path2)
10% noise	100%	100%	100%	100%	100%	100%
20% noise	98.96%	99.68%	99.42%	99.02%	99.10%	99.84%

5.4 Evaluation of BAM Algorithm in case of noise adding

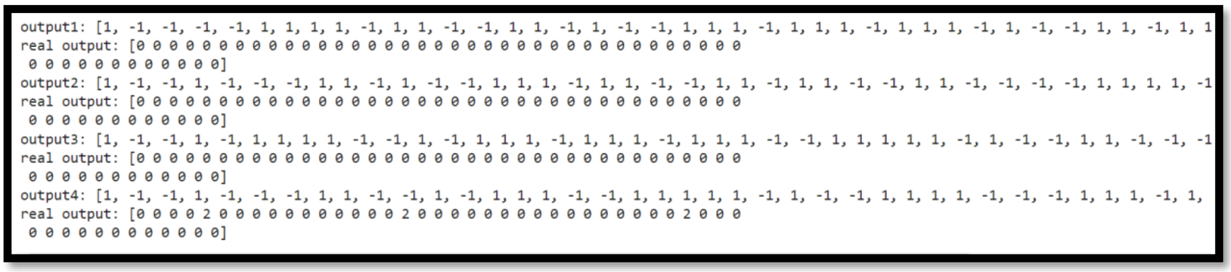
Now, we add another x and y vector to our train-sets and re-train the algorithm to investigate the results:

Further we can see the results of prediction on the new date:



```
output1: [1, -1, 1, -1, -1, -1, -1, 1, 1, 1, -1, -1, 1, -1, 1, 1, -1, -1, 1, -1, 1, 1, 1, 1, -1, -1, 1, 1, 1, 1, -1,
real output: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
output2: [1, -1, -1, -1, 1, 1, -1, 1, 1, -1, 1, -1, -1, 1, 1, 1, 1, -1, -1, 1, -1, 1, 1, 1, -1, -1, 1, 1, 1, 1, 1, -1
real output: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
output3: [1, -1, -1, -1, 1, 1, 1, 1, 1, -1, -1, 1, -1, 1, 1, 1, -1, 1, 1, 1, -1, 1, 1, 1, -1, 1, -1, -1, 1, 1, -1, 1,
real output: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
output4: [1, -1, 1, -1, -1, 1, 1, 1, 1, 1, -1, 1, 1, 1, 1, 1, -1, -1, 1, -1, 1, 1, 1, -1, -1, 1, -1, 1, 1, 1, 1, -1,
real output: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

Figure?. The first direction original-output vs predicted-output



```
output1: [1, -1, -1, -1, -1, 1, 1, 1, 1, -1, 1, 1, -1, -1, 1, 1, -1, 1, -1, -1, 1, 1, 1, -1, 1, 1, 1, -1, 1, 1, 1, -1, 1, -1, -1, 1, 1, -1, 1, 1
real output: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0]
output2: [1, -1, -1, 1, -1, -1, -1, 1, 1, -1, 1, -1, -1, 1, 1, 1, -1, 1, 1, -1, -1, 1, 1, -1, 1, 1, -1, -1, 1, 1, -1, -1, -1, -1, 1, 1, 1, 1, -1
real output: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0]
output3: [1, -1, -1, 1, -1, 1, 1, 1, 1, -1, -1, 1, -1, 1, 1, 1, -1, 1, 1, 1, -1, 1, 1, 1, -1, -1, 1, 1, 1, 1, 1, -1, 1, -1, -1, 1, 1, -1, -1, -1
real output: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0]
output4: [1, -1, -1, 1, -1, -1, -1, 1, 1, -1, -1, 1, -1, 1, 1, 1, -1, -1, 1, 1, 1, 1, 1, -1, 1, -1, -1, 1, 1, 1, 1, -1, -1, -1, 1, 1, 1, -1, 1,
real output: [0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

Figure?. The second direction original-output vs predicted-output

As we can see in the Return path we could not predict the input properly and as you can see there are lots of digits which are not same with the original-output!

And then we can state that the performance of our network has been decrease and we loss some of data during predictions.

Because when we add a new variable the weight matrix tends to be in a situation that m > n which means the number of rows is greater than the number of columns or the number of rows is greater than the number of variables and then we could not solve this system of equations using weight matrix and **should be use from pseudo Inverse rule algorithm to predict properly.**

6 REFERENCES

[1] https://www.geeksforgeeks.org/hopfield-neural-network/

[2] https://stackoverflow.com/questions/21465988/python-equivalent-to-hold-on-in-matlab

[3] https://www.geeksforgeeks.org/bidirectional-associative-memory-bam-implementation-from-scratch/

[4] https://www.codegrepper.com/code-examples/python/change+image+size+python

[5] https://www.hackerearth.com/practice/notes/extracting-pixel-values-of-an-image-in-python/

[6] https://numpy.org/doc/stable/reference/generated/numpy.concatenate.html

[7] https://www.geeksforgeeks.org/how-to-compute-the-pseudoinverse-of-a-matrix-in-pytorch/

[8] https://www.geeksforgeeks.org/moore-penrose-pseudoinverse-mathematics/

[9] https://www.geeksforgeeks.org/association-rule/

[10]https://www.researchgate.net/publication/236093245_Learning_the_pseudoinverse_solution_to_network_weights

[11] https://stackoverflow.com/questions/21254472/multiple-plots-in-one-figure-in-python

[12] https://stackoverflow.com/questions/9638826/plot-a-black-and-white-binary-map-in-matplotlib

[13] https://matplotlib.org/3.5.0/api/_as_gen/matplotlib.pyplot.legend.html