University of Tehran

Electrical and Computer Engineering Department

Neural Networks and Deep Learning

Homework 3

Name	Danial Saeedi
Student No	810198571
Date	Monday, December 20, 2021

Table of contents

1. Question 1: Character Recognition using Hebbian Learning Rule 3
2. Question 2: Auto-associative Net 14
3. Question 3: Discrete Hopfield Net 20
4. Question 4: Auto-associative Net 24

1. Question 1: Character Recognition using Hebbian Learning Rule

Plotting Input and Output

In [12]:
```
from matplotlib import pyplot as plt

fig = plt.figure(figsize=(8, 8))
fig.add_subplot(2, 3, 1)
plt.imshow(A.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('Letter A')
fig.add_subplot(2, 3, 2)
plt.imshow(B.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('Letter B')
fig.add_subplot(2, 3, 3)
plt.imshow(C.reshape((9, 7)), interpolation='nearest',cmap='Greys')
plt.title('Letter C')

fig.add_subplot(2, 3, 4)
plt.imshow(small_A.reshape((5, 3)), interpolation='nearest',cmap='Greys')

fig.add_subplot(2, 3, 5)
plt.imshow(small_B.reshape((5, 3)), interpolation='nearest',cmap='Greys')

fig.add_subplot(2, 3, 6)
plt.imshow(small_C.reshape((5, 3)), interpolation='nearest',cmap='Greys')


plt.show()
```

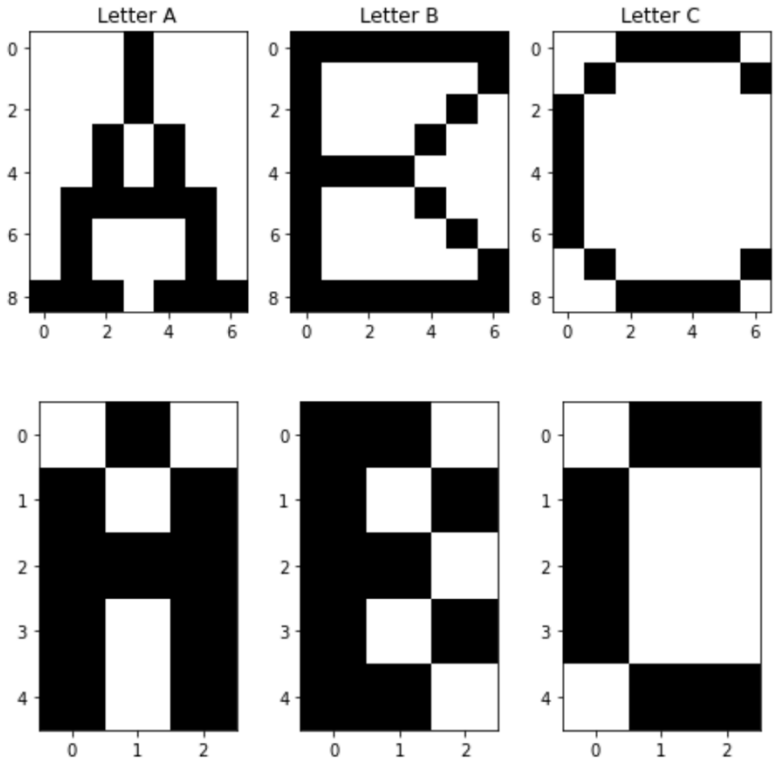


Figure 1 Dataset

Define the model

```
In [ ]: class Network() :

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

                self.in_features = in_features
                self.out_features = out_features
                # Initialize Weights & Biases
                self.init_wad()

            # Initialize Weights with 0
            def init_wad(self) :
                self.weights = np.zeros(shape=(in_features*out_features,)).reshape(-1,out_features)

            #Activation Function
            def h(self,num) :
                if num >= 0 :
                    return 1
                else :
                    return -1
            def forward(self, x):
                net = np.matmul(x,self.weights)
                prediction = pd.Series(net)
                prediction = prediction.apply(self.h).values
                return prediction

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

            def train(self,X_train,y_train,epochs = 1) :
                error_per_epoch = []
                for epoch in range(0,epochs):

                    for i in range(len(X_train)) :
                      # Backpropagation
                      self.backward(X_train[i],y_train[i])
```

Part 1

The input and output layers should have 9*7 = 63 and 3*5 = 15 neurons respectively.

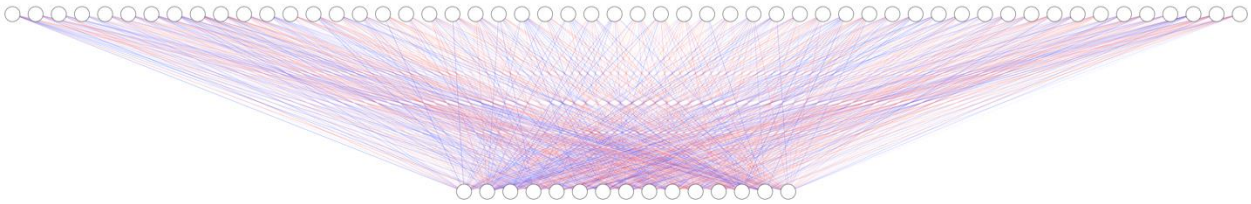


Figure 2 Model's Architecture

Define the model

```
In [ ]: in_features = 63
        out_features = 15
        model = Network(in_features, out_features)
```

Train the model

```
In [17]: model.train(X_train,y_train, epochs = 1)
```

Prediction

```
In [ ]: predicted_A = model.forward(X_train[0])
        predicted_B = model.forward(X_train[1])
        predicted_C = model.forward(X_train[2])
```

As you can see, this model is able to produce the desired outputs for all thr
ee inputs:

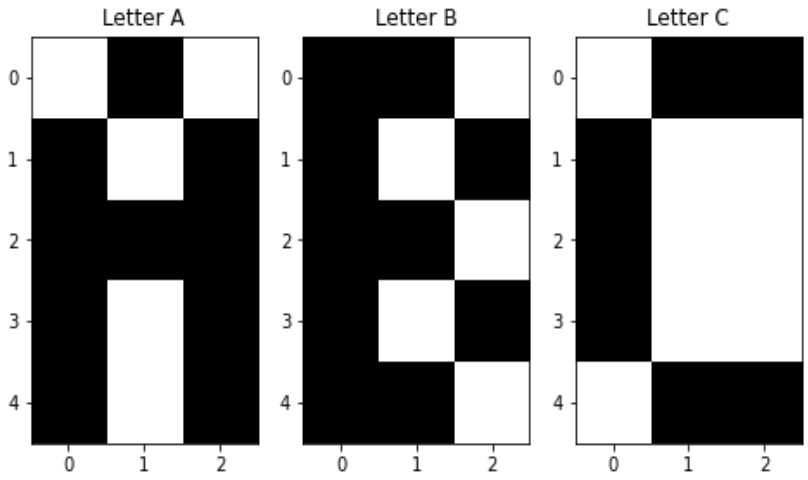


Figure 3 The Model's Result

Part 2

The smallest dimension we can reduce to is 2x2.

Define the model

```
In [45]: in_features = 63
         out_features = 4
         model2 = Network(in_features, out_features)
```

Train the model

```
In [ ]: model2.train(X_train,y_train2, epochs = 1)
```

Prediction

```
In [ ]: predicted_A = model2.forward(X_train[0])
        predicted_B = model2.forward(X_train[1])
        predicted_C = model2.forward(X_train[2])
```

As you can see, this model is able to produce the desired output for all three inputs:

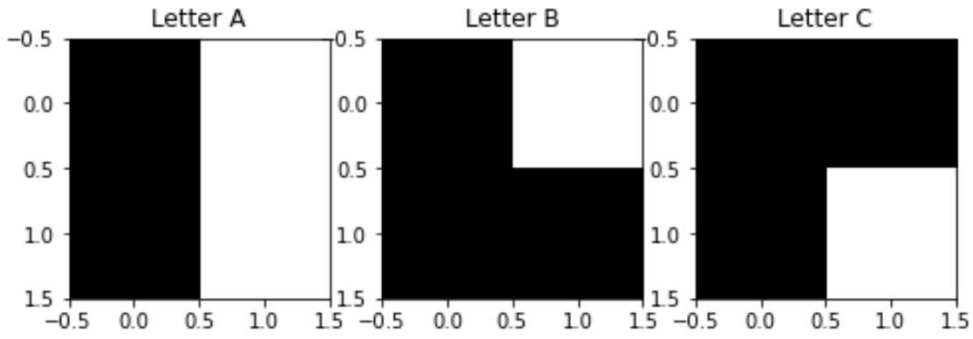


Figure 4 The Model's Result

Part 3

This function produces noise:

```
In [ ]: import math
def noise_producer(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
```

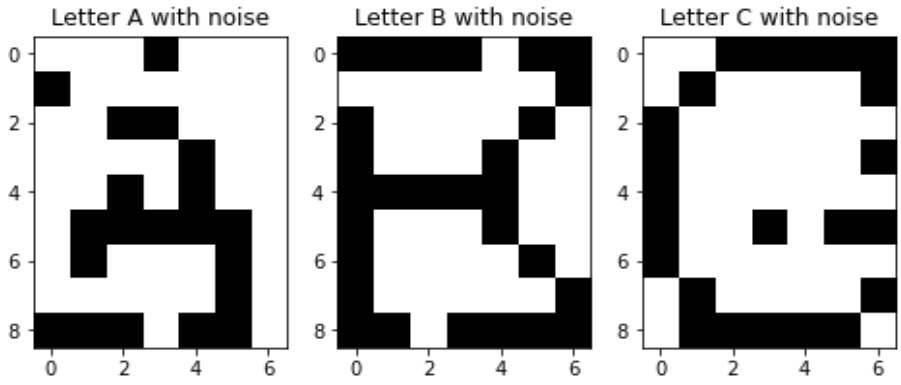


Figure 5 10% Noise

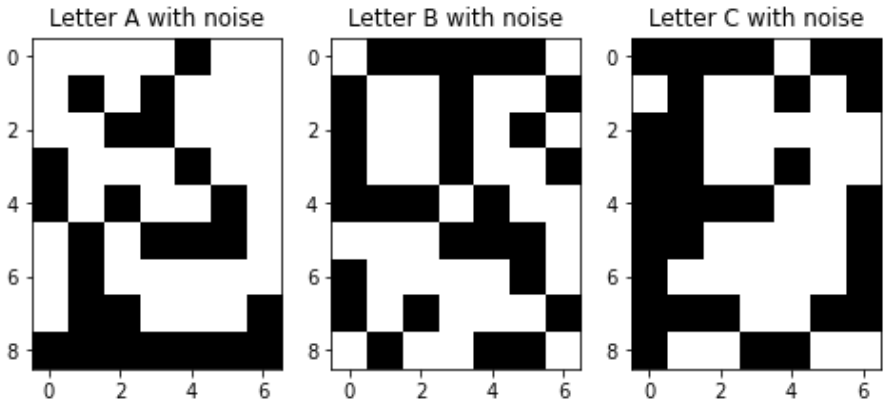


Figure 6 40% Noise

Adding 10% noise to input of part 1's model

The accuracy of part 1's model with 10% noise applied is 100%.

```
In [ ]: predicted_A = model.forward(A_noise_10)
        predicted_B = model.forward(B_noise_10)
        predicted_C = model.forward(C_noise_10)
```

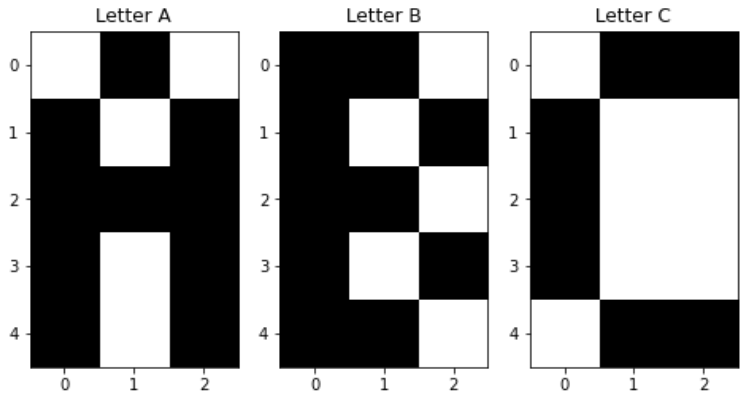


Figure 7 The Model's Result

Adding 40% noise to input of part 1's model

The accuracy of part 1's model with 40% noise applied is 66%.

```
In [ ]: predicted_A = model.forward(A_noise_40)
        predicted_B = model.forward(B_noise_40)
        predicted_C = model.forward(C_noise_40)
```

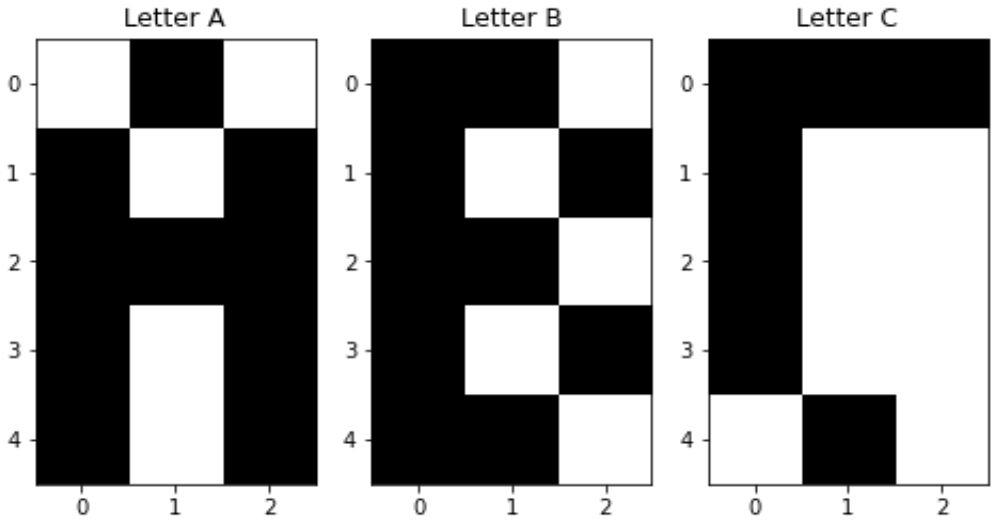


Figure 8 The Model's Result

Adding 10% noise to input of part 2's model

The accuracy of part 2's model with 10% noise applied is 100%.

```
In [ ]: predicted_A = model2.forward(A_noise_40)
        predicted_B = model2.forward(B_noise_40)
        predicted_C = model2.forward(C_noise_40)
```

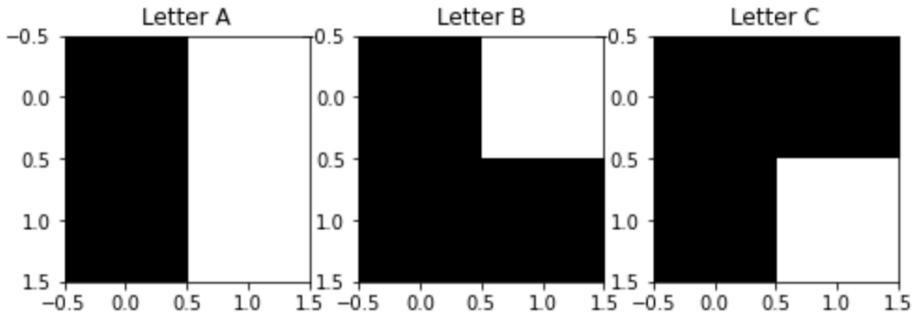


Figure 9 The Model's Result

Adding 40% noise to input of part 2's model

The accuracy of part 2's model with 40% noise applied is 100%.

```
In [ ]: predicted_A = model2.forward(A_noise_40)
        predicted_B = model2.forward(B_noise_40)
        predicted_C = model2.forward(C_noise_40)
```

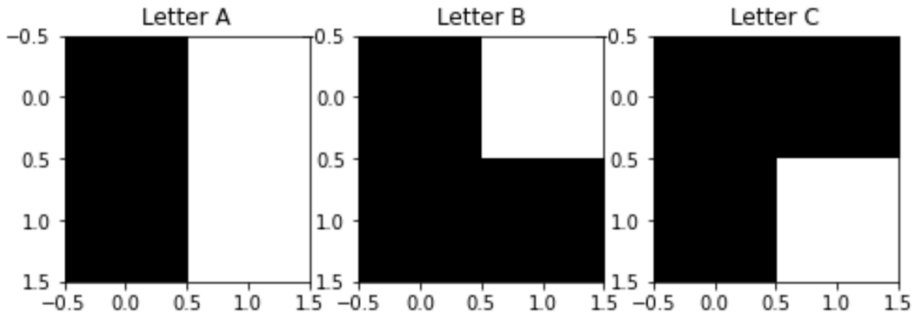


Figure 10 The Model's Result

Part 4

This function turns random pixels into 0.

```
In [ ]: import math
        def noise_producer2(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
```

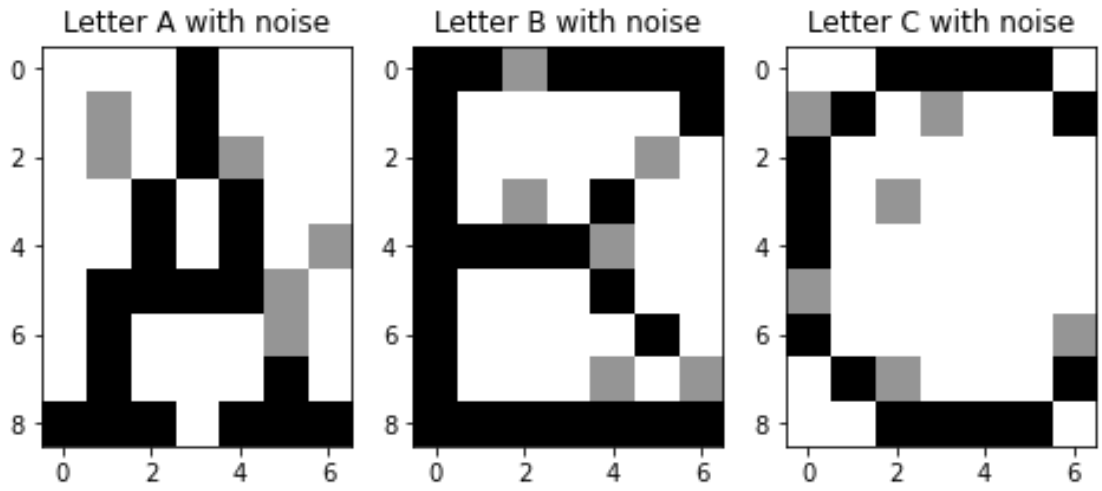


Figure 11 10% Destroyed Information

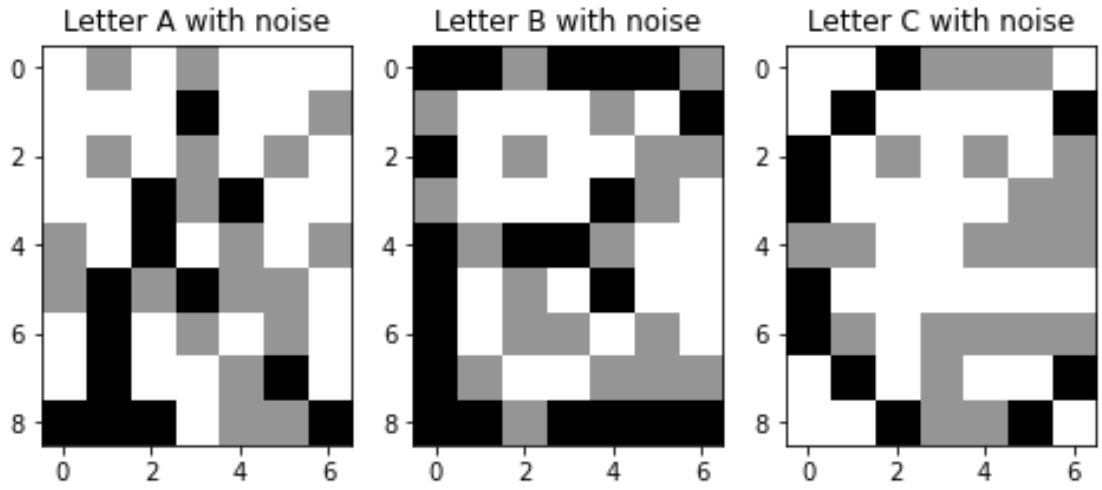


Figure 12 40% Destroyed Information

Destroying 10% of pixels (part 1's model)

The accuracy of this model is 100% with 10% destroyed pixels.

```
In [ ]: predicted_A = model.forward(A_noise_10)
        predicted_B = model.forward(B_noise_10)
        predicted_C = model.forward(C_noise_10)
```

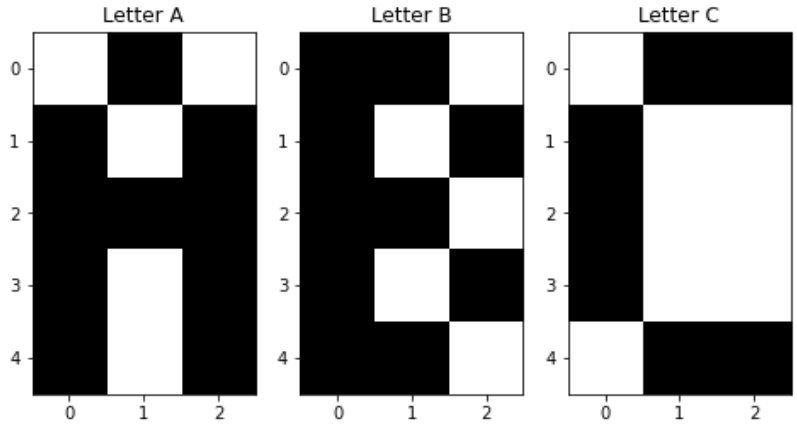


Figure 13 The Model's Result

Destroying 40% of pixels (part 1's model)

The accuracy of part 1's model with 40% destroyed information is 100%.

```
In [ ]: predicted_A = model.forward(A_noise_40)
        predicted_B = model.forward(B_noise_40)
        predicted_C = model.forward(C_noise_40)
```

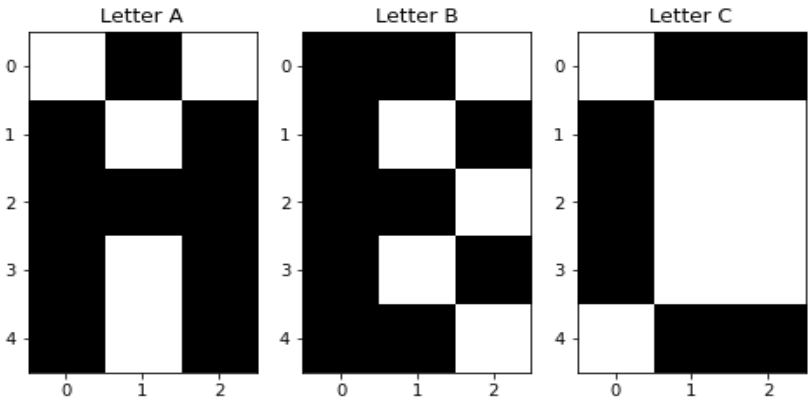


Figure 14 The Model's Result

Destroying 10% of pixels (part 2's model)

The accuracy of part 2's model with 10% destroyed information is 100%.

```
In [ ]: predicted_A = model2.forward(A_noise_10)
        predicted_B = model2.forward(B_noise_10)
        predicted_C = model2.forward(C_noise_10)
```

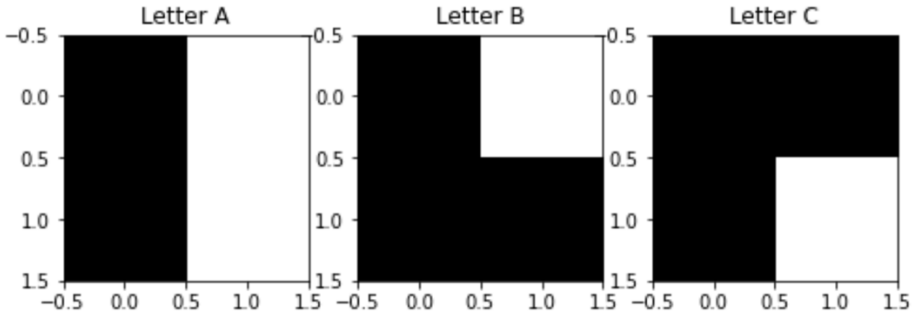


Figure 15 The Model's Result

Destroying 40% of pixels (part 2's model)

The accuracy of part 2's model with 40% destroyed information is 100%.

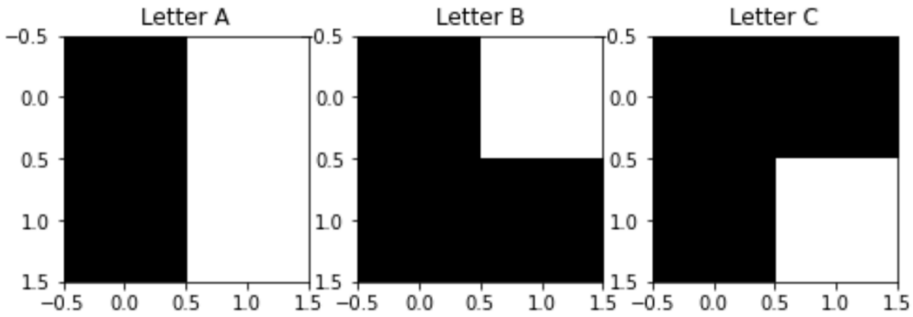


Figure 16 The Model's Result

Part 5

Resistance of the model with information loss and noise

The model was able to predict correct pattern for 10% noise. But the model couldn't predict correct pattern for 40% noise.

The model was able to predict correct pattern for **both** 10% and 40% with information loss.

So the model is more **resistant** to **information** loss than noise.

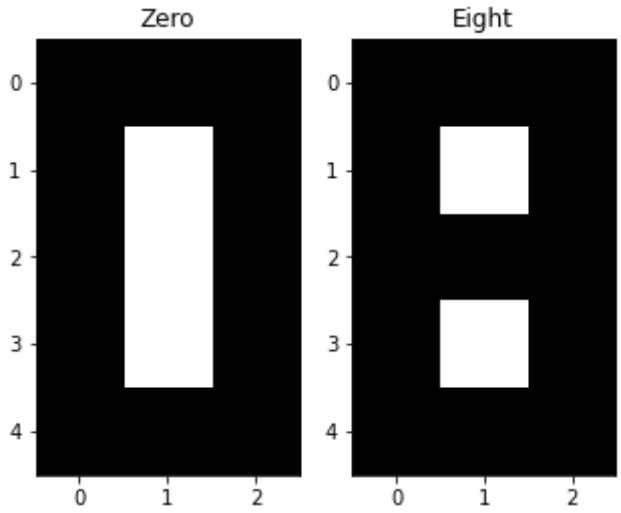
The effect of output dimension

Reducing output dimension resulted in better **resistance** to both information loss and noise.

2. Question 2: Auto-associative Net

Plotting Input

In [ ]:
```
from matplotlib import pyplot as plt
fig = plt.figure(figsize=(8, 8))
fig.add_subplot(1, 3, 1)
plt.imshow(zero.reshape((5, 3)), interpolation='nearest',cmap='Greys')
plt.title('Zero')
fig.add_subplot(1, 3, 2)
plt.imshow(eight.reshape((5, 3)), interpolation='nearest',cmap='Greys')
plt.title('Eight')
plt.show()
```



Define Model

In [ ]:
```
class Network() :

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

        self.in_features = in_features
        self.out_features = out_features
        # Initialize Weights & Biases
        self.init_wad()

    # Initialize Weights with 0
    def init_wad(self) :
        self.weights = np.zeros(shape=(self.in_features*self.out_features,)).reshape(-1,self.out_features)

    #Activation Function
    def h(self,num) :
        if num >= 0 :
            return 1
        else :
            return -1
    def forward(self, x):
        net = np.matmul(x,self.weights)
        prediction = pd.Series(net)
        prediction = prediction.apply(self.h).values
        return prediction

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

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

            for i in range(len(X_train)) :
              # Backpropagation
              self.backward(X_train[i],y_train[i])

        # Modified Hebbian Learning Rule
        self.weights = self.weights-len(X_train)*np.identity(self.in_features)
```

Part 1: Training with Modified Hebbian Learning Rule

In [ ]:
```
in_features = 15
out_features = 15
model = Network(in_features, out_features)
```

In [ ]:
```
model.train(X_train,X_train, epochs = 2)
```

In [ ]:
```
predicted_zero = model.forward(X_train[0])
predicted_eight = model.forward(X_train[1])
```

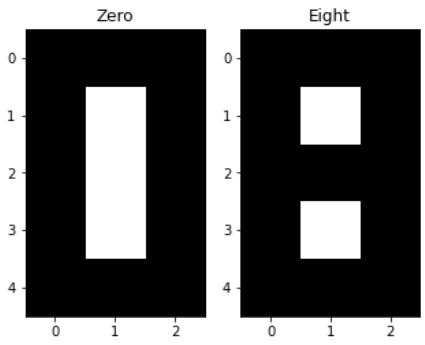


Figure 17 The model's output for two inputs

Part 2: Model's performance with noised pixels

Testing the model with only one noised pixel

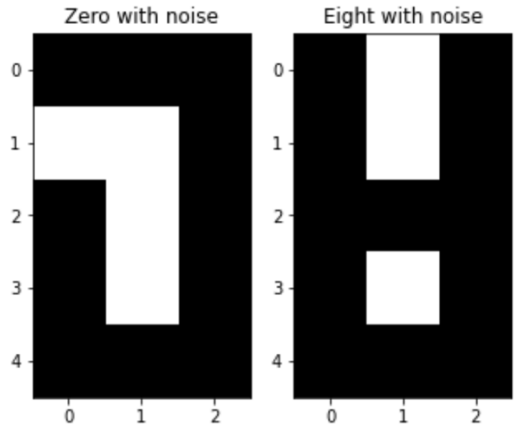


Figure 18 Input with noise

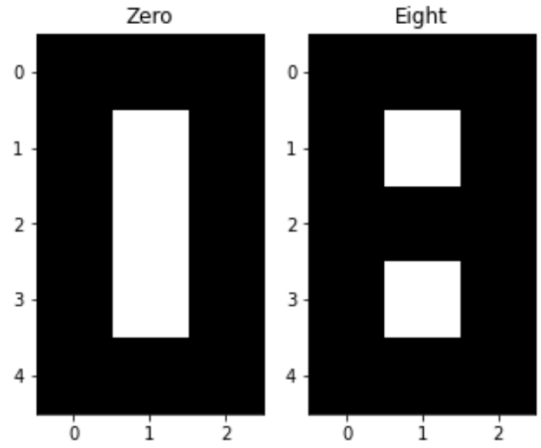


Figure 19 Output(Denoised)

Model's accuracy with all possible noised pixels

The functions that I used here:

In [16]:
```
def which_index(arr):
    return [
        i for i in range(len(arr))
        if arr[i] == True
    ]
```

In [ ]:
```
import random

# p : number of 1's to change, n : number of -1's to change
def make_noise(p,n,arr):
    with_noise = np.copy(arr)
    negative_indices = which_index(arr == -1)
    positive_indices = which_index(arr == 1)

    random_negative = random.sample(negative_indices, len(negative_indices) if n > len(negative_indices) else n )
    random_positive = random.sample(positive_indices, len(positive_indices) if p > len(positive_indices) else p)

    with_noise[random_negative] = 1
    with_noise[random_positive] = -1

    return with_noise
```

In [ ]:
```
accuracy_per_wrong_pixels = []
for i in range(1,15):
  accumulate_accuracy = 0
  # producing all possible combination of -1 and 1 wrong pixels
  # p : number of 1's to change, n : number of -1's to change
  # n + p = i
  for p in range(0,i+1) :
    n = i-p

    zero_noised = make_noise(p,n,zero)
    eight_noised = make_noise(p,n,eight)

    predicted_zero = model.forward(zero_noised)
    predicted_eight = model.forward(eight_noised)

    correct = 0

    if np.array_equal(predicted_zero, zero):
      correct += 1
    if np.array_equal(predicted_eight, eight):
      correct += 1

    accumulate_accuracy += correct*100/2

  accuracy_per_wrong_pixels.append(accumulate_accuracy / (i+1) )
```

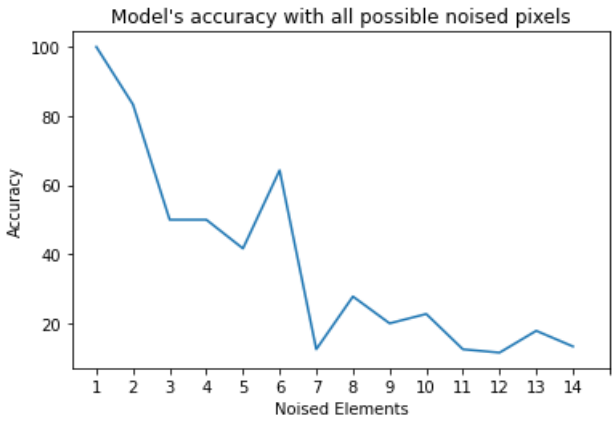


Figure 20 Model's Accuracy with all possible noised pixels

Part 3: Model's performance with missing pixels

Testing the model with only one missing pixel

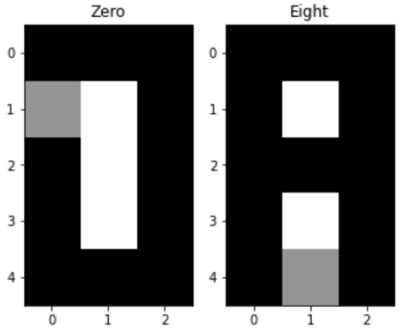


Figure21 missing pixel

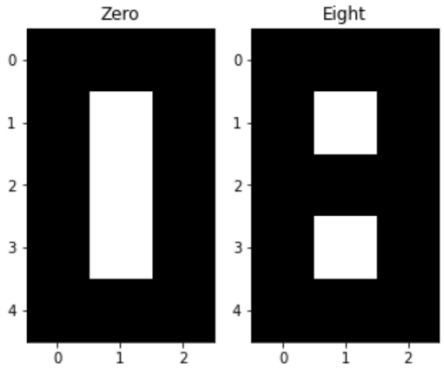


Figure 22 Output of the model with one missing value

Model's accuracy with all possible missing pixels

Model’s resistance to missing value(destroyed value) is much more than noised values and this result can be generalized to models that use Modified Hebbian Learning Rule.

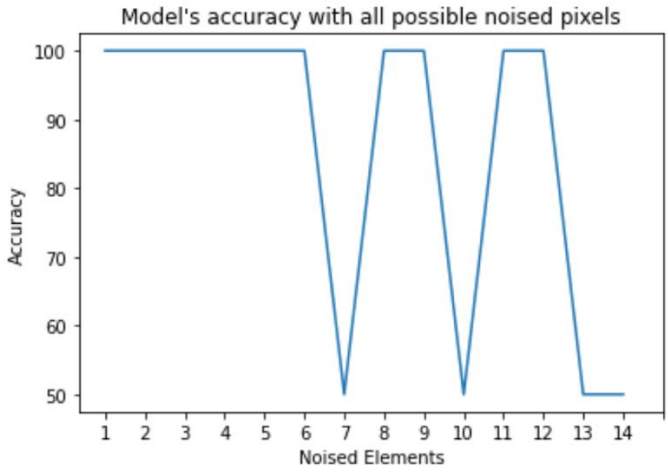


Figure23 Model's accuracy with all possible missing values

Part 4

Testing the model with only one noised pixel

The model accuracy with only one noised pixel is 100%.

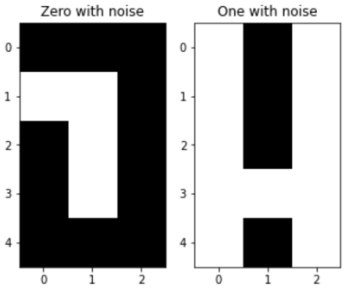


Figure 24 One Pixel Noised

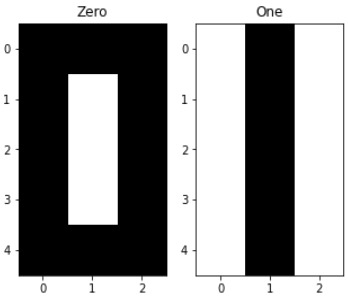


Figure 25 Model's output (Denoised)

Model's accuracy with all possible noised pixels

The patters in part 4 are less similar to each other. So the **rate** of decrease of accuracy based on noised elements is **slower** than part 2. Overall, this part is more **capable** of **retrieving** the **original** input.

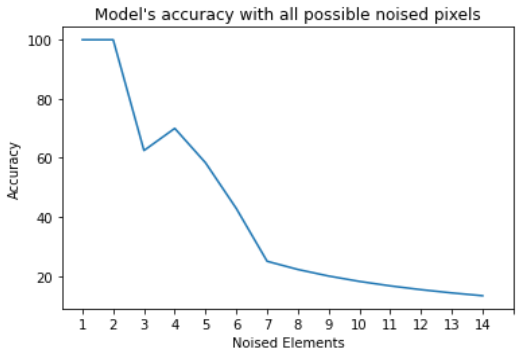


Figure26 Model's accuracy with all possible noised pixel

3. Question 3: Discrete Hopfield Net

Importing Dependencies

In [29]:
```
import numpy as np
from PIL import Image
import os
import matplotlib.pyplot as plt
```

Part 1: Loading images

In [30]:
```
# Set the threshold value
threshold = 180

path = os.getcwd()

path_train = path + '/pacman_Train.jpg'
path_test  = path + '/pacman_Test.jpg'

def read_binarize_img(path_img):

    # Read the image
    img_train = Image.open(path_img).convert(mode="L")
    img_train = img_train.resize(size=(100,100))

    # Binarize the image
    img_train_array = np.asarray(img_train,dtype=np.uint8)
    x = np.zeros(img_train_array.shape,dtype=np.float)
    x[img_train_array > threshold] = 1
    x[x==0] = -1

    return x
```

In [78]:
```
# Read images
train = read_binarize_img(path_train)
test = read_binarize_img(path_test)
```

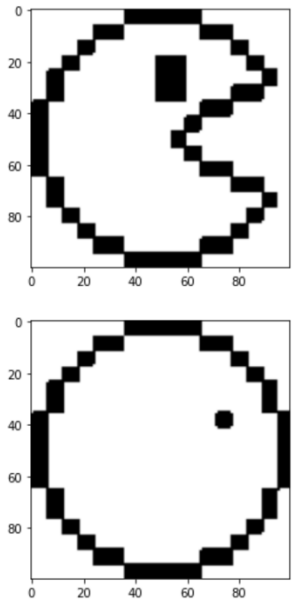


Figure 27 The train and test image

Define Model

```
In [104]: class DiscreteHopfieldNet(object):

              def __init__(self):
                  self.weight = None

              # Modified Hebbian Learning Rule
              def train(self, data,):
                  mat = np.vstack(data)
                  I = len(data) * np.identity(np.size(mat, 1))
                  self.weight = np.dot(mat.T, mat) - I

              def predict(self, data, theta=0.5, iterations=1000):
                  # Random orders for computations
                  indexs = np.random.randint(0, len(self.weight) - 1, (iterations, len(data)))
                  for ind in indexs:
                      diagonal = np.diagonal(np.dot(self.weight[ind], data.T))
                      diagonal = np.expand_dims(diagonal, -1)
                      value = np.apply_along_axis(lambda x: 1 if x > theta else -1, 1, diagonal)

                      for i in range(len(data)):
                          data[i, ind[i]] = value[i]

                  return data

In [105]: size = (100, 100)

In [106]: model = DiscreteHopfieldNet()
```

Part 2: Training(Constructing Weight Matrix)

```
In [107]: %%time
          model.train([train.flatten()])

          CPU times: user 1.64 s, sys: 419 ms, total: 2.05 s
          Wall time: 970 ms
```

Part 3

Iterations = 1000

```
In [124]: %%time
          recovery = model.predict(np.array([test.flatten()], copy=False),iterations=1000)

          CPU times: user 428 ms, sys: 11.5 ms, total: 440 ms
          Wall time: 90.3 ms

In [125]: recovery = recovery[0].reshape(size)
```

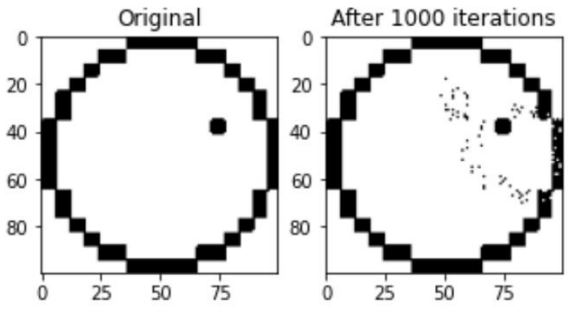


Figure 28 Original vs Output

More than 1000 iterations are needed to retrieve the desired output.

Iterations = 50000

```
In [128]: %%time
          recovery = model.predict(np.array([test.flatten()], copy=False),iterations=50000)

          CPU times: user 14.9 s, sys: 154 ms, total: 15 s
          Wall time: 3.05 s

In [129]: recovery = recovery[0].reshape(size)
```

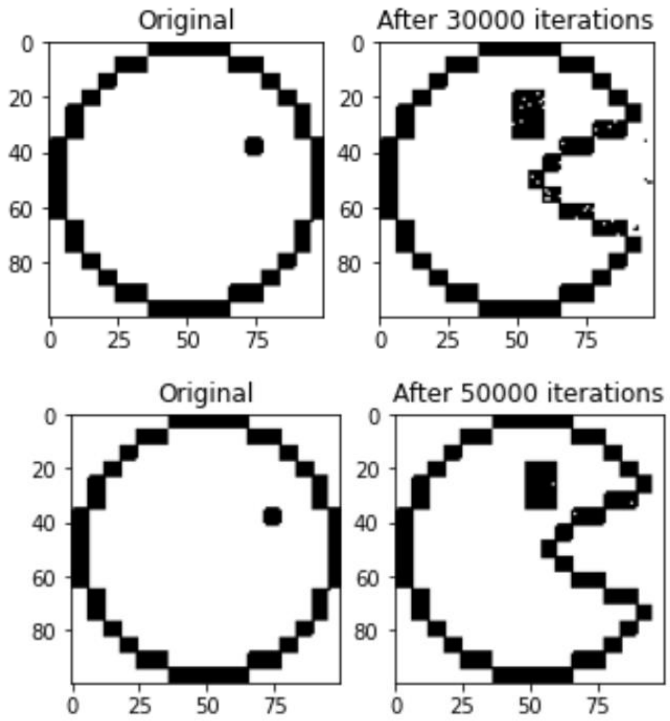


Figure29 Original vs Output

The desired output is produced after 50000 iterations.

Part 4

The desired output is produced after 50000 iterations. Compared to part 3, some of the pixels are still incorrect because the ghost image is much different than the train image. The test image in part 2 is similar to train image (that's why it is easier and quicker to retrieve the desired image).

Iterations = 1000

```
In [19]: %%time
         recovery = model.predict(np.array([test2.flatten()], copy=False),iterations=1000)

         CPU times: user 350 ms, sys: 4.99 ms, total: 355 ms
         Wall time: 71.8 ms
```

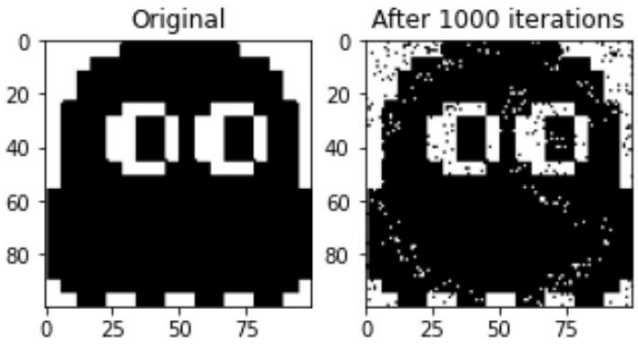


Figure30 Original vs Output

More than 1000 iterations are needed to retrieve the desired output.

Iterations = 50000

```
In [22]: %%time
         recovery = model.predict(np.array([test2.flatten()], copy=False),iterations=50000)

         CPU times: user 14.1 s, sys: 101 ms, total: 14.2 s
         Wall time: 2.87 s
```

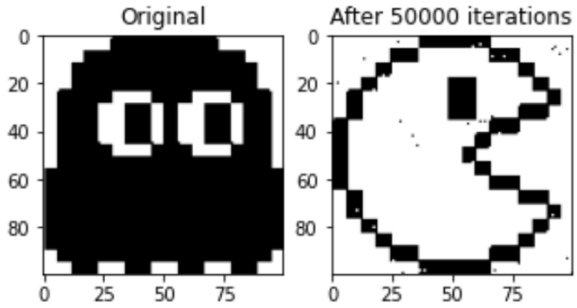


Figure31 Original vs Output

4. Question 4: Auto-associative Net

Plotting Input

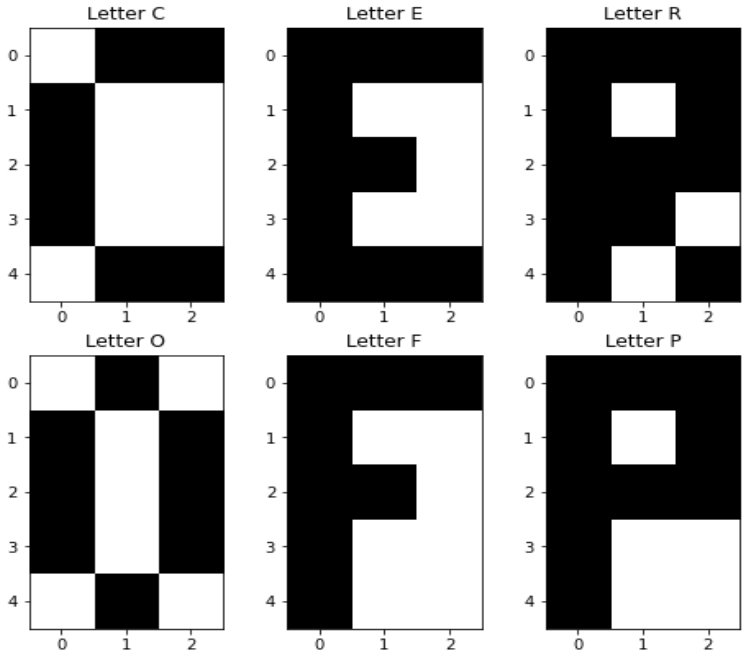


Figure32 Inputs

Define Model

In [6]:
```
class BAM() :

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

        self.in_features = in_features
        self.out_features = out_features
        # Initialize Weights & Biases
        self.init_wad()

    # Initialize Weights with 0
    def init_wad(self) :
        self.weights = np.zeros(shape=(self.in_features*self.out_features,)).reshape(-1,self.out_features)

    #Activation Function bipolar
    def h(self,num,threshold = 0.25) :
        if num > threshold :
            return 1
        elif num < threshold:
            return -1
        else :
            return threshold

    def forward(self, x):
        net = np.matmul(x,self.weights)
        prediction = pd.Series(net)
        prediction = prediction.apply(self.h).values
        return prediction

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

In [7]:
```
model = BAM(in_features=15,out_features=3)
```

Part 1

```
In [8]: X_train = np.concatenate([C.reshape(1,15),E.reshape(1,15), R.reshape(1,15)], axis=0)
        y_train = np.concatenate([C_code.reshape(1,3),E_code.reshape(1,3), R_code.reshape(1,3)], axis=0)

In [9]: model.initialize_weights(X_train, y_train,epochs=1)
```

```
array([[-1.,  1.,  1.],
       [-3., -1., -1.],
       [-3., -1., -1.],
       [-3., -1., -1.],
       [ 3.,  1.,  1.],
       [ 1.,  3., -1.],
       [-3., -1., -1.],
       [-1.,  1.,  1.],
       [ 1.,  3., -1.],
       [-3., -1., -1.],
       [ 1.,  3., -1.],
       [ 3.,  1.,  1.],
       [-1.,  1.,  1.],
       [-1., -3.,  1.],
       [-3., -1., -1.]])
```

Figure33 Weight Matrix

Part 2

Forward

The trained model is able to classify C and R perfectly. The model mistakes E with C since E is **quite similar** to C. So the forward accuracy for training data set is 2/3 = 66%.

```
In [11]: y_1 = model.forward(X_train[0])
         y_2 = model.forward(X_train[1])
         y_3 = model.forward(X_train[2])

         print('Input : C Output :',y_1)
         print('Input : E Output :',y_2)
         print('Input : R Output :',y_3)

         Input : C Output : [-1 -1 -1]
         Input : E Output : [-1 -1 -1]
         Input : R Output : [-1  1 -1]
```

Backward

The model is able to regenerate each training input with corresponding code. So the backward accuracy is 100% for training data set.

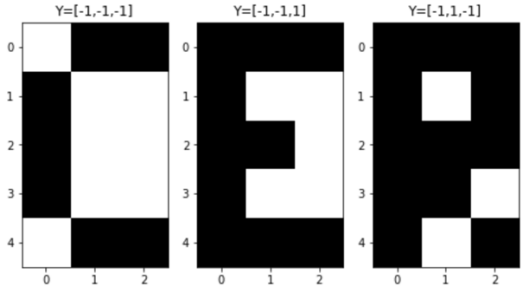


Figure34 Generated in backward direction

Part 3

The average accuracy of this model with 40% noise applied to each input in 100 runs is 51.66%.

This function is used for producing random noise:

In [14]:
```
import math
def noise_producer(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
```

Calculate Accuracy

The average accuracy with 40% noised pixels in 100 runs is 51.66666%

In [42]:
```
correct = 0
total = 0
for iteration in range(0,100) :
    total += 3
    # 40% Noise
    C_noise = noise_producer(C,0.4)
    E_noise = noise_producer(E,0.4)
    R_noise = noise_producer(R,0.4)

    X_train_noised = np.concatenate([C_noise.reshape(1,15),E_noise.reshape(1,15), R_noise.reshape(1,15)], axis=0)
    y_train = np.concatenate([C_code.reshape(1,3),E_code.reshape(1,3), R_code.reshape(1,3)], axis=0)

    for i in range(len(X_train_noised)) :
      prediction = model.forward(X_train_noised[i])
      if np.array_equal(prediction, y_train[i]):
        correct += 1
```

In [45]:
```
print("Model's accuracy with 40% noised in 100 runs: ",correct*100/total)
```

```
Model's accuracy with 40% noised in 100 runs:  51.666666666666664
```

Part 4

The memory or storage capacity of BAM may be given as $\min(m, n)$, where n is the number of units in the X layer and m is the number of units in the Y layer. In this problem, n and m are 15 and 3 respectively. So the maximum number of patters that can be stored in this model is **3**.

Part 5

```
In [96]: model2 = BAM(in_features=15,out_features=3)
```

```
array([[ 2., -2., -2.],
       [-2., -2., -2.],
       [ 0., -4., -4.],
       [-2., -2., -2.],
       [ 2.,  2.,  2.],
       [ 0.,  4.,  0.],
       [-2., -2., -2.],
       [ 2., -2., -2.],
       [ 0.,  4.,  0.],
       [-2., -2., -2.],
       [ 0.,  4.,  0.],
       [ 0.,  4.,  4.],
       [ 2., -2., -2.],
       [-4.,  0.,  4.],
       [-4.,  0.,  0.]])
```

Figure 35 Weights after training

Forward

The trained model is able to classify C, O and F perfectly. But other letters are mistaken with letter C or F. So the forward accuracy for training data set is 3/6 = 50%

```
Input : C Output : [-1 -1 -1]
Input : E Output : [-1 -1 -1]
Input : R Output : [-1 -1 -1]
Input : O Output : [-1  1  1]
Input : F Output : [ 1 -1 -1]
Input : P Output : [ 1 -1 -1]
```

Backward

The model is able to regenerate letter F only in backward direction. So the backward accuracy is 1/6 = 16% for training data set.

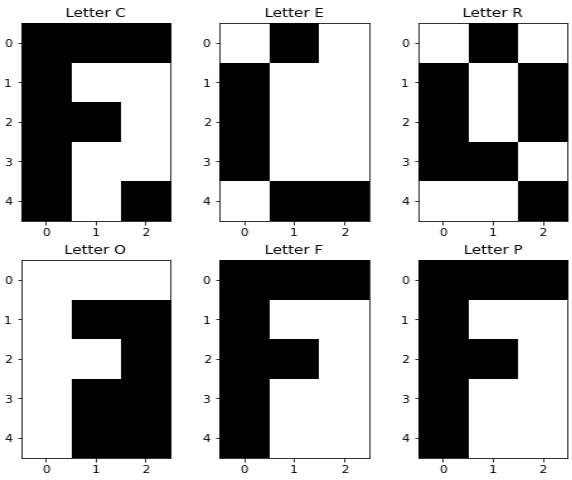


Figure 36 Generated in backward direction