University of Tehran
Neural Network and Deep Learning
HW3

Milad Mohamadi
Student Number: 810100462

May 14, 2022

Questions List Table

1 Question-1 3

2 Question-2 34

3 Question-3 52

4 part 1 52

5 Discrete Hopfield Network: 52

6 Structure & Architecture 52

7 Training Algorithm 52

8 part 2 53

9 part 3 57

10 part 4 57

11 part 5 60

12 Question-4 61

13 Part 1 63

14 Part 2 63

15 Model evaluation (direction) 63

16 Model evaluation(Reverse) 64

17 Part 3 64

18 Model evaluation with 10% noise(direction) 64

19 Model evaluation with 10% noise(Reverse) 65

20 Model evaluation with 20% noise(direction) 65

21 Model evaluation with 20% noise(Reverse) 66

22 Part 5 66

23 Model evaluation(Direction) 67

24 Model evaluation(Reverse) 67

1 Question-1

Part 1

Hebbian Learning Rule 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 :

1. Set all weights to zero, wi = 0 for i=1 to n, and bias to zero.
2. For each input vector, S(input vector) : t(target output pair), repeat steps 3-5.
3. Set activations for input units with the input vector Xi = Si for i = 1 to n.
4. Set the corresponding output value to the output neuron, i.e. y = t.
5. Update weight and bias by applying Hebb rule for all i = 1 to n:

```
Wi(new) = wi(old) + xi * y
  B(new) = B(old) + y
```

Activation Function:

```
if net >= 0    :   f(net) = 1
if net < 0     :   f(net) = -1
```

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
import random
import copy
```

Inputs and outputs

[ ]:
```
input_1 = np.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,-1,-1,1,-1,-1,-1,
    -1,-1,-1,1,-1,-1,-1,
    -1,-1,-1,1,-1,-1,-1,
    -1,-1,-1,1,-1,-1,-1,
])

input_2 = np.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,-1,-1,-1,-1,-1,1,
    1,1,1,1,1,1,1,
    -1,-1,-1,-1,-1,-1,-1,
    -1,-1,-1,1,-1,-1,-1,
])


input_3 = np.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,-1,-1,-1,-1,-1,1,
    1,-1,-1,-1,-1,-1,1,
    1,-1,-1,-1,-1,-1,1,
    1,1,1,1,1,1,1,
])


output_1 = np.array([
    -1,-1,1,
    1,1,1,
    1,1,-1,
    -1,1,-1,
    -1,1,-1,
])

output_2 = np.array([
    1,-1,1,
    1,-1,1,
    1,1,1,
    -1,-1,-1,
    -1,1,-1
])

output_3 = np.array([
    1,-1,1,
    1,-1,1,
    1,1,1,
    1,-1,1,
    1,1,1
])
```

plotting inputs and outputs

[ ]:
```
fig = plt.figure(figsize=(8, 8))
fig.add_subplot(2, 3, 1)
plt.imshow(input_1.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('input_1')
fig.add_subplot(2, 3, 2)
plt.imshow(input_2.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('input_2')
fig.add_subplot(2, 3, 3)
plt.imshow(input_3.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('input_3')

fig.add_subplot(2, 3, 4)
plt.imshow(output_1.reshape((5, 3)), interpolation='nearest',cmap='Greys')
plt.title('output_1')
fig.add_subplot(2, 3, 5)
plt.imshow(output_2.reshape((5, 3)), interpolation='nearest',cmap='Greys')
plt.title('output_2')
fig.add_subplot(2, 3, 6)
plt.imshow(output_3.reshape((5, 3)), interpolation='nearest',cmap='Greys')
plt.title('output_3')
```

```
plt.show()
```

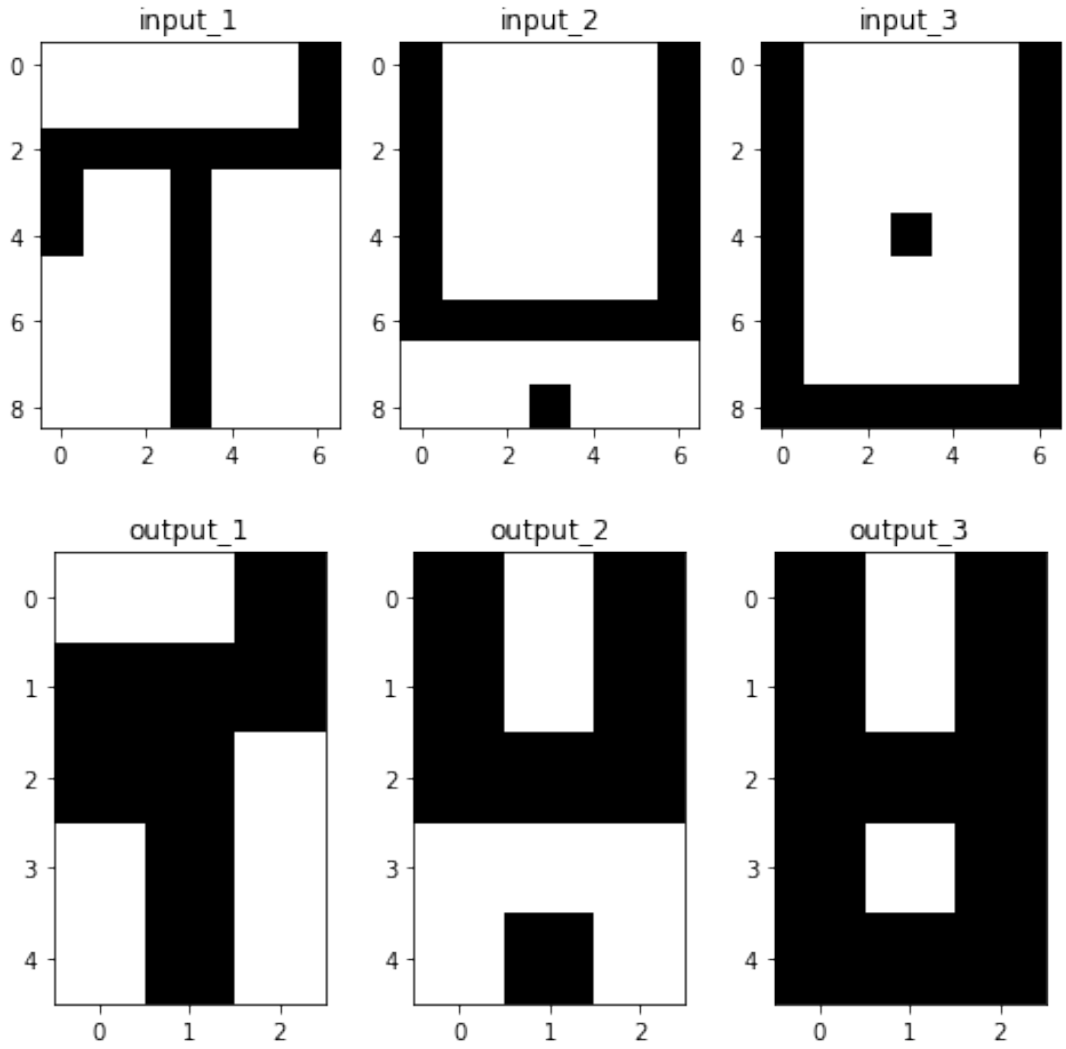


Figure 1: inputs and outputs

Define the model

```
[ ]: class Hebbian:
    def __init__(self, _S, _T):
        self.S = _S
        self.T = _T
        self.W = np.zeros((len(_S[1]), len(_T[1])))

    def train(self):
        for trainSample in range(len(self.S)):
            x = self.S[trainSample]
            y = self.T[trainSample]
            for i in range(len(x)):
                for j in range(len(y)):
```

```
                    self.W[i][j] = self.W[i][j] + x[i]*y[j]
    def showResult_1(self, X):
        y_t = np.dot(X,self.W)
        for i in range(len(y_t)): #sign
            if(y_t[i] >= 0):
                y_t[i] = 1
            else:
                y_t[i] = -1
        plt.figure()
        plt.imshow(y_t.reshape((5, 3)), cmap='binary')

    def showResult_2(self, X):
        y_t = np.dot(X,self.W)
        for i in range(len(y_t)):
            if(y_t[i] >= 0):
                y_t[i] = 1
            else:
                y_t[i] = -1
        plt.figure()
        plt.imshow(y_t.reshape((2, 2)), cmap='binary')
```

Train the model

[ ]:
```
input_1_reshaped = input_1.reshape((input_1.shape[0]))
input_2_reshaped = input_2.reshape((input_2.shape[0]))
input_3_reshaped = input_3.reshape((input_3.shape[0]))
output_1_reshaped = output_1.reshape((output_1.shape[0]))
output_2_reshaped = output_2.reshape((output_2.shape[0]))
output_3_reshaped = output_3.reshape((output_3.shape[0]))

S = np.array([copy.deepcopy(input_1_reshaped), copy.deepcopy(input_2_reshaped),␣
 ↪copy.deepcopy(input_3_reshaped)])
T = np.array([copy.deepcopy(output_1_reshaped), copy.deepcopy(output_2_reshaped),␣
 ↪copy.deepcopy(output_3_reshaped)])
heb = Hebbian(S, T)
heb.train()
```

Part 2

Weights

[ ]:
```
heb.W
```

[ ]:
```
array([[ 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.]])
```

Part 3

As we can see all three of our optimal outputs have been produced

```
[ ]: heb.showResult_1(input_1)
     heb.showResult_1(input_2)
     heb.showResult_1(input_3)
```

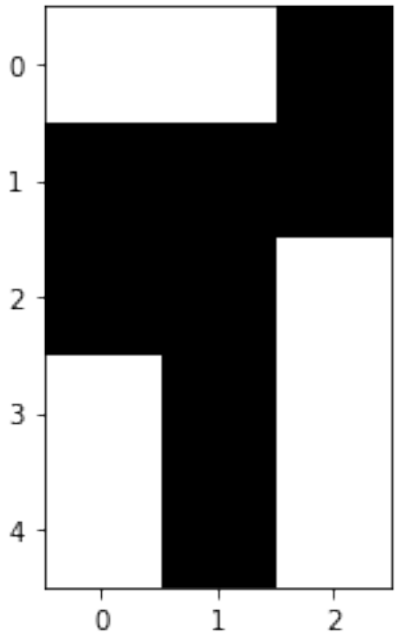


Figure 2: predicted image for input-1

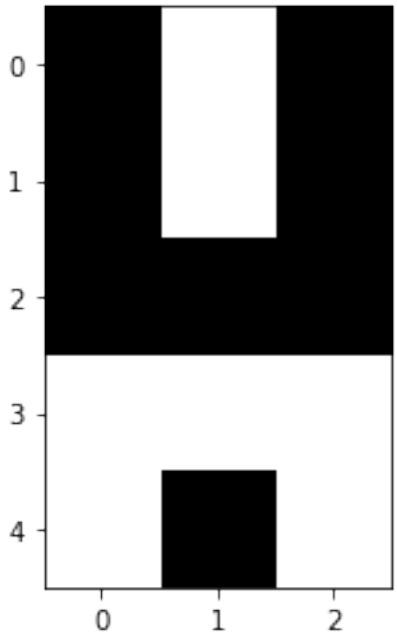


Figure 3: predicted image for input-2

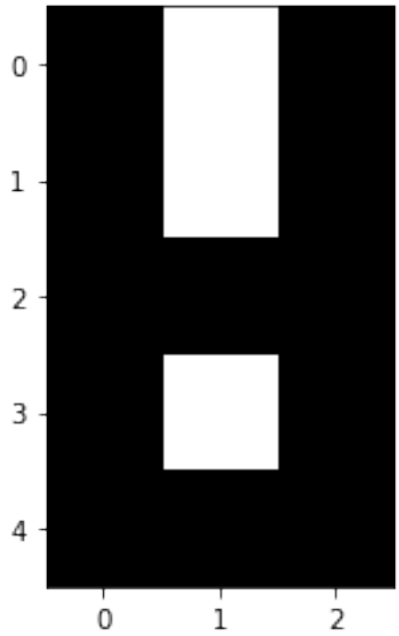


Figure 4: predicted image for input-3

Part 4

The smallest dimension we can reduce to is 2x2

[ ]:
```
input_1_smallest = np.array([
    1, 1,
    -1,1
]).reshape(2,2)

input_2_smallest = np.array([
    1, 1,
    -1,-1
]).reshape(2,2)

input_3_smallest = np.array([
    1, 1,
    1,-1
]).reshape(2,2)
```

[ ]:
```
plt.imshow(input_1_smallest.reshape((2,2)), interpolation='nearest',cmap='Greys')
plt.show()
```

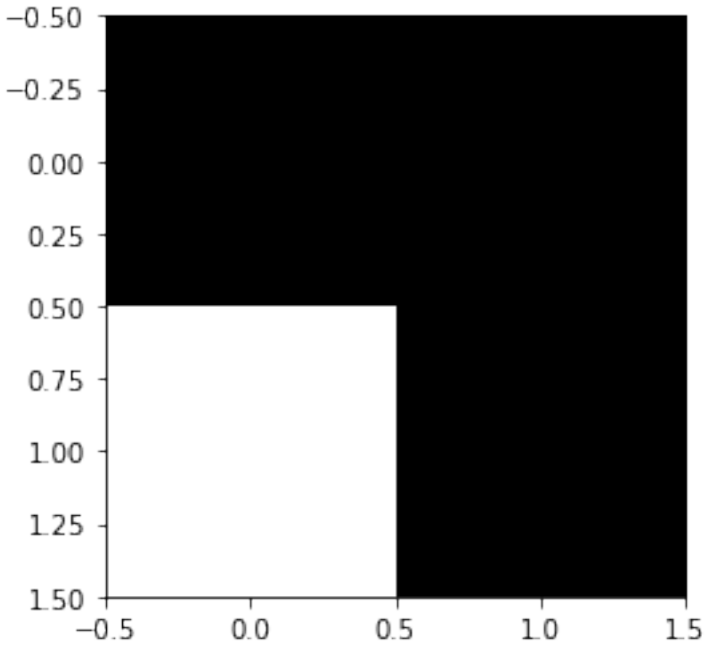


Figure 5: input-1 in 2*2 size

```
[ ]: input_1_reshaped = input_1.reshape((input_1.shape[0]))
     input_2_reshaped = input_2.reshape((input_2.shape[0]))
     input_3_reshaped = input_3.reshape((input_3.shape[0]))
     input_1_smallest_reshaped = input_1_smallest.reshape((4))
     input_2_smallest_reshaped = input_2_smallest.reshape((4))
     input_3_smallest_reshaped = input_3_smallest.reshape((4))

     S = np.array([copy.deepcopy(input_1_reshaped), copy.deepcopy(input_2_reshaped),␣
      ↪copy.deepcopy(input_3_reshaped)])
     T = np.array([copy.deepcopy(input_1_smallest_reshaped), copy.
      ↪deepcopy(input_2_smallest_reshaped), copy.deepcopy(input_3_smallest_reshaped)])
     heb2 = Hebbian(S,T)
     heb2.train()
```

```
[ ]: heb2.showResult_2(input_1)
     heb2.showResult_2(input_2)
     heb2.showResult_2(input_3)
```

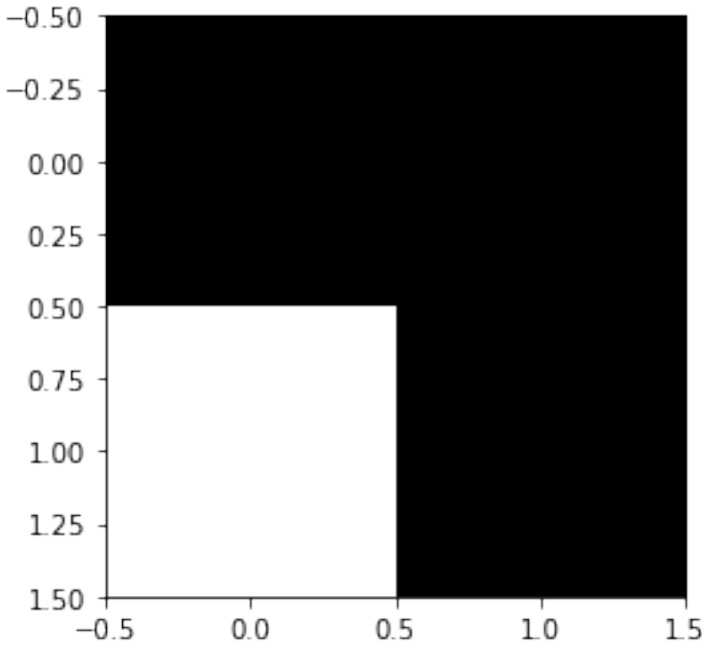


Figure 6: predicted image for input-1 in 2*2 size

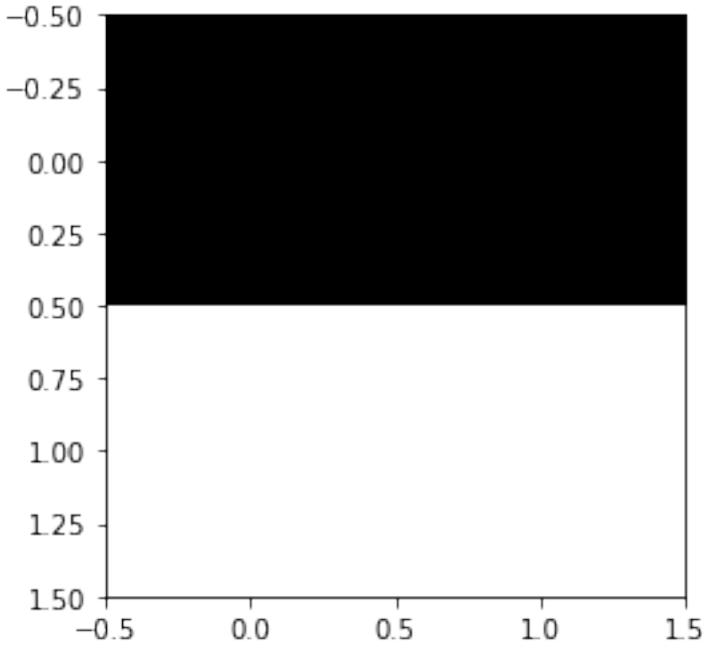


Figure 7: predicted image for input-2 in 2*2 size

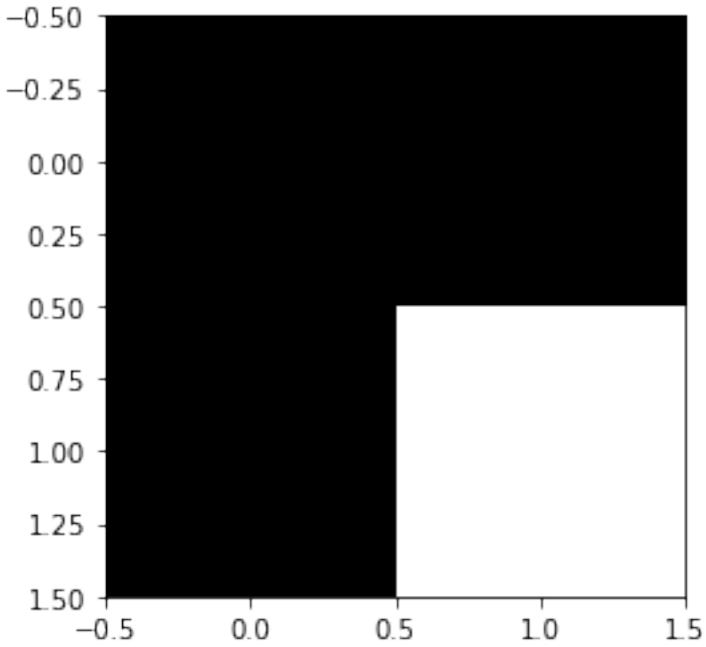


Figure 8: predicted image for input-3 in 2*2 size

Part 5

Adding noise 20% to input of parts two and four of the model

The accuracy of part 2's model with 20% noise applied is 100%.

[ ]:
```
def ResultWhitNoise(arr,percentage):
    pixels = int(np.prod(arr.shape)*percentage)
    with_noise = np.copy(arr)
    for pixel in range(pixels) :
      random_pixel = np.random.choice(np.prod(with_noise.shape), 1)
      if with_noise.flat[random_pixel] == -1 :
        with_noise.flat[random_pixel] = 1
      else:
        with_noise.flat[random_pixel] = -1

    return with_noise
```

[ ]:
```
input_1_noise_20 = ResultWhitNoise(input_1,0.2)
input_2_noise_20 = ResultWhitNoise(input_2,0.2)
input_3_noise_20 = ResultWhitNoise(input_3,0.2)
```

[ ]:
```
fig = plt.figure(figsize=(8, 8))
fig.add_subplot(1, 3, 1)
plt.imshow(input_1_noise_20.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('input_1 whit noise')
fig.add_subplot(1, 3, 2)
plt.imshow(input_2_noise_20.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('input_2 whit noise')
fig.add_subplot(1, 3, 3)
```

```
plt.imshow(input_3_noise_20.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('input_3 with noise')
plt.show()
```

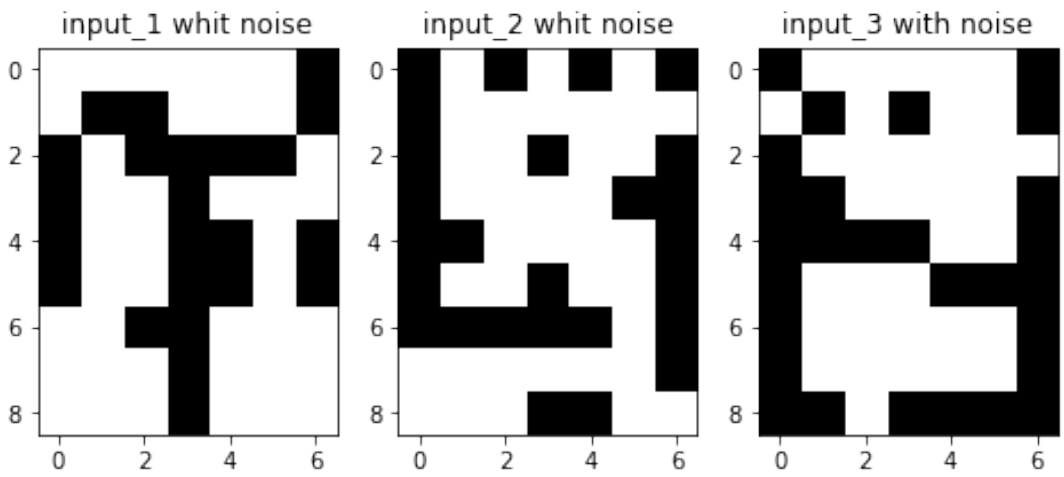


Figure 9: inputs images after 20% noise

[ ]:
```
S = np.array([copy.deepcopy(input_1), copy.deepcopy(input_2), copy.
 ↪deepcopy(input_3)])
T = np.array([copy.deepcopy(output_1), copy.deepcopy(output_2), copy.
 ↪deepcopy(output_3)])
heb3 = Hebbian(S,T)
heb3.train()
```

[ ]:
```
heb3.showResult_1(input_1_noise_20)
heb3.showResult_1(input_2_noise_20)
heb3.showResult_1(input_3_noise_20)
```

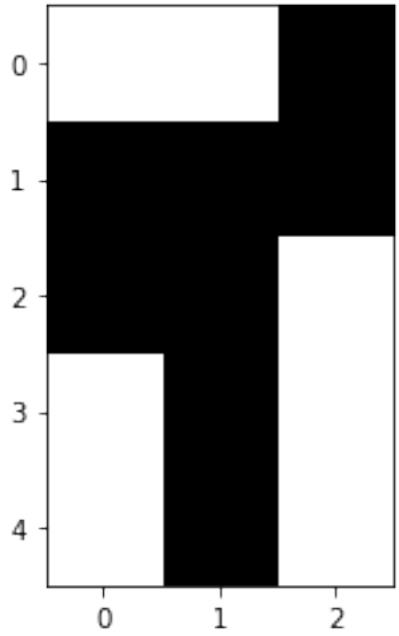


Figure 10: predicted image for input-1 with 20% noise in 5*3 size

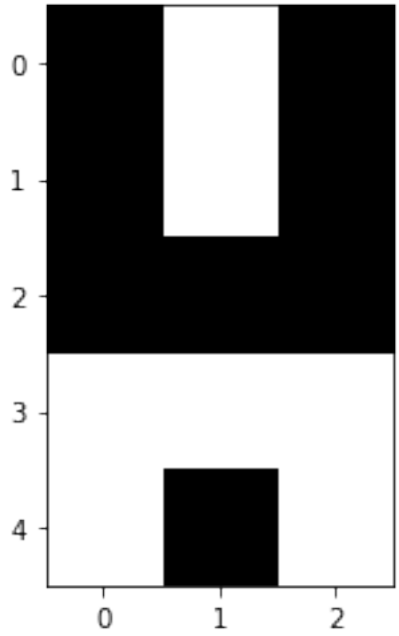


Figure 11: predicted image for input-2 with 20% noise in 5*3 size

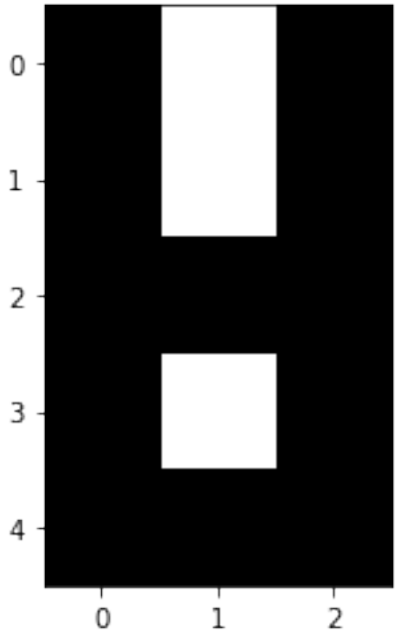


Figure 12: predicted image for input-3 with 20% noise in 5*3 size

The accuracy of part 4's model with 20% noise applied is 100%.

[ ]:
```
input_1_noise_20 = ResultWhitNoise(input_1,0.2)
input_2_noise_20 = ResultWhitNoise(input_2,0.2)
input_3_noise_20 = ResultWhitNoise(input_3,0.2)
```

[ ]:
```
S = np.array([copy.deepcopy(input_1), copy.deepcopy(input_2), copy.
 ↪deepcopy(input_3)])
T = np.array([copy.deepcopy(input_1_smallest_reshaped), copy.
 ↪deepcopy(input_2_smallest_reshaped), copy.deepcopy(input_3_smallest_reshaped)])
heb4 = Hebbian(S,T)
heb4.train()
```

[ ]:
```
heb4.showResult_2(input_1_noise_20)
heb4.showResult_2(input_2_noise_20)
heb4.showResult_2(input_3_noise_20)
```

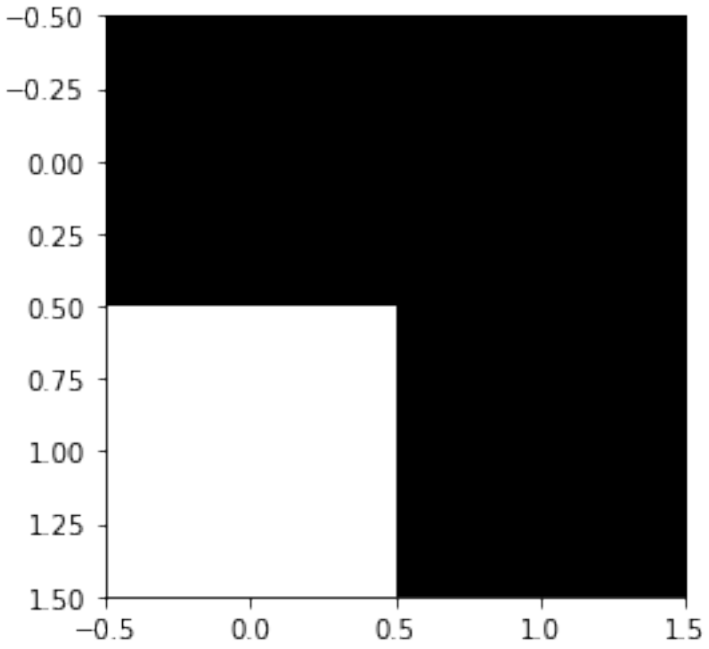


Figure 13: predicted image for input-1 with 20% noise in 2*2 size

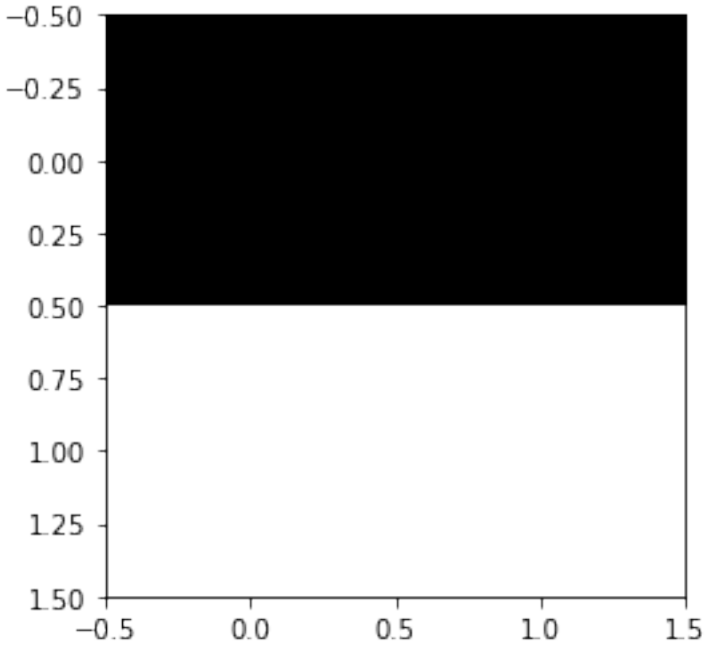


Figure 14: predicted image for input-2 with 20% noise in 2*2 size

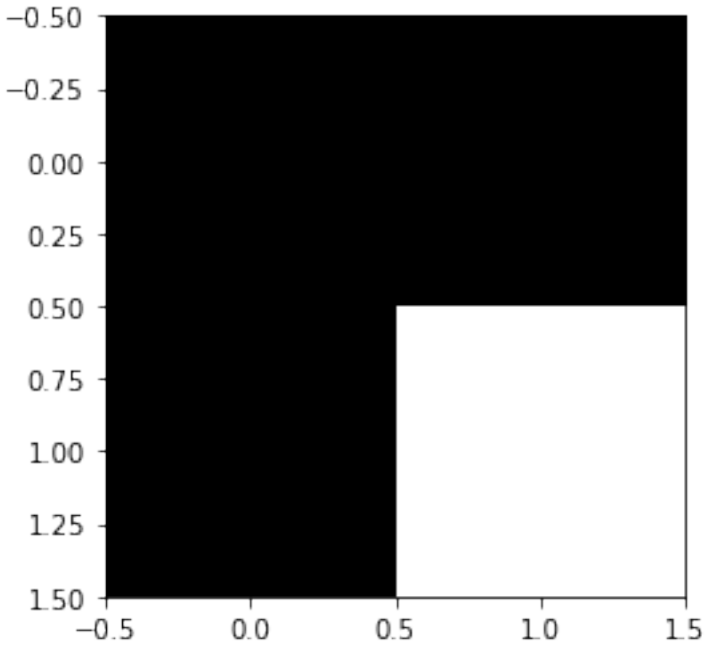


Figure 15: predicted image for input-3 with 20% noise in 2*2 size

Part 5

Adding noise 60% to input of parts two and four of the model

The accuracy of part 2's model with 60% noise applied is 66.6%.

[ ]:
```
input_1_noise_60 = ResultWhitNoise(input_1,0.6)
input_2_noise_60 = ResultWhitNoise(input_2,0.6)
input_3_noise_60 = ResultWhitNoise(input_3,0.6)
```

[ ]:
```
fig = plt.figure(figsize=(8, 8))
fig.add_subplot(1, 3, 1)
plt.imshow(input_1_noise_60.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('input_1 with noise')
fig.add_subplot(1, 3, 2)
plt.imshow(input_2_noise_60.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('input_2 with noise')
fig.add_subplot(1, 3, 3)
plt.imshow(input_3_noise_60.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('input_3 with noise')
plt.show()
```

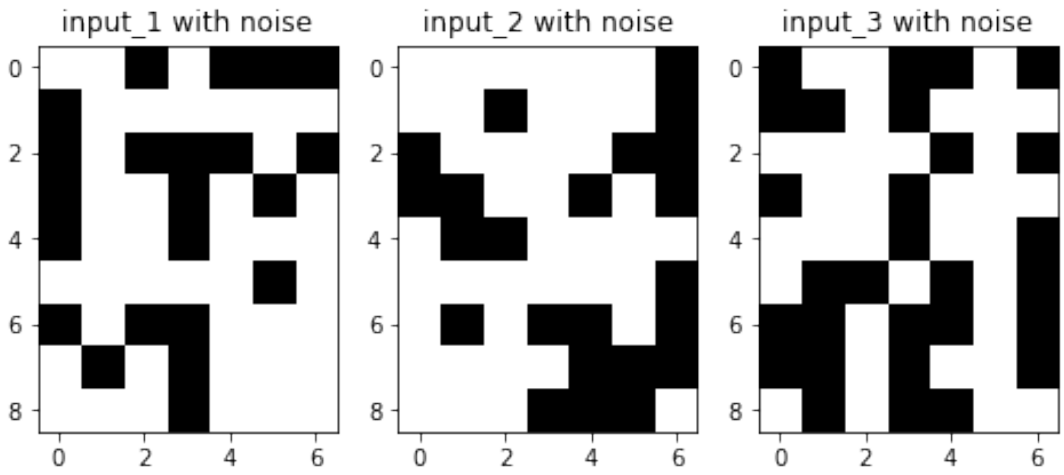


Figure 16: inputs images with 60% noise

```
[ ]: S = np.array([copy.deepcopy(input_1), copy.deepcopy(input_2), copy.
      ↪deepcopy(input_3)])
     T = np.array([copy.deepcopy(output_1), copy.deepcopy(output_2), copy.
      ↪deepcopy(output_3)])
     heb5 = Hebbian(S,T)
     heb5.train()
```

```
[ ]: heb5.showResult_1(input_1_noise_60)
     heb5.showResult_1(input_2_noise_60)
     heb5.showResult_1(input_3_noise_60)
```

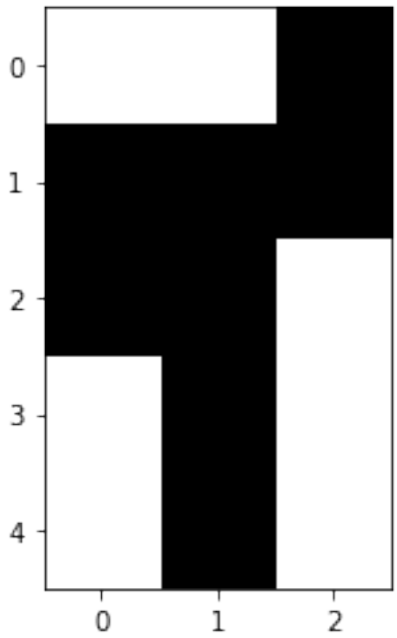


Figure 17: predicted image for input-1 with 60% noise in 5*3 size

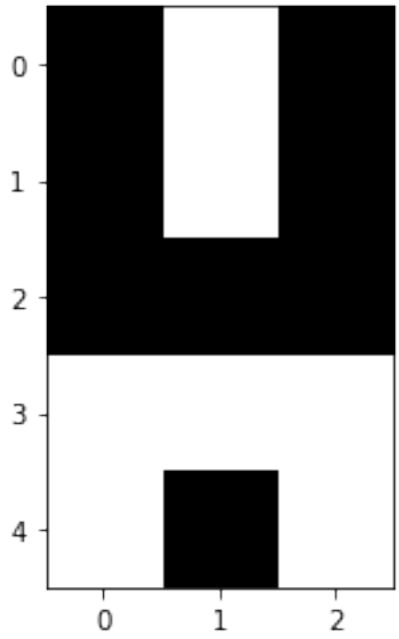


Figure 18: predicted image for input-2 with 60% noise in 5*3 size

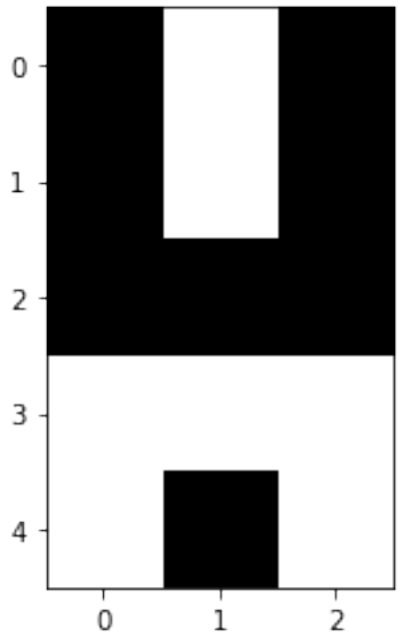


Figure 19: predicted image for input-3 with 60% noise in 5*3 size

The accuracy of part 4's model with 60% noise applied is 33.3%.

```
[ ]: input_1_noise_60 = ResultWhitNoise(input_1,0.6)
     input_1_noise_60 = ResultWhitNoise(input_2,0.6)
     input_1_noise_60 = ResultWhitNoise(input_3,0.6)
```

```
[ ]: S = np.array([copy.deepcopy(input_1), copy.deepcopy(input_2), copy.
      ↪deepcopy(input_3)])
     T = np.array([copy.deepcopy(input_1_smallest_reshaped), copy.
      ↪deepcopy(input_2_smallest_reshaped), copy.deepcopy(input_3_smallest_reshaped)])
     heb6 = Hebbian(S,T)
     heb6.train()
```

```
[ ]: heb6.showResult_2(input_1_noise_60)
     heb6.showResult_2(input_2_noise_60)
     heb6.showResult_2(input_3_noise_60)
```

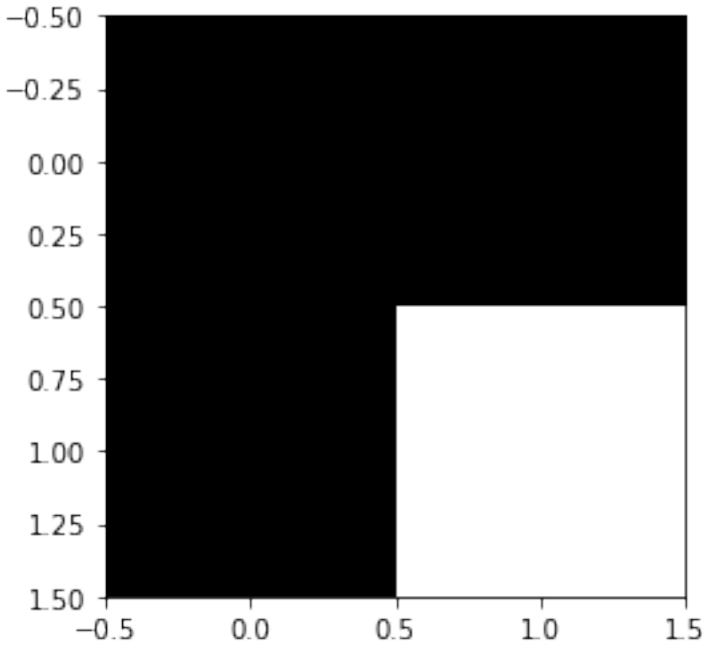


Figure 20: predicted image for input-1 with 60% noise in 2*2 size

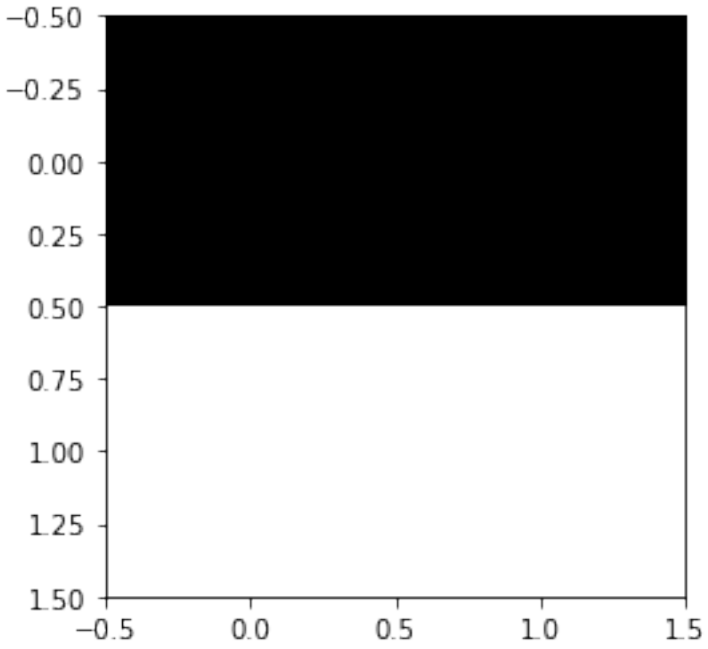


Figure 21: predicted image for input-2 with 60% noise in 2*2 size

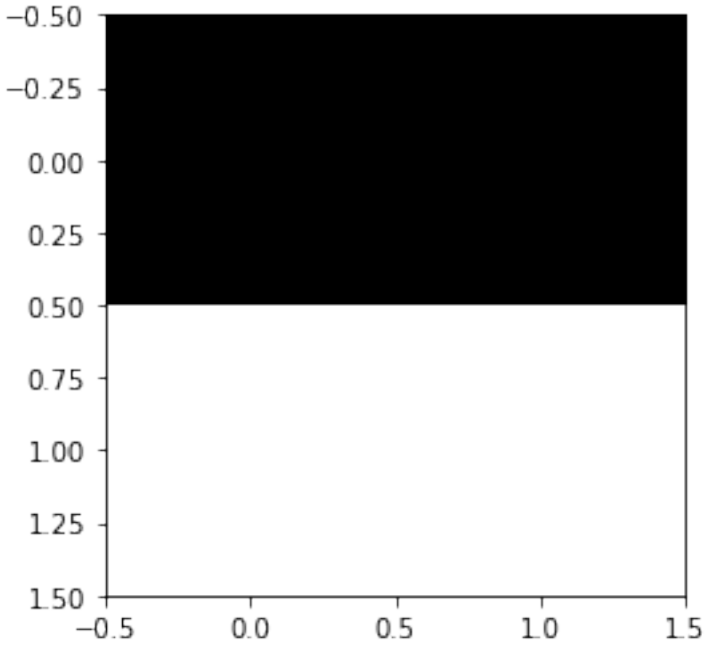


Figure 22: predicted image for input-3 with 60% noise in 2*2 size

Part 6

Results of part 2's Inputs with 20% & 60% Lost

```
[ ]: def showResultWithLost(arr,percentage):
         pixels = int(np.prod(arr.shape)*percentage)
         with_noise = np.copy(arr)
         for pixel in range(pixels) :
           random_pixel = np.random.choice(np.prod(with_noise.shape), 1)
           with_noise.flat[random_pixel] = 0
         return with_noise
```

```
[ ]: input_1_lost_20 = showResultWithLost(input_1,0.2)
     input_2_lost_20 = showResultWithLost(input_2,0.2)
     input_3_lost_20 = showResultWithLost(input_3,0.2)
```

```
[ ]: fig = plt.figure(figsize=(8, 8))
     fig.add_subplot(1, 3, 1)
     plt.imshow(input_1_lost_20.reshape((9, 7)), interpolation='nearest',cmap='Greys')
     plt.title('input_1 with lost')
     fig.add_subplot(1, 3, 2)
     plt.imshow(input_2_lost_20.reshape((9, 7)), interpolation='nearest',cmap='Greys')
     plt.title('input_2 with lost')
     fig.add_subplot(1, 3, 3)
     plt.imshow(input_3_lost_20.reshape((9, 7)), interpolation='nearest',cmap='Greys')
     plt.title('input_3 with lost')
     plt.show()
```

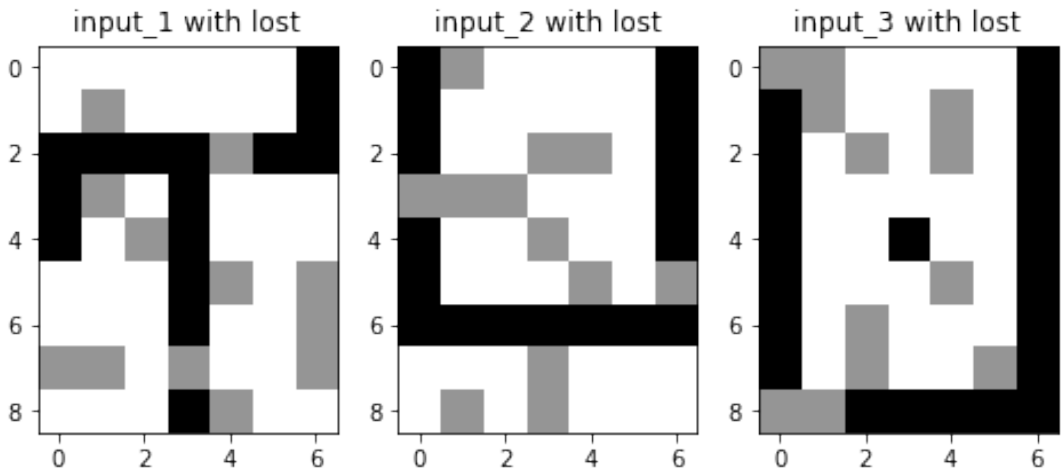


Figure 23: inputs images with 20% loss

The accuracy of model is 100%

```
[ ]: S = np.array([copy.deepcopy(input_1), copy.deepcopy(input_2), copy.
      ↪deepcopy(input_3)])
     T = np.array([copy.deepcopy(output_1), copy.deepcopy(output_2), copy.
      ↪deepcopy(output_3)])
     heb7 = Hebbian(S,T)
     heb7.train()
```

```
[ ]: heb7.showResult_1(input_1_lost_20)
     heb7.showResult_1(input_2_lost_20)
     heb7.showResult_1(input_3_lost_20)
```

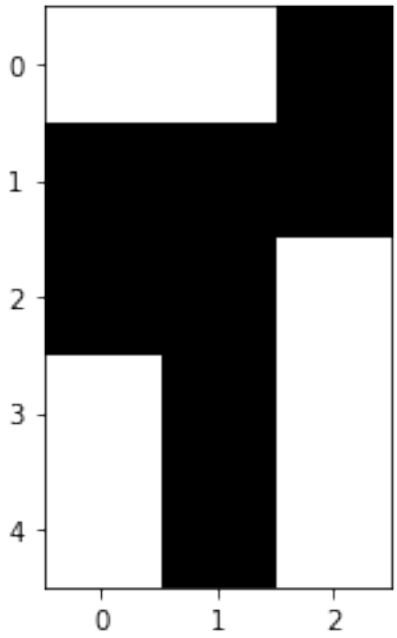


Figure 24: predicted image for input-1 with 20% loss in 5*3 size

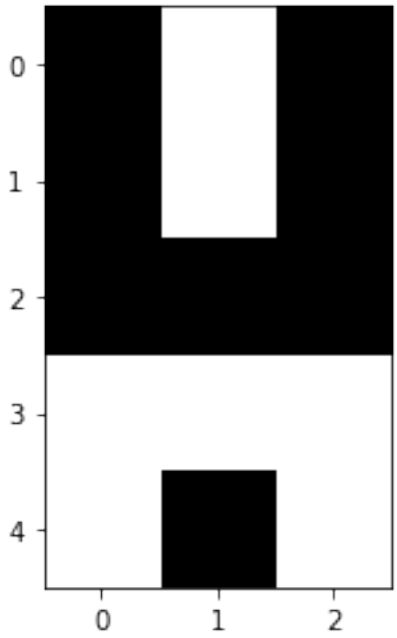


Figure 25: predicted image for input-2 with 20% loss in 5*3 size

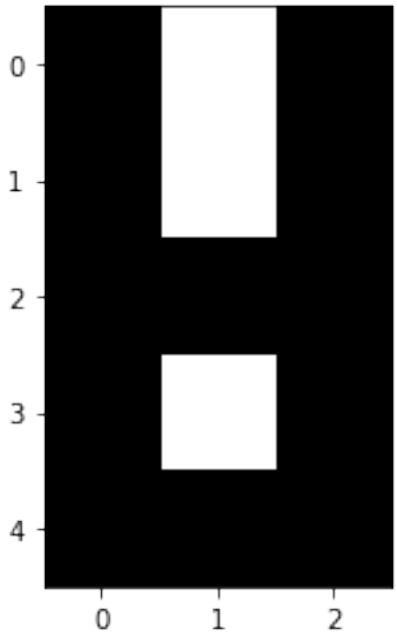


Figure 26: predicted image for input-3 with 20% loss in 5*3 size

[ ]:
```
input_1_lost_60 = showResultWithLost(input_1,0.6)
input_2_lost_60 = showResultWithLost(input_2,0.6)
input_3_lost_60 = showResultWithLost(input_3,0.6)
```

[ ]:
```
fig = plt.figure(figsize=(8, 8))
fig.add_subplot(1, 3, 1)
plt.imshow(input_1_lost_60.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('input_1 with lost')
fig.add_subplot(1, 3, 2)
plt.imshow(input_2_lost_60.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('input_2 with lost')
fig.add_subplot(1, 3, 3)
plt.imshow(input_3_lost_60.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('input_3 with lost')
plt.show()
```

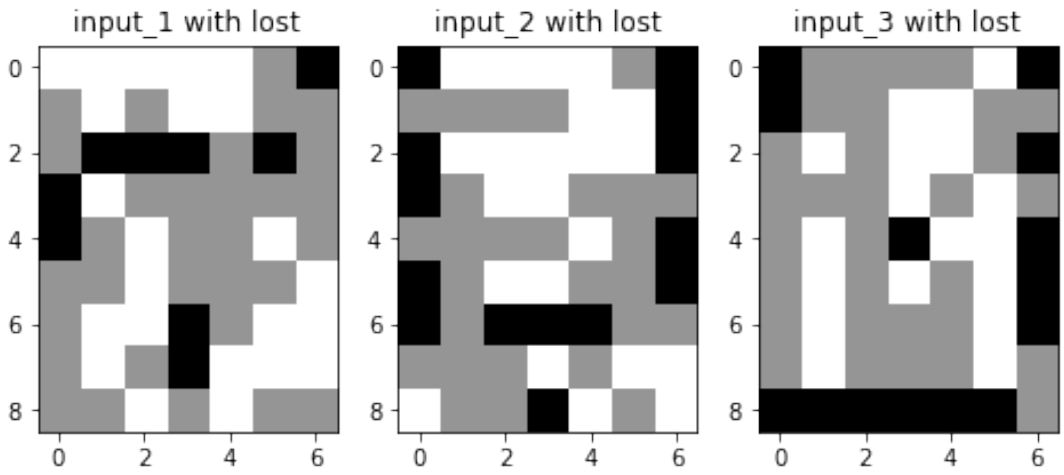


Figure 27: inputs images with 60% loss

The accuracy of model is 100%

```
[ ]: S = np.array([copy.deepcopy(input_1), copy.deepcopy(input_2), copy.
      ↪deepcopy(input_3)])
     T = np.array([copy.deepcopy(output_1), copy.deepcopy(output_2), copy.
      ↪deepcopy(output_3)])
     heb8 = Hebbian(S,T)
     heb8.train()
```

```
[ ]: heb8.showResult_1(input_1_lost_60)
     heb8.showResult_1(input_2_lost_60)
     heb8.showResult_1(input_3_lost_60)
```

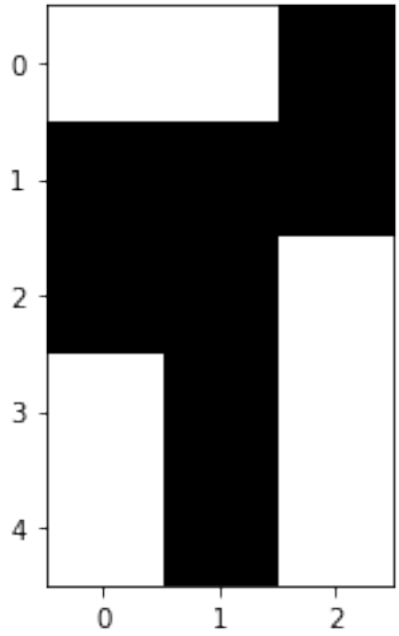


Figure 28: predicted image for input-1 with 60% loss in 5*3 size

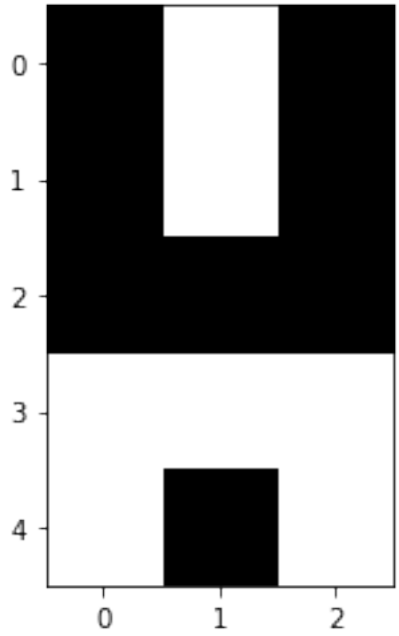


Figure 29: predicted image for input-2 with 60% loss in 5*3 size

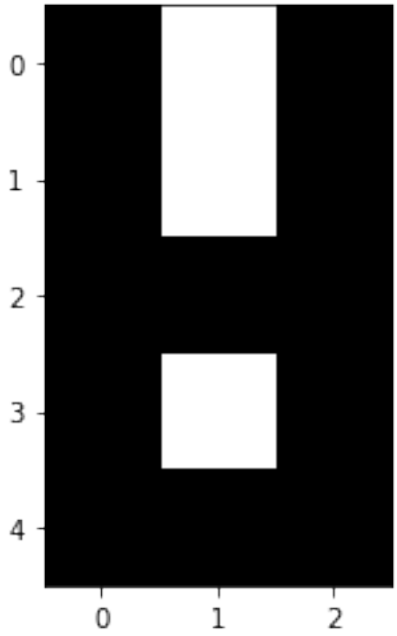


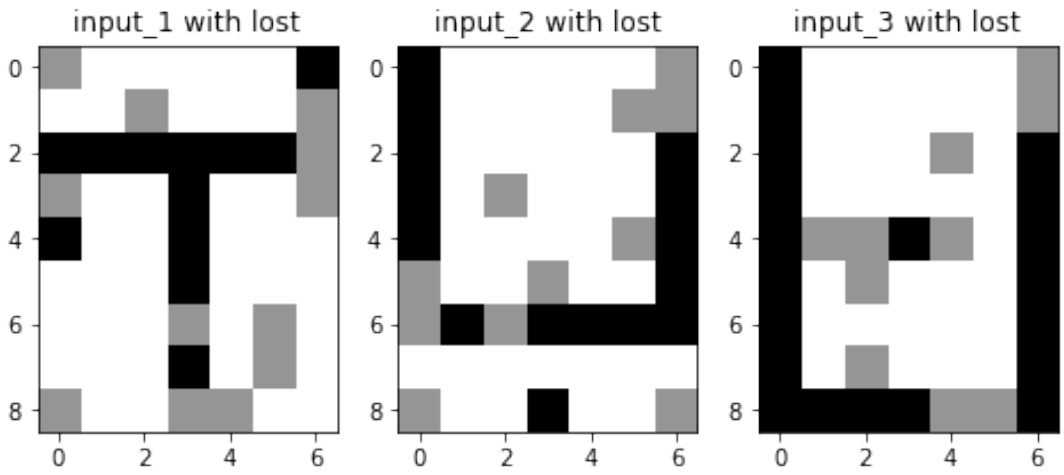
Figure 30: predicted image for input-3 with 60% loss in 5*3 size

Results of part 4's Inputs with 20% & 60% Lost

The accuracy of model for both (20 and 60) is 100%

[ ]:
```
input_1_lost_20 = showResultWithLost(input_1,0.2)
input_2_lost_20 = showResultWithLost(input_2,0.2)
input_3_lost_20 = showResultWithLost(input_3,0.2)
```

[ ]:
```
fig = plt.figure(figsize=(8, 8))
fig.add_subplot(1, 3, 1)
plt.imshow(input_1_lost_20.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('input_1 with lost')
fig.add_subplot(1, 3, 2)
plt.imshow(input_2_lost_20.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('input_2 with lost')
fig.add_subplot(1, 3, 3)
plt.imshow(input_3_lost_20.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('input_3 with lost')
plt.show()
```




```
[ ]: S = np.array([copy.deepcopy(input_1), copy.deepcopy(input_2), copy.
      ↪deepcopy(input_3)])
     T = np.array([copy.deepcopy(input_1_smallest_reshaped), copy.
      ↪deepcopy(input_2_smallest_reshaped), copy.deepcopy(input_3_smallest_reshaped)])
     heb9 = Hebbian(S,T)
     heb9.train()
```

```
[ ]: heb9.showResult_2(input_1_lost_20)
     heb9.showResult_2(input_2_lost_20)
     heb9.showResult_2(input_3_lost_20)
```

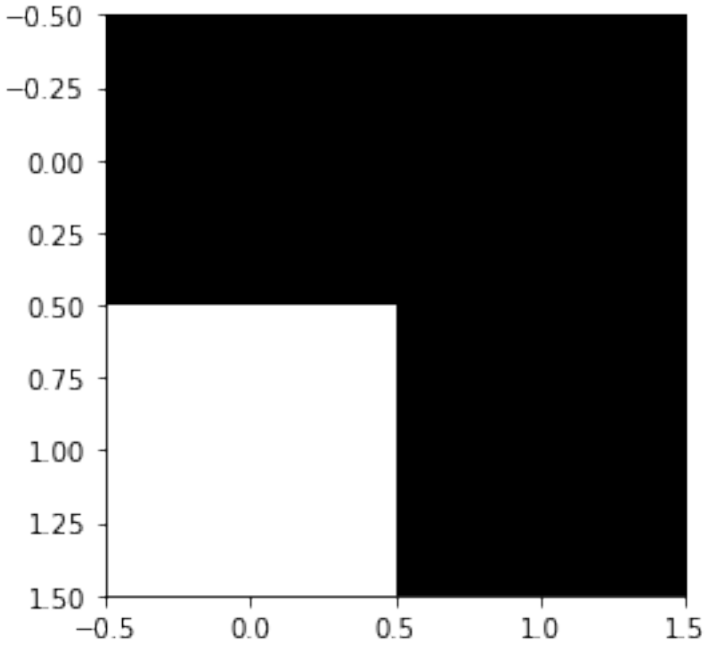


Figure 31: predicted image for input-1 with 20% loss in 2*2 size

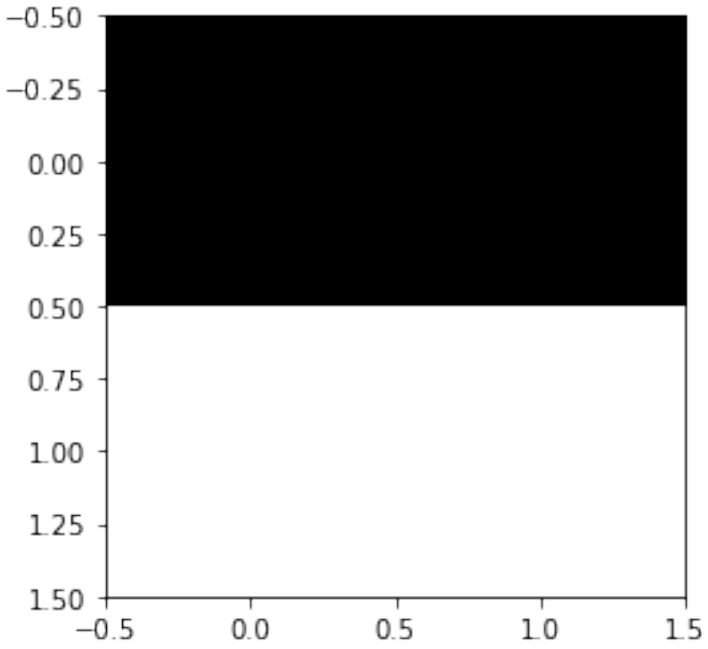


Figure 32: predicted image for input-2 with 20% loss in 2*2 size

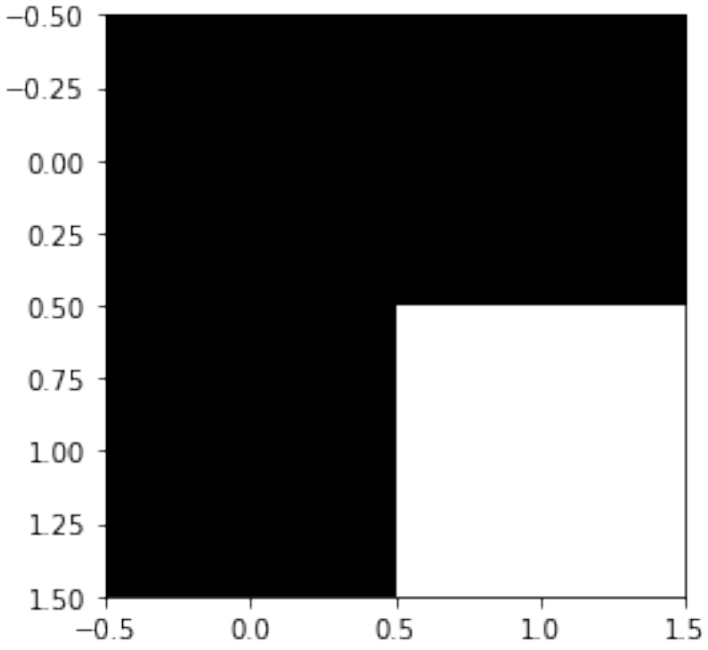
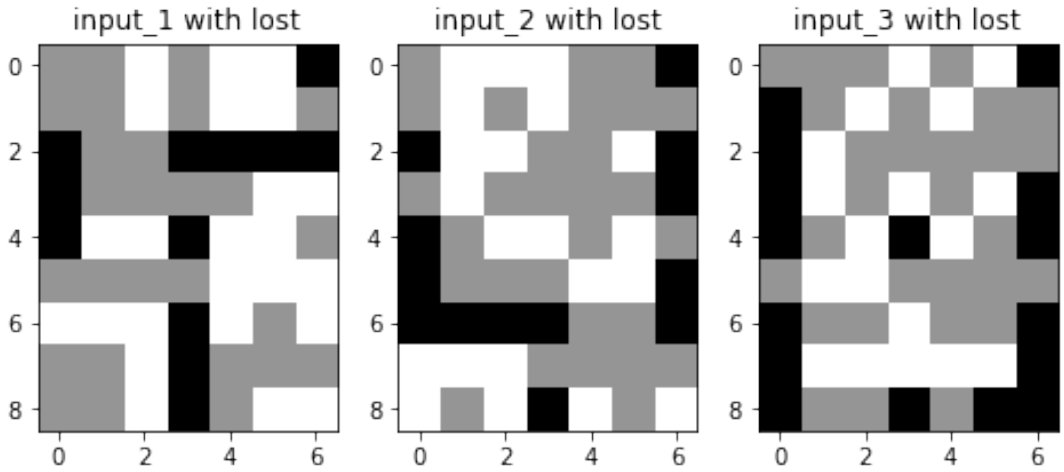


Figure 33: predicted image for input-3 with 20% loss in 2*2 size

[ ]:
```
input_1_lost_60 = showResultWithLost(input_1,0.6)
input_2_lost_60 = showResultWithLost(input_2,0.6)
input_3_lost_60 = showResultWithLost(input_3,0.6)
```

[ ]:
```
fig = plt.figure(figsize=(8, 8))
fig.add_subplot(1, 3, 1)
plt.imshow(input_1_lost_60.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('input_1 with lost')
fig.add_subplot(1, 3, 2)
plt.imshow(input_2_lost_60.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('input_2 with lost')
fig.add_subplot(1, 3, 3)
plt.imshow(input_3_lost_60.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('input_3 with lost')
plt.show()
```



[ ]:
```
S = np.array([copy.deepcopy(input_1), copy.deepcopy(input_2), copy.
 ↪deepcopy(input_3)])
T = np.array([copy.deepcopy(input_1_smallest_reshaped), copy.
 ↪deepcopy(input_2_smallest_reshaped), copy.deepcopy(input_3_smallest_reshaped)])
heb10 = Hebbian(S,T)
heb10.train()
```

[ ]:
```
heb10.showResult_2(input_1_lost_60)
heb10.showResult_2(input_2_lost_60)
heb10.showResult_2(input_3_lost_60)
```

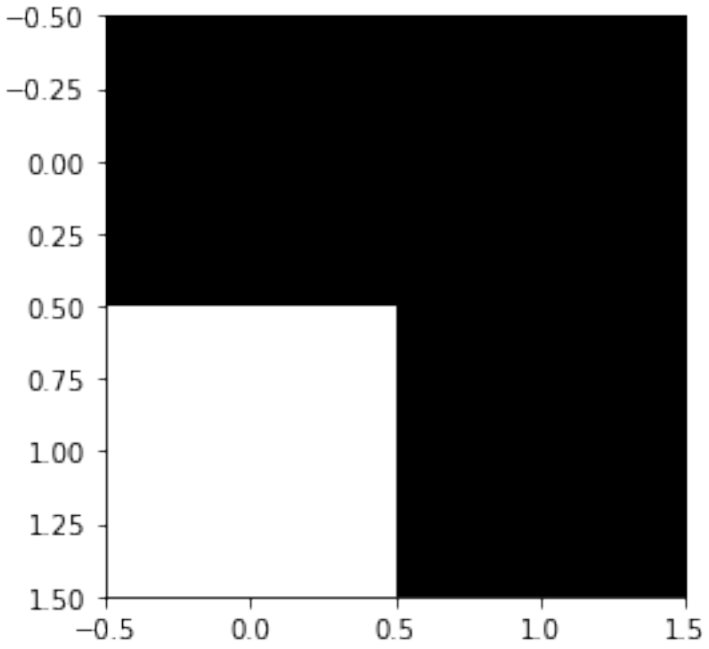


Figure 34: predicted image for input-1 with 60% loss in 2*2 size

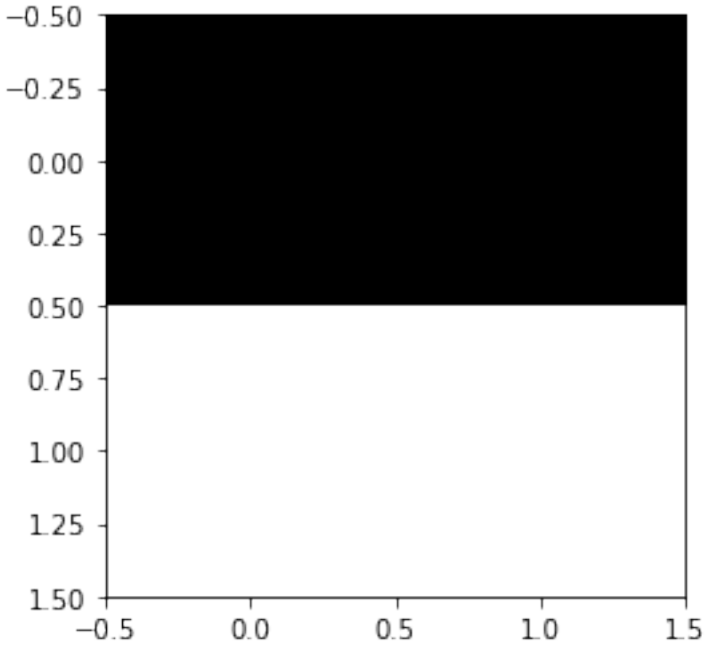


Figure 35: predicted image for input-2 with 60% loss in 2*2 size

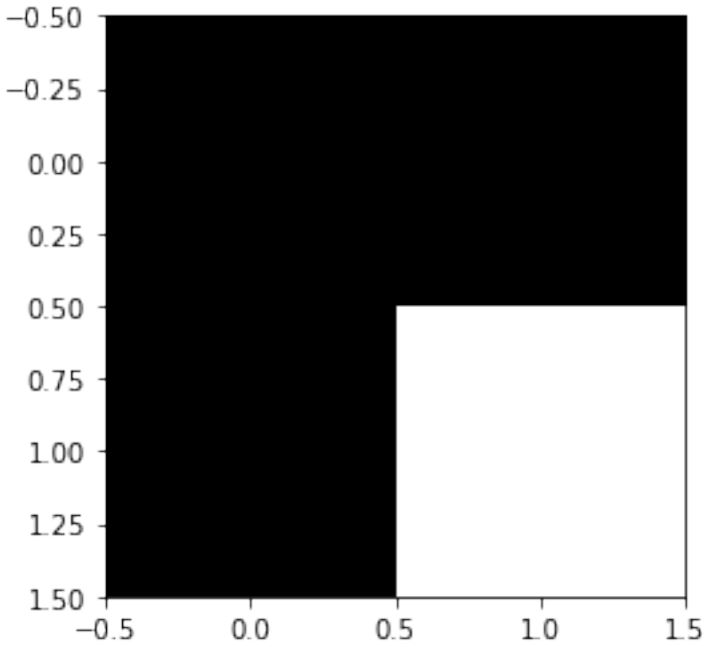


Figure 36: predicted image for input-3 with 60% loss in 2*2 size

Part 7

The model with 20% noise can predict properly for both inputs(2 and 4).

The accuracy of model whit 60% noise for part 2's input is 66.6% and for part 4's input is 33.3%.

The model with 20 and 60% information loss can predict properly for both inputs(2 and 4).

so the model is resistant to information loss than noise.

Reduce dimensions helps network resistance when losing information(to a point)

2 Question-2

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

[2]:
```
zero = np.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
])

one = np.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
])

two = np.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

])
```

[3]:
```
class Hebbian:
    def __init__(self, _S, _T):
        self.S = _S
        self.T = _T
        self.W = np.zeros((len(_S[1]), len(_T[1])))

    def train(self):
        for trainSample in range(len(self.S)):
            x = self.S[trainSample]
            y = self.T[trainSample]
            for i in range(len(x)):
                for j in range(len(y)):
                    self.W[i][j] = self.W[i][j] + x[i]*y[j]
    def showResult(self, X):
        y_t = np.dot(X,self.W)
        for i in range(len(y_t)): #sign
            if(y_t[i] >= 0):
                y_t[i] = 1
            else:
```

```
                y_t[i] = -1
        plt.figure()
        plt.imshow(y_t.reshape((7, 5)), cmap='binary')
```

[4]:
```
S = np.array([copy.deepcopy(zero), copy.deepcopy(one), copy.deepcopy(two)])
T = np.array([copy.deepcopy(zero), copy.deepcopy(one), copy.deepcopy(two)])
heb = Hebbian(S, T)
heb.train()
```

Part 1

Weights of Hebbian Learning Rule

[5]:
```
heb.W
```

[5]:
```
array([[ 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.]])
```

Part 2

The input and expected output image

[6]:
```
fig = plt.figure(figsize=(8, 8))
fig.add_subplot(1, 3, 1)
plt.imshow(zero.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('Zero')
fig.add_subplot(1, 3, 2)
plt.imshow(one.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('one')
fig.add_subplot(1, 3, 3)
plt.imshow(two.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('two')
plt.show()
```

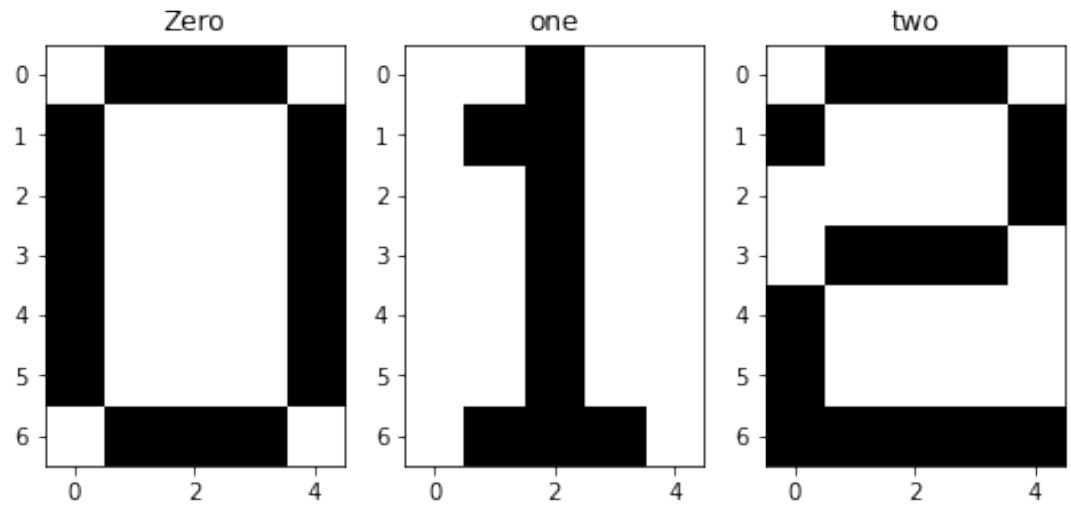


Figure 37: The input and expected output image

image of network output

The model accuracy is 100%

[7]:
```
heb.showResult(zero)
heb.showResult(one)
heb.showResult(two)
```

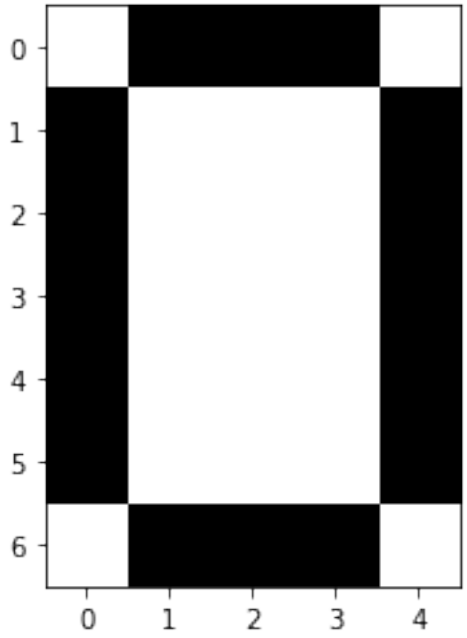


Figure 38: predicted image for zero

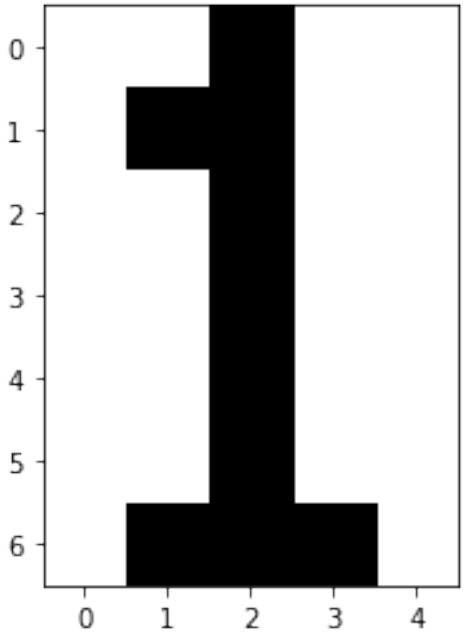


Figure 39: predicted image for one

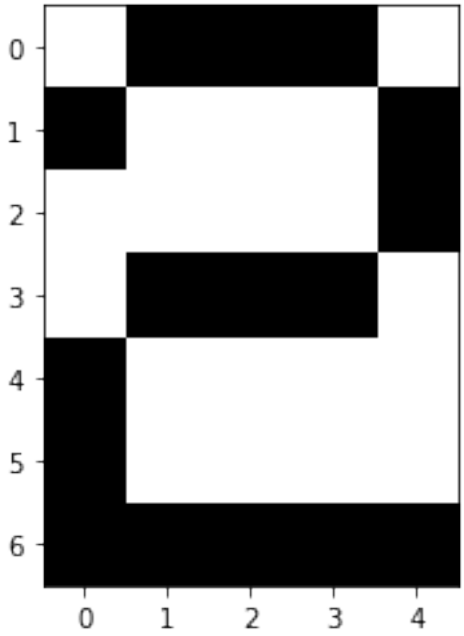


Figure 40: predicted image for two

Part 3

[8]:
```
def ResultWhitNoise(arr,percentage):
    pixels = int(np.prod(arr.shape)*percentage)
    with_noise = np.copy(arr)
    for pixel in range(pixels) :
      random_pixel = np.random.choice(np.prod(with_noise.shape), 1)
      if with_noise.flat[random_pixel] == -1 :
        with_noise.flat[random_pixel] = 1
      else:
        with_noise.flat[random_pixel] = -1

    return with_noise
```

[9]:
```
zero_noise_20 = ResultWhitNoise(zero,0.2)
one_noise_20  = ResultWhitNoise(one,0.2)
two_noise_20  = ResultWhitNoise(two,0.2)
```

The input image with 20% noise

[10]:
```
fig = plt.figure(figsize=(8, 8))
fig.add_subplot(1, 3, 1)
plt.imshow(zero_noise_20.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('zero with noise')
fig.add_subplot(1, 3, 2)
plt.imshow(one_noise_20.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('one with noise')
fig.add_subplot(1, 3, 3)
plt.imshow(two_noise_20.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('two with noise')
plt.show()
```

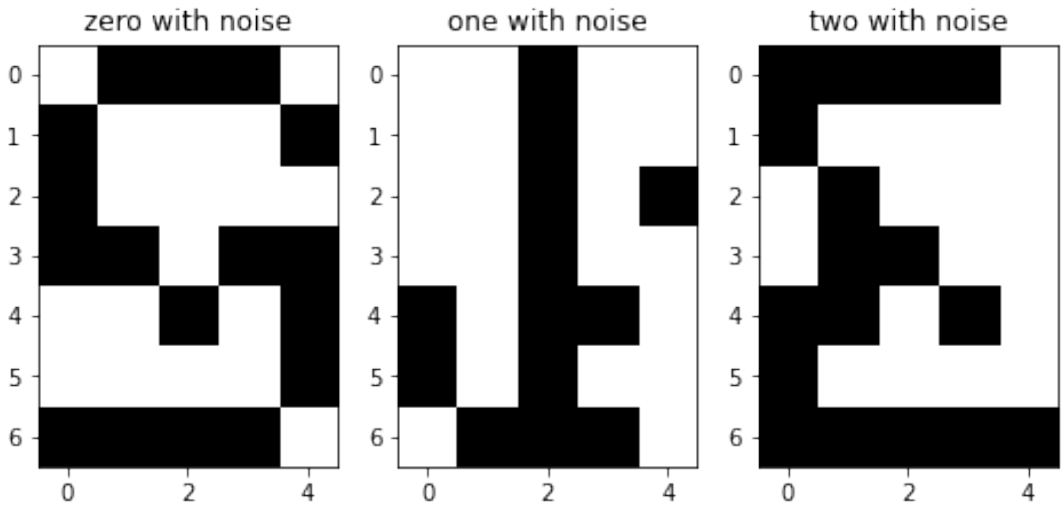


Figure 41: inputs with 20% noise

[11]:
```
S = np.array([copy.deepcopy(zero), copy.deepcopy(one), copy.deepcopy(two)])
T = np.array([copy.deepcopy(zero), copy.deepcopy(one), copy.deepcopy(two)])
heb2 = Hebbian(S, T)
heb2.train()
```

The output and expected output image with 20% noise

accuracy is 100%

[12]:
```
heb2.showResult(zero_noise_20)
heb2.showResult(one_noise_20)
heb2.showResult(two_noise_20)
```

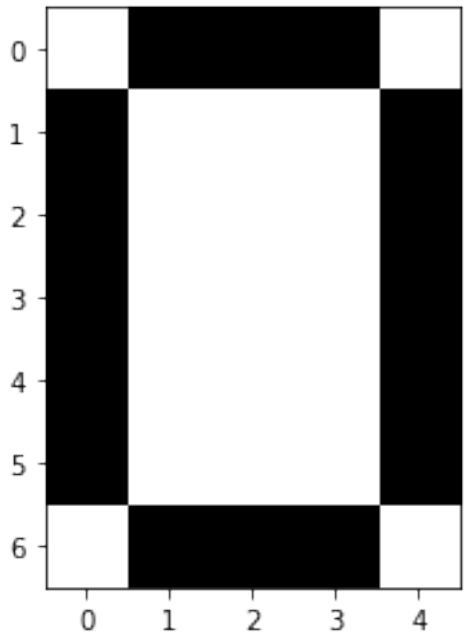


Figure 42: predicted image for zero with 20% noise

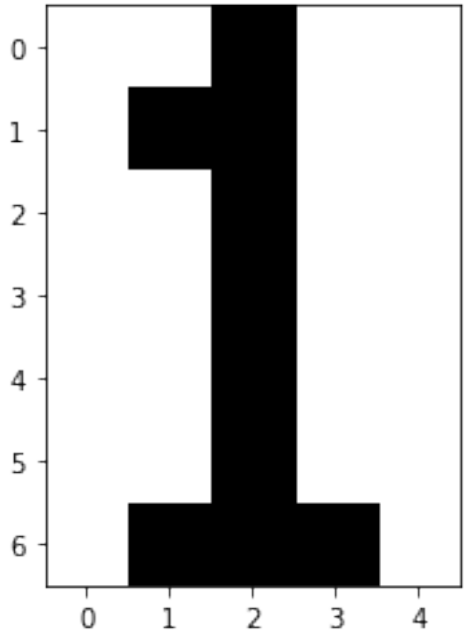


Figure 43: predicted image for one with 20% noise

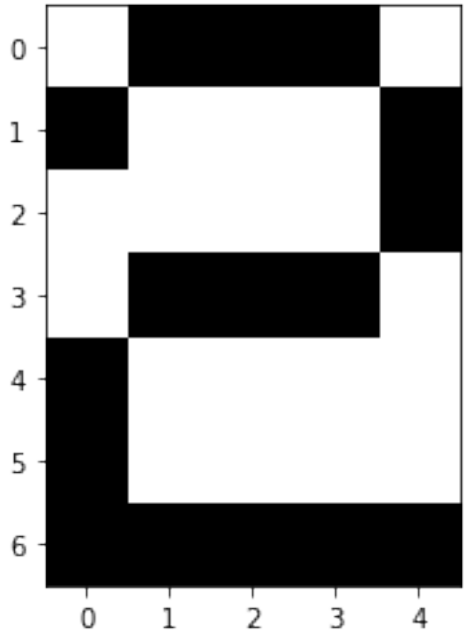


Figure 44: predicted image for two with 20% noise

[20]:
```
zero_noise_80 = ResultWhitNoise(zero,0.8)
one_noise_80  = ResultWhitNoise(one,0.8)
two_noise_80  = ResultWhitNoise(two,0.8)
```

The input image with 80% noise

[21]:
```
fig = plt.figure(figsize=(8, 8))
fig.add_subplot(1, 3, 1)
plt.imshow(zero_noise_80.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('zero with noise')
fig.add_subplot(1, 3, 2)
plt.imshow(one_noise_80.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('one with noise')
fig.add_subplot(1, 3, 3)
plt.imshow(two_noise_80.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('two with noise')
plt.show()
```

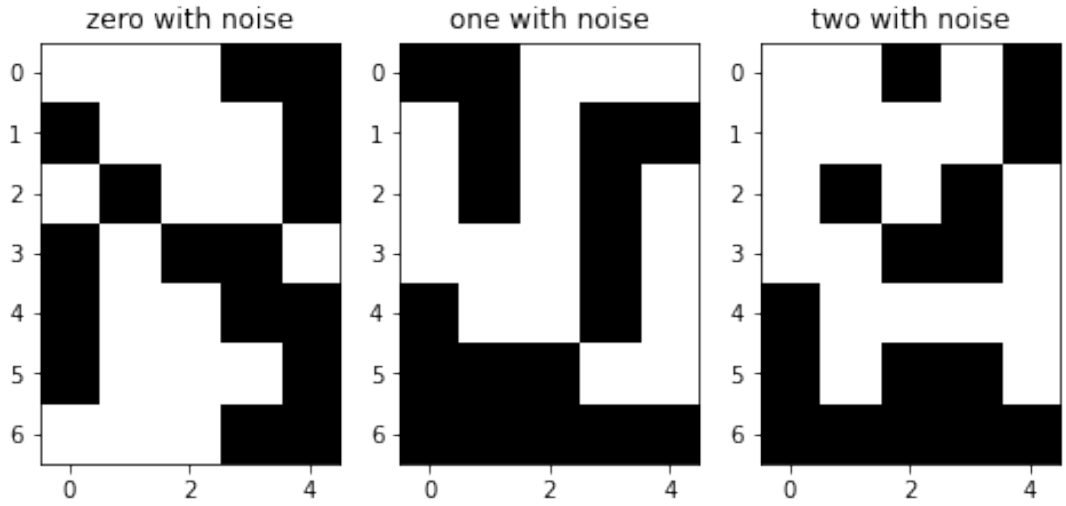


Figure 45: input images with 80% noise

The expected output image

[22]:
```
fig = plt.figure(figsize=(8, 8))
fig.add_subplot(1, 3, 1)
plt.imshow(zero.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('zero')
fig.add_subplot(1, 3, 2)
plt.imshow(one.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('one')
fig.add_subplot(1, 3, 3)
plt.imshow(two.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('two')
plt.show()
```

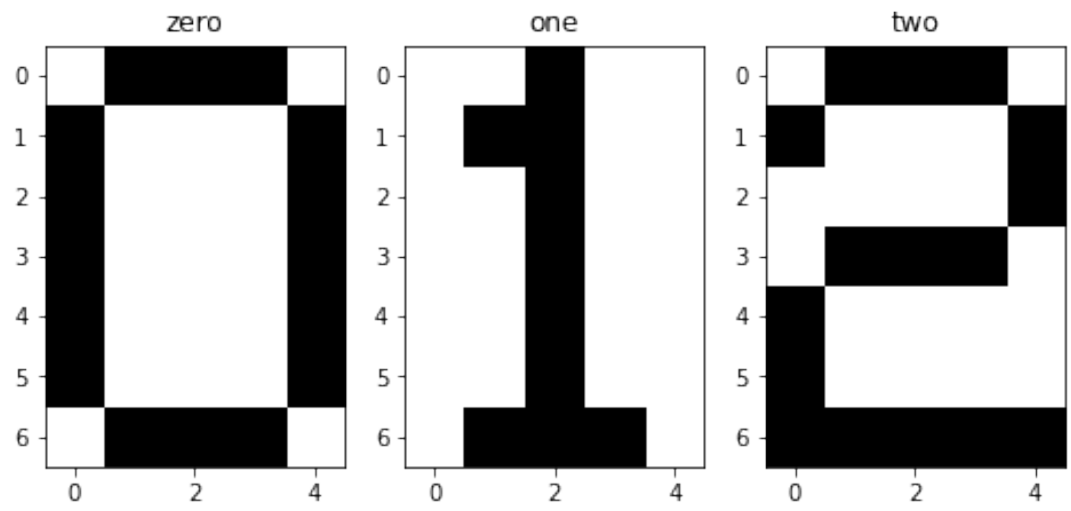


Figure 46: The expected output image

[23]:
```
S = np.array([copy.deepcopy(zero), copy.deepcopy(one), copy.deepcopy(two)])
T = np.array([copy.deepcopy(zero), copy.deepcopy(one), copy.deepcopy(two)])
heb3 = Hebbian(S, T)
heb3.train()
```

The network output image with 80% noise

accuracy is 33.3

[24]:
```
heb3.showResult(zero_noise_80)
heb3.showResult(one_noise_80)
heb3.showResult(two_noise_80)
```

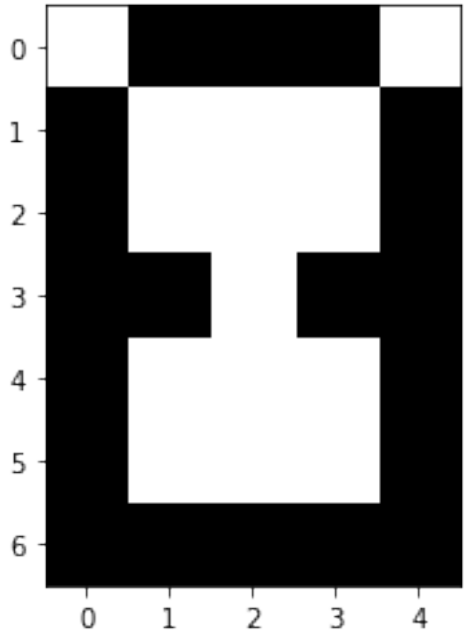


Figure 47: predicted image for zero with 80% noise

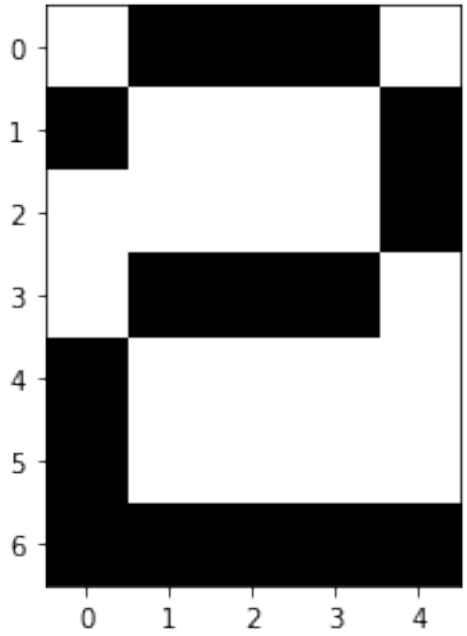


Figure 48: predicted image for one with 80% noise

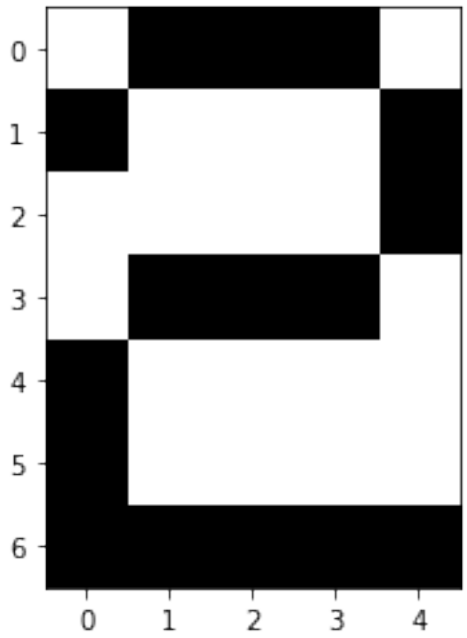


Figure 49: predicted image for two with 80% noise

Edit distance is a way of quantifying how dissimilar two strings (e.g., words) are to one another by counting the minimum number of operations required to transform one string into the other

Due to the function output and output from the network number one has the highest sensitivity among other numbers

[27]:
```
def edit_distance(list1, list2):
    difference = 0
    for i in range(len(list1)):
        if list1[i] != list2[i]:
            difference += 1

    return difference

print(edit_distance(zero,zero_noise_80 ))
print(edit_distance(one,one_noise_80 ))
print(edit_distance(two,two_noise_80 ))
```

```
12
18
10
```

Part 4

[28]:
```
def showResultWithLost(arr,percentage):
    pixels = int(np.prod(arr.shape)*percentage)
    with_noise = np.copy(arr)
    for pixel in range(pixels) :
      random_pixel = np.random.choice(np.prod(with_noise.shape), 1)
      with_noise.flat[random_pixel] = 0
    return with_noise
```

[29]:
```
zero_loss_20 = showResultWithLost(zero, 0.2)
one_loss_20 = showResultWithLost(one, 0.2)
two_loss_20 = showResultWithLost(two, 0.2)
```

The input image with 20% loss

[41]:
```
fig = plt.figure(figsize=(8, 8))
fig.add_subplot(1, 3, 1)
plt.imshow(zero_loss_20.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('zero_loss_20')
fig.add_subplot(1, 3, 2)
plt.imshow(one_loss_20.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('one_loss_20')
fig.add_subplot(1, 3, 3)
plt.imshow(two_loss_20.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('two_loss_20')
plt.show()
```

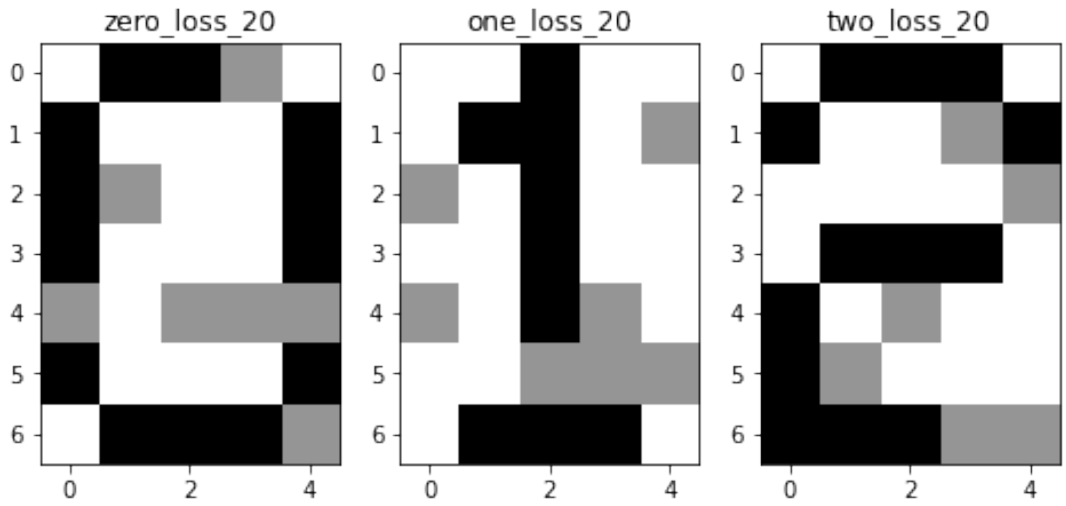


Figure 50: input images with 20% loss

The expected output image

[31]:
```
fig = plt.figure(figsize=(8, 8))
fig.add_subplot(1, 3, 1)
plt.imshow(zero.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('zero')
fig.add_subplot(1, 3, 2)
plt.imshow(one.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('one')
fig.add_subplot(1, 3, 3)
plt.imshow(two.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('two')
plt.show()
```

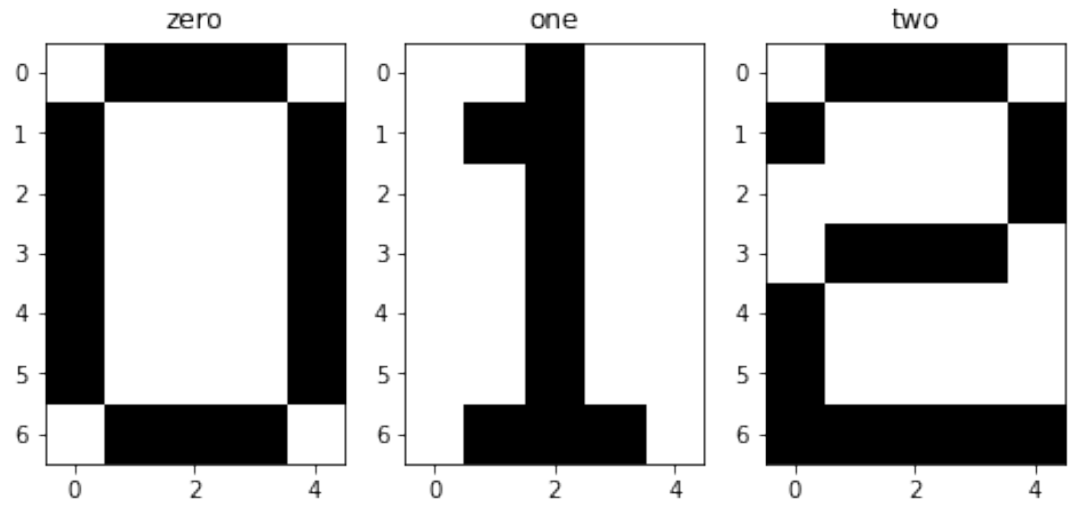


Figure 51: The expected output images

[32]:
```
S = np.array([copy.deepcopy(zero), copy.deepcopy(one), copy.deepcopy(two)])
T = np.array([copy.deepcopy(zero), copy.deepcopy(one), copy.deepcopy(two)])
heb4 = Hebbian(S, T)
heb4.train()
```

The network output image

accuracy is 100%

[33]:
```
heb4.showResult(zero_loss_20)
heb4.showResult(one_loss_20)
heb4.showResult(two_loss_20)
```

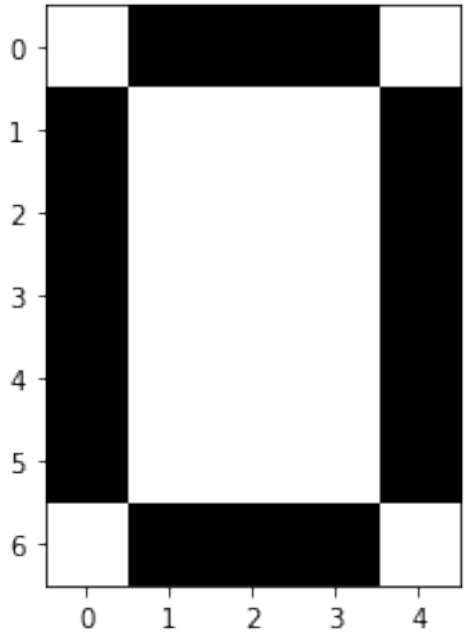


Figure 52: predicted image for zero with 20% loss

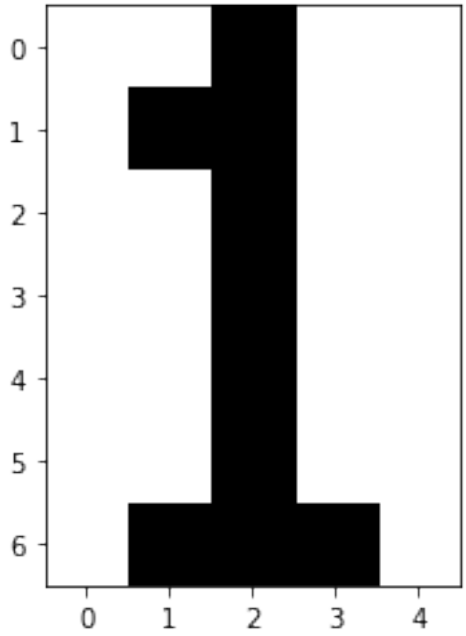


Figure 53: predicted image for one with 20% loss

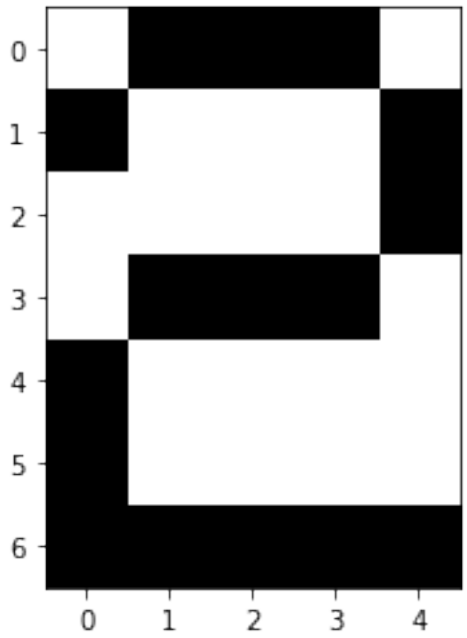


Figure 54: predicted image for two with 20% loss

[42]:
```
zero_loss_80 = showResultWithLost(zero, 0.8)
one_loss_80 = showResultWithLost(one, 0.8)
two_loss_80 = showResultWithLost(two, 0.8)
```

The input image with 80% loss

[37]:
```
fig = plt.figure(figsize=(8, 8))
fig.add_subplot(1, 3, 1)
plt.imshow(zero_loss_80.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('zero_loss_80')
fig.add_subplot(1, 3, 2)
plt.imshow(one_loss_80.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('one_loss_80')
fig.add_subplot(1, 3, 3)
plt.imshow(two_loss_80.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('two_loss_80')
plt.show()
```

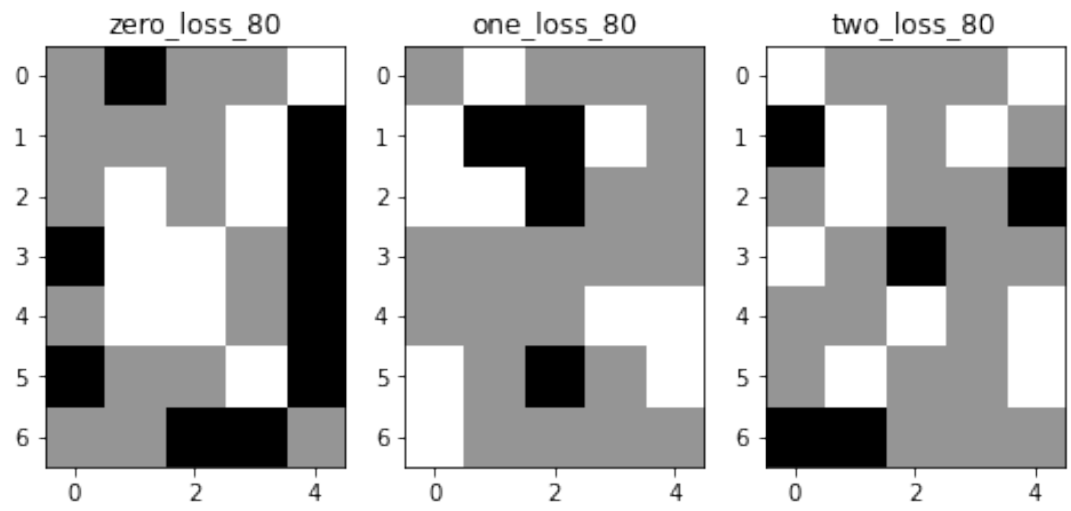


Figure 55: input images with 80% loss

The expected output image

[38]:
```
fig = plt.figure(figsize=(8, 8))
fig.add_subplot(1, 3, 1)
plt.imshow(zero.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('zero')
fig.add_subplot(1, 3, 2)
plt.imshow(one.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('one')
fig.add_subplot(1, 3, 3)
plt.imshow(two.reshape((7, 5)), interpolation='nearest',cmap='Greys')
plt.title('two')
plt.show()
```

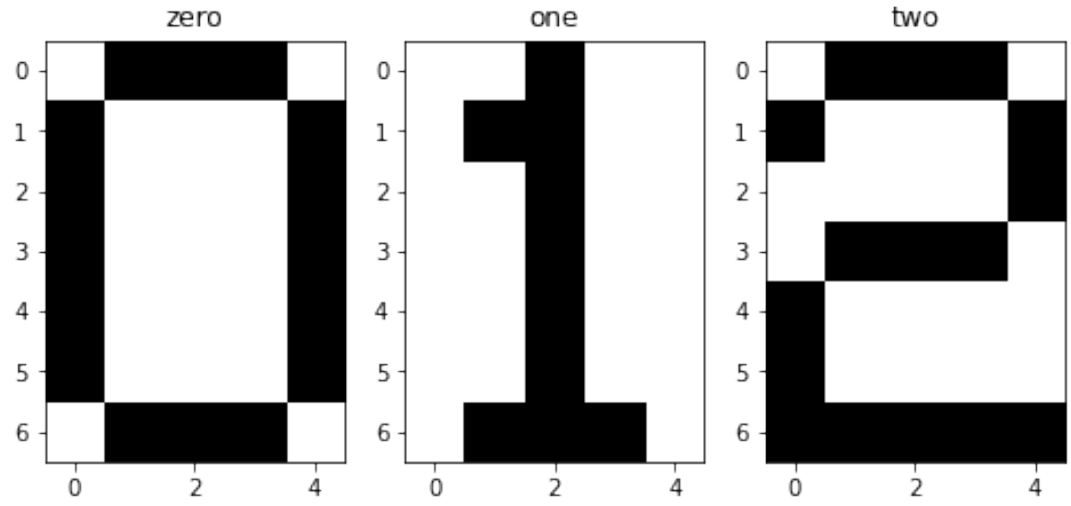


Figure 56: expected output image

[43]:
```
S = np.array([copy.deepcopy(zero), copy.deepcopy(one), copy.deepcopy(two)])
T = np.array([copy.deepcopy(zero), copy.deepcopy(one), copy.deepcopy(two)])
heb5 = Hebbian(S, T)
heb5.train()
```

The image of network output

accuracy is 100%

[44]:
```
heb5.showResult(zero_loss_80)
heb5.showResult(one_loss_80)
heb5.showResult(two_loss_80)
```

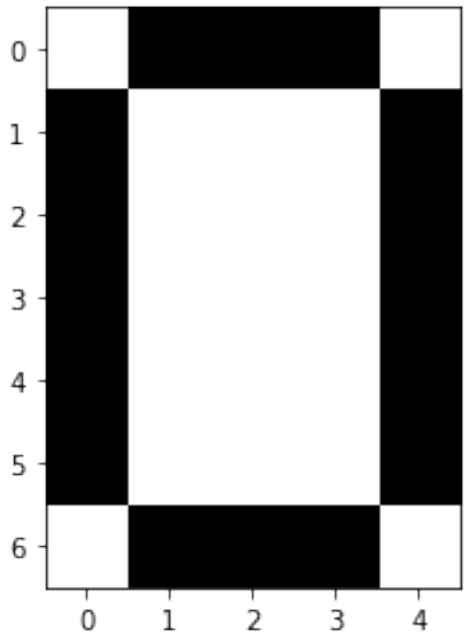


Figure 57: predicted image for zero with 80% loss

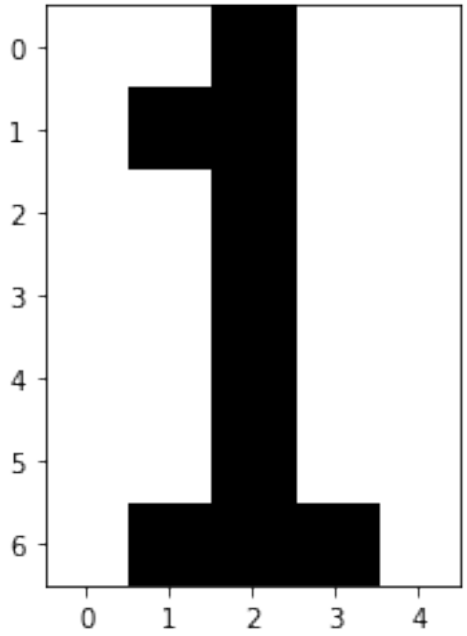


Figure 58: predicted image for one with 80% loss

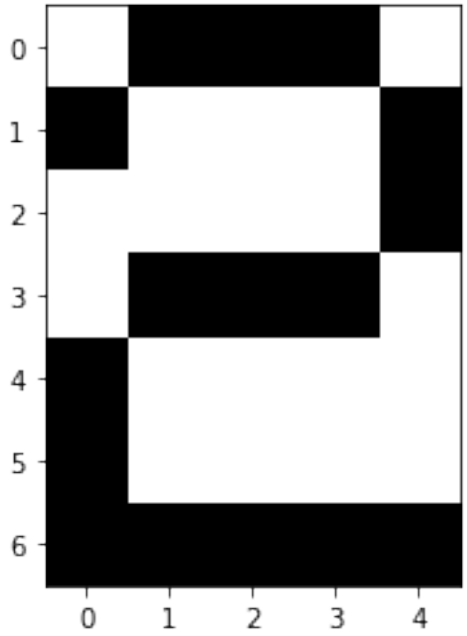


Figure 59: predicted image for two with 80% loss

3 Question-3

4 part 1

5 Discrete Hopfield Network:

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:

1. Binary (0/1)
2. 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

6 Structure & Architecture

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

7 Training Algorithm

For storing a set of input patterns S(p) [p = 1 to P], where S(p) = S1(p) ... Si(p) ... Sn(p), the weight matrix is given by:

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

$$w(ij) = [2si(p)1][2sj(p)1] \quad \text{for} ij$$

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

$$w(ij) = [si(p)][sj(p)] \quad \text{for} ij$$

[1]:
```
from google.colab import drive

drive.mount('/content/gdrive')
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call
drive.mount("/content/gdrive", force_remount=True).
```

[2]:
```
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
```

8 part 2

Reading images and changing size to 64*64

[3]:
```
Test_1 = Image.open(r"/content/gdrive/MyDrive/test1.png").convert(mode="L").
 ↪resize(size =(64,64))
Test_2 = Image.open(r"/content/gdrive/MyDrive/test2.png").convert(mode="L").
 ↪resize(size =(64,64))
Test_3 = Image.open(r"/content/gdrive/MyDrive/test3.png").convert(mode="L").
 ↪resize(size =(64,64))
Train = Image.open(r"/content/gdrive/MyDrive/train.jpg").convert(mode="L").
 ↪resize(size =(64,64))
```

bipolar function

[4]:
```
def bipolar(image):
  threshold = 120
  image_array = np.asarray(image,dtype=np.uint8)
  x = np.zeros(image_array.shape,dtype=np.float)
  x[image_array > threshold] = 1
  x[image_array < threshold] = -1
  return x
```

[5]:
```
X_train =  bipolar(Train)
X_test_1 = bipolar(Test_1)
X_test_2 = bipolar(Test_2)
X_test_3 = bipolar(Test_3)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:4:
DeprecationWarning: `np.float` is a deprecated alias for the builtin `float`. To
silence this warning, use `float` by itself. Doing this will not modify any
behavior and is safe. If you specifically wanted the numpy scalar type, use
`np.float64` here.
Deprecated in NumPy 1.20; for more details and guidance:
https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  after removing the cwd from sys.path.
```

plotting train and test images in bipolar form

[6]:
```
plt.imshow(np.repeat(X_train[:,:,np.newaxis], repeats=3, axis=2))
plt.title("train image")
plt.show()
plt.imshow(np.repeat(X_test_1[:,:,np.newaxis], repeats=3, axis=2))
plt.title("test_1 image")
plt.show()
plt.imshow(np.repeat(X_test_2[:,:,np.newaxis], repeats=3, axis=2))
plt.title("test_2 image")
plt.show()
plt.imshow(np.repeat(X_test_3[:,:,np.newaxis], repeats=3, axis=2))
plt.title("test_3 image")
plt.show()
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

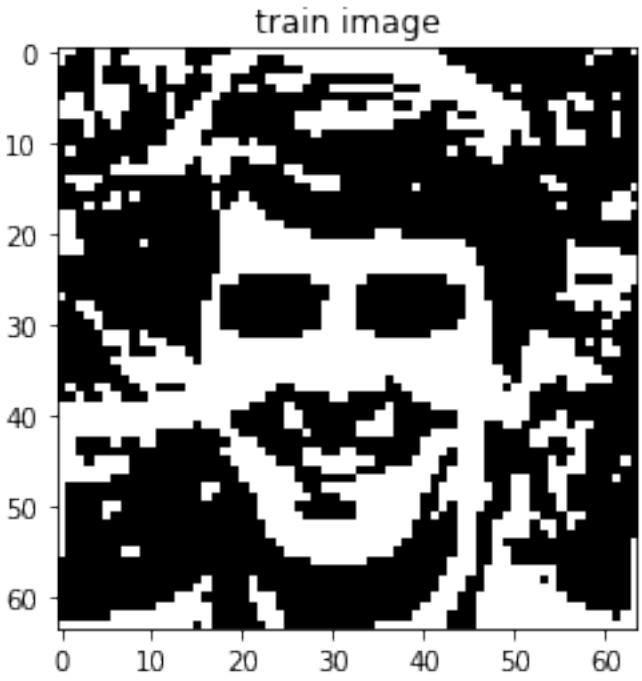


Figure 60: The train image, bipolarized

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

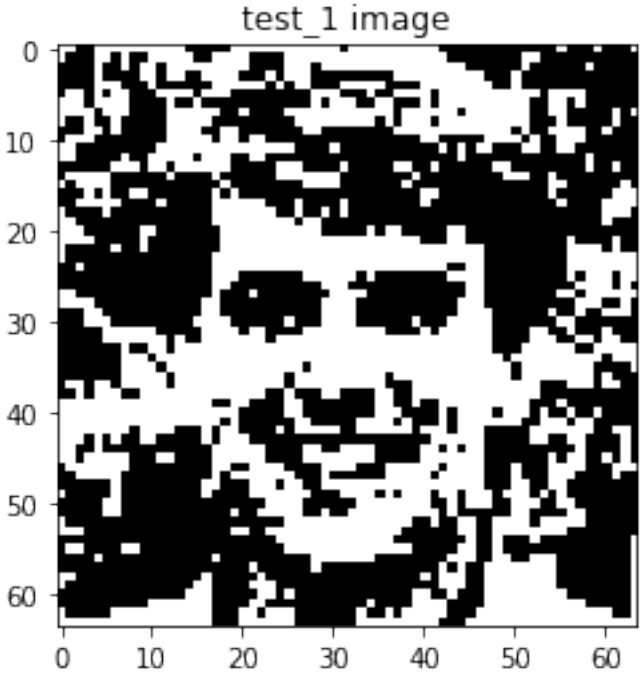


Figure 61: The test image-1, bipolarized

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

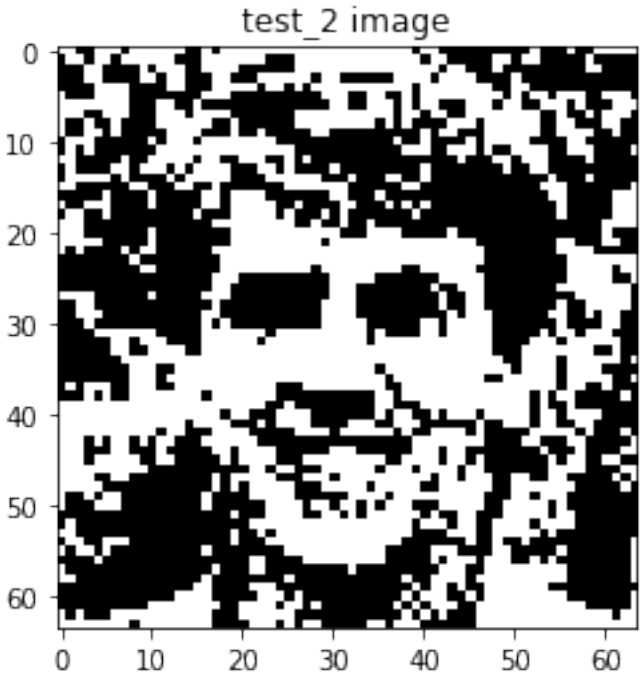


Figure 62: The test image-2, bipolarized

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

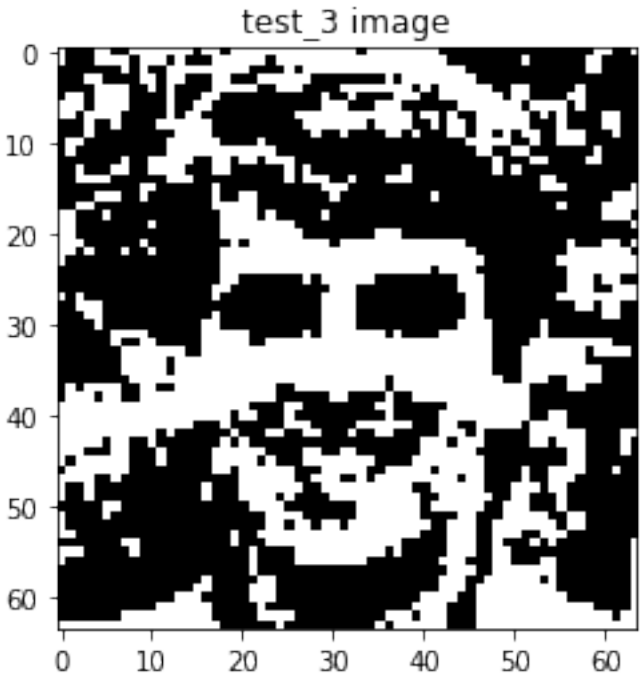


Figure 63: The test image-3, bipolarized

[7]:
```
class DiscreteHopfieldNet():

      def __init__(self):
        self.weight = None

      def train(self,data):
        a = data[0]
        a = np.expand_dims(a, 1)
        self.weight = np.matmul(a, a.T)
        for i in range(len(a)):
            self.weight[i][i] = 0


      def Activation(self,yi, theta=0):
           if yi > theta:
             return 1
           else:
             return -1

      def predict(self, input_image, weights):
        history = np.zeros((input_image.shape[0]))
        y = input_image
        iter_times = 50
        random_indices = np.random.permutation(np.arange(len(input_image)))
        for i in range(iter_times):
           for index in random_indices:
              y[index] = y[index] + np.dot(y, weights[index])
              y[index] = self.Activation(y[index])
        history = y
        return history
```

9 part 3

Matrix of weights

[8]:
```
model = DiscreteHopfieldNet()
```

[9]:
```
model.train([X_train.reshape(-1)])
weights = model.weight
weights
```

[9]:
```
array([[ 0.,  1., -1., ...,  1.,  1.,  1.],
       [ 1.,  0., -1., ...,  1.,  1.,  1.],
       [-1., -1.,  0., ..., -1., -1., -1.],
       ...,
       [ 1.,  1., -1., ...,  0.,  1.,  1.],
       [ 1.,  1., -1., ...,  1.,  0.,  1.],
       [ 1.,  1., -1., ...,  1.,  1.,  0.]])
```

10 part 4

predict function evaluation

[10]:
```
pred_X_train =  model.predict(X_train.reshape(-1), weights)
pred_X_test_1 = model.predict(X_test_1.reshape(-1), weights)
pred_X_test_2 = model.predict(X_test_2.reshape(-1), weights)
```

```
pred_X_test_3 = model.predict(X_test_3.reshape(-1), weights)
```

[11]:
```
plt.imshow(pred_X_test_1.reshape(64, 64), cmap=plt.cm.gray)
plt.show()
plt.imshow(pred_X_test_2.reshape(64, 64), cmap=plt.cm.gray)
plt.show()
plt.imshow(pred_X_test_3.reshape(64, 64), cmap=plt.cm.gray)
plt.show()
```

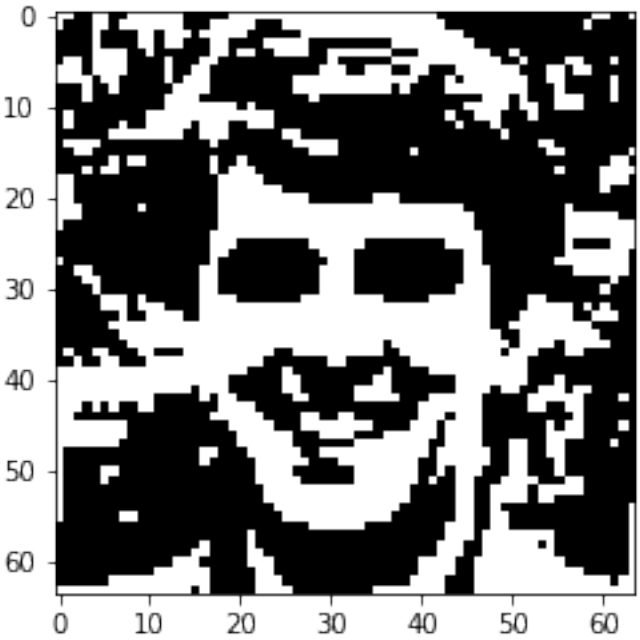


Figure 64: predicted image for Test-1

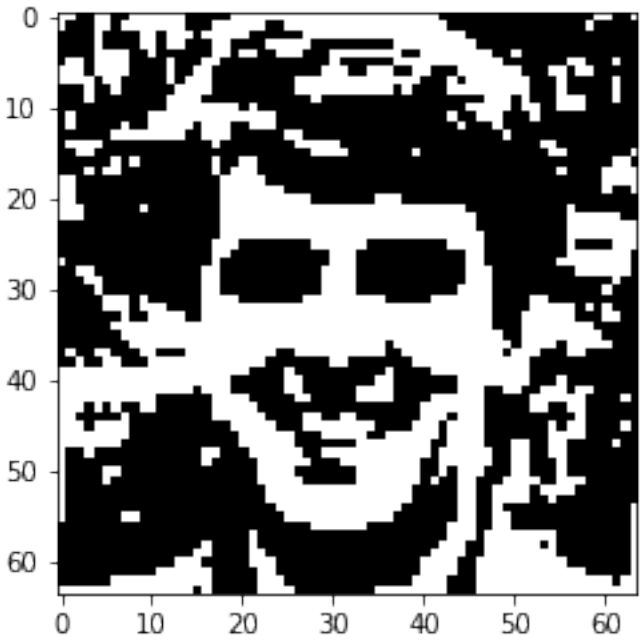


Figure 65: predicted image for Test-2

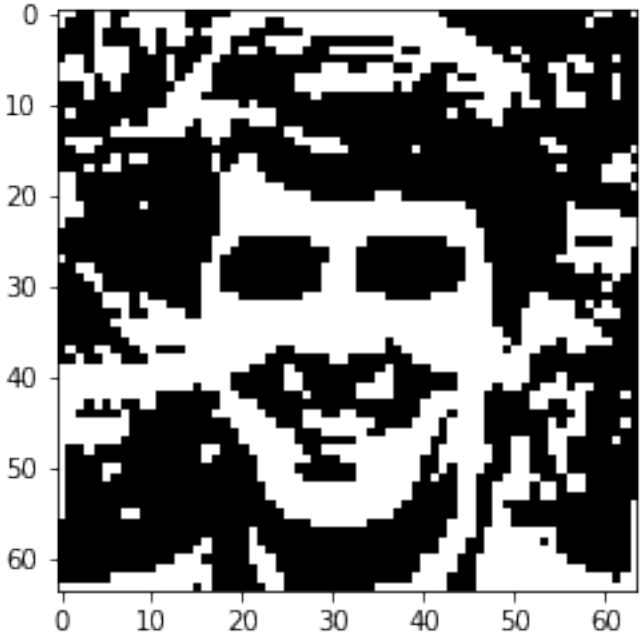


Figure 66: predicted image for Test-3

11 part 5

[18]:
```
def compute_hamming_distance(a, b):
    return len(np.where(a != b)[0])
```

[19]:
```
test1_hamming = compute_hamming_distance(pred_X_test_1, pred_X_train)
test2_hamming = compute_hamming_distance(pred_X_test_2, pred_X_train)
test3_hamming = compute_hamming_distance(pred_X_test_3, pred_X_train)
```

[20]:
```
print(test1_hamming)

print(test2_hamming)

print(test3_hamming)
```

```
44
43
33
```

12 Question-4

[1]:
```
import numpy as np
import pandas as pd
import math
```

[2]:
```
X_Clinton = np.array([
                    1,0,0,0,0,1,1,
                    1,1,0,1,1,0,0,
                    1,1,0,1,0,0,1,
                    1,1,0,1,1,1,0,
                    1,1,1,0,1,0,0,
                    1,1,0,1,1,1,1,
                    1,1,0,1,1,1,0
])

X_Hillary = np.array([
                    1,0,0,1,0,0,0,
                    1,1,0,1,0,0,1,
                    1,1,0,1,1,0,0,
                    1,1,0,1,1,0,0,
                    1,1,0,0,0,0,1,
                    1,1,1,0,0,1,0,
                    1,1,1,1,0,0,1

])

X_Kenstar = np.array([
                    1,0,0,1,0,1,1,
                    1,1,0,0,1,0,1,
                    1,1,0,1,1,1,0,
                    1,1,1,0,0,1,1,
                    1,1,1,0,1,0,0,
                    1,1,0,0,0,0,1,
                    1,1,1,0,0,1,0
])

Y_Clinton = np.array([
                      1,0,1,0,0,0,0,
                      1,1,1,0,0,1,0,
                      1,1,0,0,1,0,1,
                      1,1,1,0,0,1,1,
                      1,1,0,1,0,0,1,
                      1,1,0,0,1,0,0,
                      1,1,0,0,1,0,1,
                      1,1,0,1,1,1,0,
                      1,1,1,0,1,0,0
])

Y_Hillary = np.array([
                      1,0,0,0,1,1,0,
                      1,1,0,1,0,0,1,
                      1,1,1,0,0,1,0,
                      1,1,1,0,0,1,1,
                      1,1,1,0,1,0,0,
                      1,0,0,1,1,0,0,
                      1,1,0,0,0,0,1,
```

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

Y_Kenstar = np.array([
                    1,0,0,0,1,1,1,
                    1,1,0,0,1,0,1,
                    1,1,0,1,1,1,0,
                    1,1,1,0,1,0,0,
                    1,1,0,1,1,0,0,
                    1,1,0,0,1,0,1,
                    1,1,0,1,1,0,1,
                    1,1,0,0,0,0,1,
                    1,1,0,1,1,1,0
])
```

[3]:
```
class BAM() :

    def __init__(self, in_features, out_features, seed = 42) :
        np.random.seed(seed)

        self.in_features = in_features
        self.out_features = out_features
        self.init_wad()


    def init_wad(self) :
        self.weights = np.zeros(shape=(self.in_features*self.out_features,)).
↪reshape(-1,self.out_features)

    #bipolar
    def h(self,num,threshold = 0.25) :

        if num > threshold :
            return 1
        elif num < threshold:
            return 0
        else :
            return threshold
    # X---> Y
    def forward(self, x):
        net = np.matmul(x,self.weights)
        prediction = pd.Series(net)
        prediction = (prediction - np.mean(prediction)) / np.std(prediction)
        prediction = prediction.apply(self.h).values
        return prediction
    # Y---> X
    def backward(self,y) :
        net = np.matmul(y,self.weights.T)
        prediction = pd.Series(net)
        prediction = prediction.apply(self.h).values
        return prediction

    def calculate_new_weights(self,x,y) :
        new_weight = np.matmul(x.reshape(-1,1),y.reshape(1,-1))
        self.weights = self.weights + new_weight
```

```
    def initialize_weights(self,X_train,y_train,epochs = 1) :
        for epoch in range(0,epochs):

            for i in range(len(X_train)) :

                self.calculate_new_weights(X_train[i],y_train[i])
```

[4]:
```
model = BAM(in_features=49,out_features=63)
```

[5]:
```
X_train = np.concatenate([X_Clinton.reshape(1,49), X_Hillary.reshape(1,49),␣
↪X_Kenstar.reshape(1,49)], axis=0)
y_train = np.concatenate([Y_Clinton.reshape(1,63), Y_Hillary.reshape(1,63),␣
↪Y_Kenstar.reshape(1,63)], axis=0)
```

13 Part 1

matrix of weights

[6]:
```
model.initialize_weights(X_train, y_train,epochs=1)
model.weights
```

[6]:
```
array([[3., 0., 1., ..., 2., 1., 1.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [1., 0., 1., ..., 1., 0., 0.],
       [2., 0., 1., ..., 2., 1., 0.],
       [1., 0., 0., ..., 0., 0., 1.]])
```

14 Part 2

15 Model evaluation (direction)

[7]:
```
y_1 = model.forward(X_train[0])
y_2 = model.forward(X_train[1])
y_3 = model.forward(X_train[2])
```

[8]:
```
Accuracy_Clinton = np.sum(y_train[0] == y_1)/63
Accuracy_Clinton
```

[8]:
```
0.8888888888888888
```

[9]:
```
Accuracy_Hillary = np.sum(y_train[1] == y_2)/63
Accuracy_Hillary
```

[9]:
```
0.9206349206349206
```

[10]:
```
Accuracy_Kenstar = np.sum(y_train[2] == y_3)/63
Accuracy_Kenstar
```

[10]:
```
0.873015873015873
```

16 Model evaluation(Reverse)

[11]:
```
x_1 = model.backward(y_train[0])
x_2 = model.backward(y_train[1])
x_3 = model.backward(y_train[2])
```

[12]:
```
Accuracy_President = np.sum(X_train[0] == x_1)/49
Accuracy_President
```

[12]: 0.7959183673469388

[13]:
```
Accuracy_FirstLady = np.sum(X_train[1] == x_2)/49
Accuracy_FirstLady
```

[13]: 0.6938775510204082

[14]:
```
Accuracy_Gentelman = np.sum(X_train[2] == x_3)/49
Accuracy_Gentelman
```

[14]: 0.7551020408163265

17 Part 3

[15]:
```
def ShowResultWithNoise(arr,percentage):
    pixels = int(np.prod(arr.shape)*percentage)
    with_noise = np.copy(arr)
    for pixel in range(pixels) :
      random_pixel = np.random.choice(np.prod(with_noise.shape), 1)
      if with_noise.flat[random_pixel] == -1 :
        with_noise.flat[random_pixel] = 1
      else:
        with_noise.flat[random_pixel] = 1

    return with_noise
```

[16]:
```
X_Clinton_10_noised = ShowResultWithNoise(X_Clinton, 0.1)
X_Hillary_10_noised = ShowResultWithNoise(X_Hillary, 0.1)
X_Kenstar_10_noised = ShowResultWithNoise(X_Kenstar, 0.1)
Y_Clinton_10_noised = ShowResultWithNoise(Y_Clinton, 0.1)
Y_Hillary_10_noised = ShowResultWithNoise(Y_Hillary, 0.1)
Y_Kenstar_10_noised = ShowResultWithNoise(Y_Kenstar, 0.1)
```

18 Model evaluation with 10% noise(direction)

[17]:
```
X_train = np.concatenate([X_Clinton_10_noised.reshape(1,49), X_Hillary_10_noised.
 ↪reshape(1,49), X_Kenstar_10_noised.reshape(1,49)], axis=0)
y_train = np.concatenate([Y_Clinton.reshape(1,63), Y_Hillary.reshape(1,63),␣
 ↪Y_Kenstar.reshape(1,63)], axis=0)
```

[18]:
```
y_1 = model.forward(X_train[0])
y_2 = model.forward(X_train[1])
y_3 = model.forward(X_train[2])
```

[19]:
```
Accuracy_Clinton = np.sum(y_train[0] == y_1)/63
Accuracy_Clinton
```

[19]: 0.8888888888888888

[20]:
```
Accuracy_Hillary = np.sum(y_train[1] == y_2)/63
Accuracy_Hillary
```

[20]: 0.9206349206349206

[21]:
```
Accuracy_Kenstar = np.sum(y_train[2] == y_3)/63
Accuracy_Kenstar
```

[21]: 0.873015873015873

19 Model evaluation with 10% noise(Reverse)

[22]:
```
X_train = np.concatenate([X_Clinton.reshape(1,49), X_Hillary.reshape(1,49),␣
↪X_Kenstar.reshape(1,49)], axis=0)
y_train = np.concatenate([Y_Clinton_10_noised.reshape(1,63), Y_Hillary_10_noised.
↪reshape(1,63), Y_Kenstar_10_noised.reshape(1,63)], axis=0)
```

[23]:
```
x_1 = model.backward(y_train[0])
x_2 = model.backward(y_train[1])
x_3 = model.backward(y_train[2])
```

[24]:
```
Accuracy_President = np.sum(X_train[0] == x_1)/49
Accuracy_President
```

[24]: 0.7959183673469388

[25]:
```
Accuracy_FirstLady = np.sum(X_train[1] == x_2)/49
Accuracy_FirstLady
```

[25]: 0.6938775510204082

[26]:
```
Accuracy_Gentelman = np.sum(X_train[2] == x_3)/49
Accuracy_Gentelman
```

[26]: 0.7551020408163265

20 Model evaluation with 20% noise(direction)

[27]:
```
X_Clinton_20_noised = ShowResultWithNoise(X_Clinton, 0.2)
X_Hillary_20_noised = ShowResultWithNoise(X_Hillary, 0.2)
X_Kenstar_20_noised = ShowResultWithNoise(X_Kenstar, 0.2)
Y_Clinton_20_noised = ShowResultWithNoise(Y_Clinton, 0.2)
Y_Hillary_20_noised = ShowResultWithNoise(Y_Hillary, 0.2)
Y_Kenstar_20_noised = ShowResultWithNoise(Y_Kenstar, 0.2)
```

[28]:
```
X_train = np.concatenate([X_Clinton_20_noised.reshape(1,49), X_Hillary_20_noised.
↪reshape(1,49), X_Kenstar_20_noised.reshape(1,49)], axis=0)
y_train = np.concatenate([Y_Clinton.reshape(1,63), Y_Hillary.reshape(1,63),␣
↪Y_Kenstar.reshape(1,63)], axis=0)
```

[29]:
```
y_1 = model.forward(X_train[0])
y_2 = model.forward(X_train[1])
y_3 = model.forward(X_train[2])
```

[30]:
```
Accuracy_Clinton = np.sum(y_train[0] == y_1)/63
Accuracy_Clinton
```

[30]: 0.8888888888888888

[31]:
```
Accuracy_Hillary = np.sum(y_train[1] == y_2)/63
Accuracy_Hillary
```

[31]: 0.9206349206349206

[32]:
```
Accuracy_Kenstar = np.sum(y_train[2] == y_3)/63
Accuracy_Kenstar
```

[32]: 0.873015873015873

21 Model evaluation with 20% noise(Reverse)

[33]:
```
X_train = np.concatenate([X_Clinton.reshape(1,49), X_Hillary.reshape(1,49),␣
↪X_Kenstar.reshape(1,49)], axis=0)
y_train = np.concatenate([Y_Clinton_20_noised.reshape(1,63), Y_Hillary_20_noised.
↪reshape(1,63), Y_Kenstar_20_noised.reshape(1,63)], axis=0)
```

[34]:
```
x_1 = model.backward(y_train[0])
x_2 = model.backward(y_train[1])
x_3 = model.backward(y_train[2])
```

[35]:
```
Accuracy_President = np.sum(X_train[0] == x_1)/49
Accuracy_President
```

[35]: 0.7959183673469388

[36]:
```
Accuracy_FirstLady = np.sum(X_train[1] == x_2)/49
Accuracy_FirstLady
```

[36]: 0.6938775510204082

[37]:
```
Accuracy_Gentleman = np.sum(X_train[2] == x_3)/49
Accuracy_Gentleman
```

[37]: 0.7551020408163265

22 Part 5

[38]:
```
X_Lewisky = np.array([
                      1,0,0,1,1,0,0,
                      1,1,0,0,1,0,1,
                      1,1,1,0,1,1,1,
                      1,1,0,1,0,0,1,
                      1,1,1,0,0,1,1,
                      1,1,0,1,0,1,1,
                      1,1,1,1,0,0,1
])

Y_Lewisky = np.array([
                      1,0,1,0,0,1,1,
```

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

23 Model evaluation(Direction)

[39]:
```
X_train = np.concatenate([X_Clinton.reshape(1,49), X_Hillary.reshape(1,49),␣
 ↪X_Kenstar.reshape(1,49), X_Lewisky.reshape(1,49) ], axis=0)
y_train = np.concatenate([Y_Clinton.reshape(1,63), Y_Hillary.reshape(1,63),␣
 ↪Y_Kenstar.reshape(1,63), Y_Lewisky.reshape(1,63)], axis=0)
```

[40]:
```
y_1 = model.forward(X_train[0])
y_2 = model.forward(X_train[1])
y_3 = model.forward(X_train[2])
y_4 = model.forward(X_train[3])
```

[41]:
```
Accuracy_Clinton = np.sum(y_train[0] == y_1)/63
Accuracy_Clinton
```

[41]: 0.8888888888888888

[42]:
```
Accuracy_Hillary = np.sum(y_train[1] == y_2)/63
Accuracy_Hillary
```

[42]: 0.9206349206349206

[43]:
```
Accuracy_Kenstar = np.sum(y_train[2] == y_3)/63
Accuracy_Kenstar
```

[43]: 0.873015873015873

[44]:
```
Accuracy_Lewisky = np.sum(y_train[3] == y_4)/63
Accuracy_Lewisky
```

[44]: 0.6507936507936508

24 Model evaluation(Reverse)

[45]:
```
x_1 = model.backward(y_train[0])
x_2 = model.backward(y_train[1])
x_3 = model.backward(y_train[2])
x_4 = model.backward(y_train[3])
```

[46]:
```
Accuracy_President = np.sum(X_train[0] == x_1)/49
Accuracy_President
```

[46]: 0.7959183673469388

[47]:
```
Accuracy_FirstLady = np.sum(X_train[1] == x_2)/49
Accuracy_FirstLady
```

[47]: 0.6938775510204082

[48]:
```
Accuracy_Gentleman = np.sum(X_train[2] == x_3)/49
Accuracy_Gentleman
```

[48]: 0.7551020408163265

[49]:
```
Accuracy_SweetGirl = np.sum(X_train[3] == x_4)/49
Accuracy_SweetGirl
```

[49]: 0.6938775510204082

As you can see, the network's performance in the inputs and outputs for the first three characters is the same as the previous, but the network performance in the fourth character input and output is less than the rest.