UNIVERSITÉ Concordia UNIVERSITY

GINA CODY SCHOOL OF ENGINEERING AND COMPUTER SCIENCE

COEN 6331: Neural Networks

ASSIGNMENT -3

Submitted by:

Rohan Kodavalla (40196377)

I certify that this submission is my original work and meets the Faculty's Expectations of Originality

March 10, 2022

ABSTRACT

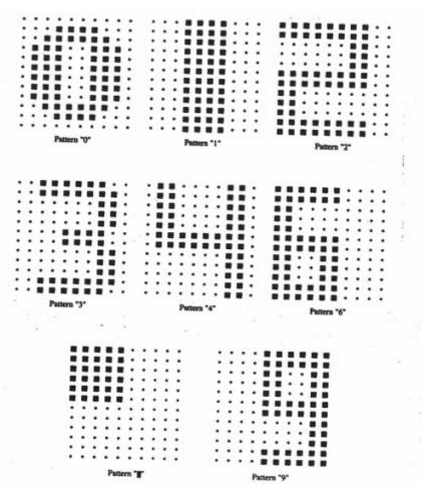
The objective of this assignment is to design a Hopfield network to recognize the patterns as an associator.

Hopfield nets serve as content-addressable ("associative") memory systems with binary threshold nodes. They are guaranteed to converge to a local minimum and, therefore, may converge to a false pattern (wrong local minimum) rather than the stored pattern (expected local minimum). Hopfield networks also provide a model for understanding human memory. We show, for example, that for synchronous operation, Hopfield's net can oscillate in the steady state and that the oscillation can be avoided by slightly altering the neural operation. Asynchronous operation of the net, on the other hand, always results in a stable steady state when the neural threshold function is properly defined, and nonzero auto connects are used. Use of nonzero neural auto connects also results in a net that converges faster than when zero auto connects are used. This is true for both asynchronous and synchronous operations.

Thus, in general, asynchronous implementation of nets with nonzero auto connects have the best convergence properties. Better convergence, however, does not necessarily imply better (or worse) steady-state accuracy.

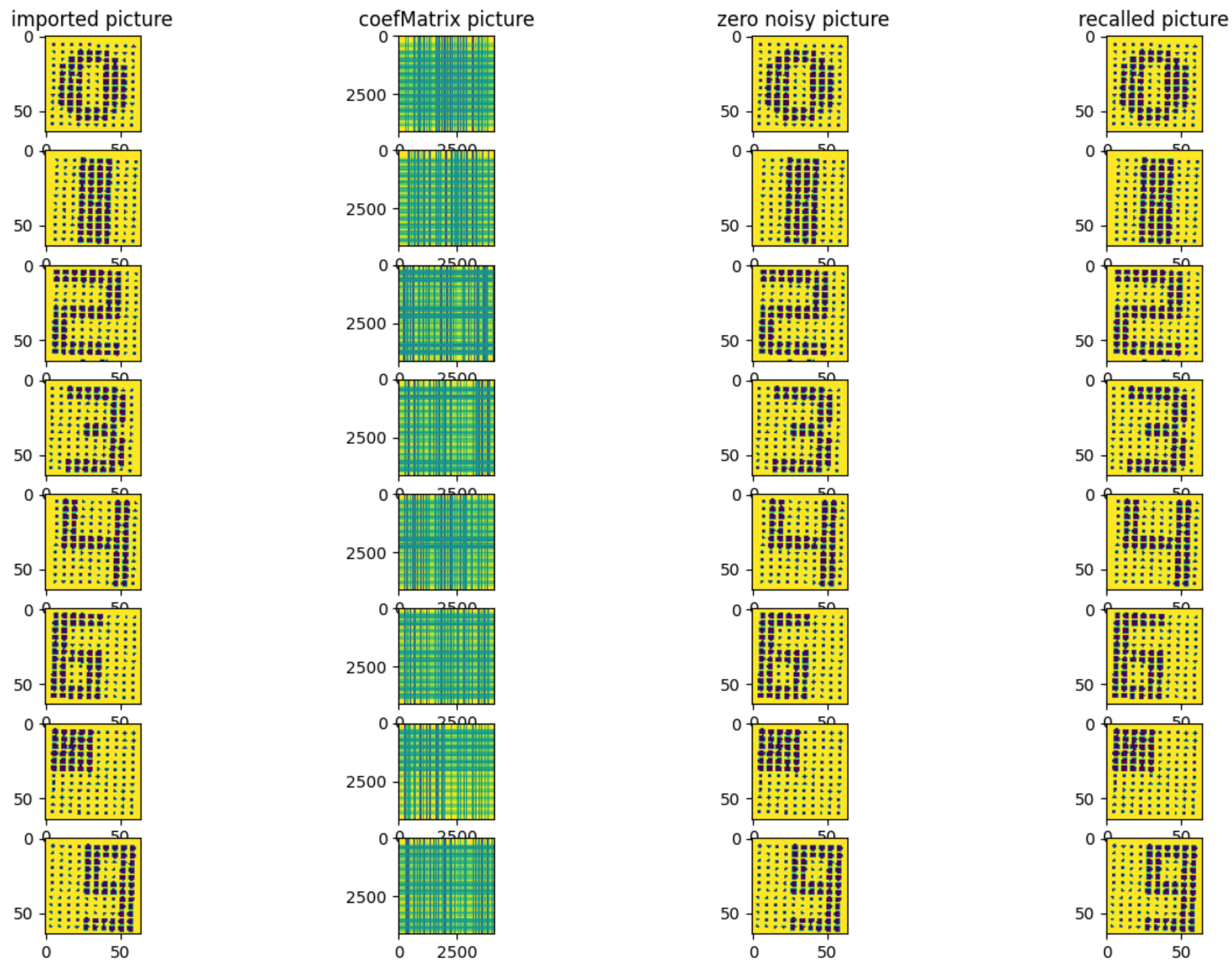
In our example of a Hopfield Network, one input image should first be stored and then be retrieved.

The input images are:



1. Synchronous learning (testing using clean patterns):

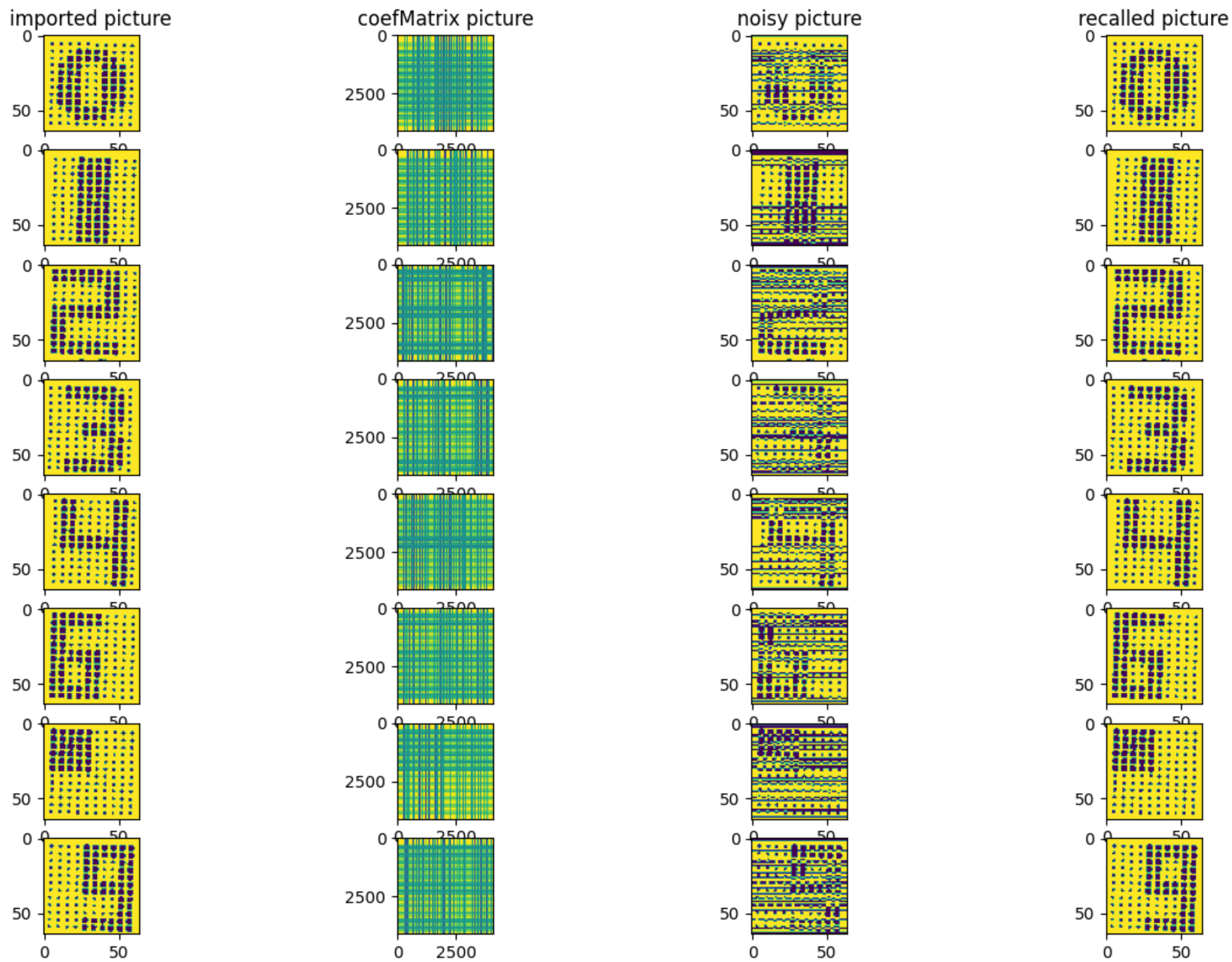
In Synchronous learning, all units are updated at the same time. From given data, the 'recall' using clean pattern is obtained as below –



As shown in results, the 'imported data' can be recalled perfectly as the 'recalled picture' when there is no noise i.e., when a 'clean pattern' is used. Also, the coefficient matrices are obtained.

2. **Synchronous learning (testing using noisy patterns):**

Now, let’s observe the ‘recalled picture’ when the patterns are corrupted with 25% noise. The corresponding output for the same is as below –



As observed, the ‘imported data’ can be recalled perfectly as the ‘recalled picture’ even when there is 25% noise along with the corresponding coefficient matrices.

CORRESPONDING CODE EXPLANATION FOR SYNCHRONOUS LEARNING PROCEDURE SHOWN ABOVE:

i) Importing the input patterns, noise –

Since an associative memory has polar states and patterns (or binary states and patterns), we convert the input image to a black and white image.

hopfield.py × hofield.png

```
                corrupted[i] = -1 * v
    return corrupted

#Import the image

zero = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/0.png','w')).copy()
one = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/1.png','w')).copy()
two = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/2.png','w')).copy()
three = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/3.png','w')).copy()
four = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/4.png','w')).copy()
six = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/6.png','w')).copy()
dash = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/-.png','w')).copy()
nine = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/9.png','w')).copy()

data = [zero, one, two, three, four, six, dash, nine]
print("Start to data preprocessing...")
data = [preprocessing(d) for d in data]

vector_list = []
noisy_list = []
#coefMatrix_list = []
predictedVec_list = []
plt.clf()

plt.figure(figsize=(15,10))
i=0
for image in data :
  #vector,noisyVec = imageGenerator(image)
  noisyVec = get_corrupted_input(image, 0)

  coefMatrix = trainer(image)
  predictedVec = prediction(noisyVec,coefMatrix)
  vector_list.append(image)
  noisy_list.append(noisyVec)
  predictedVec_list.append(predictedVec)
```

Importing data, converting to b/w image using 'rgb2gray function'

Initiate pre-processing the data

noisyVec introduces noise to imported pictures. Here, noise = 0%.

Obtaining coefficient matrix, predictedvec (used to predict the output using noisy image and coeff matrix).

Alternatively, when noise is 25% -

```
for image in data :
  #vector,noisyVec = imageGenerator(image)
  noisyVec = get_corrupted_input(image, 0.25)

```

ii) Obtaining coefficient matrix, generating predictedVec :

hopfield.py × hofield.png

```python
import numpy as np
import matplotlib.pyplot as plt
import scipy.misc as sp
import matplotlib.image as img
from skimage.color import rgb2gray
from matplotlib import pyplot as plt
from skimage.transform import resize
from skimage.filters import threshold_mean



def trainer(vector):
    vector = vector.flatten()
    #vector = np.reshape(vector, (len(vector)*len(vector)))
    coefMat = np.zeros([len(vector),len(vector)])
    for i in range(len(vector)):
        for j in range(len(vector)):
            if (i!=(i-j)):
                coefMat[i][i-j] = vector[i]*vector[i-j]
    vector = np.reshape(vector, [int(np.sqrt(len(vector))),int(np.sqrt(len(vector)))])
    return coefMat

def prediction(curuptedVec,coefMat):
    curuptedVec = curuptedVec.flatten()
    predictVec = np.zeros(len(curuptedVec))
    for i in range(len(curuptedVec)):
        temp = 0
        for j in range(len(curuptedVec)):
            temp += coefMat[i][j] * curuptedVec[j]
        if (temp>0):
            predictVec[i] = 1
        if (temp<0):
            predictVec[i] = -1

    predictVec = np.reshape(predictVec, [int(np.sqrt(len(predictVec))),int(np.sqrt(len(predictVec)))])
    return predictVec
```

iii) Pre-processing and getting corrupted input –

```python

def preprocessing(img, w=64, h=64):
    # Resize image
    img = resize(img, (w,h), mode='reflect')
    # Thresholding
    thresh = threshold_mean(img)
    binary = img > thresh
    shift = 2*(binary*1)-1 # Boolian to int
    # Reshape
    flatten = np.reshape(img, (w*h))
    return shift

def get_corrupted_input(input, corruption_level):
    corrupted = np.copy(input)
    inv = np.random.binomial(n=1, p=corruption_level, size=len(input))
    for i, v in enumerate(input):
        if inv[i]:
            corrupted[i] = -1 * v
    return corrupted

```

Resizing the input patterns to 64*64 pixels

Threshold, i.e., increasing b/w contrast for input patterns

Flatten, reshaping to final prediction size

iv) Plotting the images –

hopfield.py × hofield.png

```python
plt.figure(figsize=(15,10))
i=0
for image in data :
  noisyVec = get_corrupted_input(image, 0.25)

  coefMatrix = trainer(image)
  predictedVec = prediction(noisyVec,coefMatrix)
  vector_list.append(image)
  noisy_list.append(noisyVec)
  predictedVec_list.append(predictedVec)

  plt.subplot(8,4,i+1)
  plt.imshow(image)
  if i==0 :

    plt.title('imported picture')
  plt.subplot(8,4,i+2)
  plt.imshow(coefMatrix)
  if i==0 :

    plt.title('coefMatrix picture')
  #plt.subplot(1,4,2)
  #plt.imshow(vector);
  #plt.title('cleaned and croped picture')
  plt.subplot(8,4,i+3)
  plt.imshow(noisyVec);
  if i==0 :
    plt.title(' zero noisy picture')
  plt.subplot(8,4,i+4)
  plt.imshow(predictedVec);
  if i==0 :
    plt.title('recalled picture')
  i=i+4
plt.savefig('hofield.png')
plt.show()
```

v) Script output in command window –

```
!python3 /content/hopfield.py

/content/hopfield.py:106: FutureWarning: Non RGB image conversion is now deprecated. For RGBA images, please use rg
  zero = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/0.png','w')).copy()
/content/hopfield.py:107: FutureWarning: Non RGB image conversion is now deprecated. For RGBA images, please use rg
  one = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/1.png','w')).copy()
/content/hopfield.py:108: FutureWarning: Non RGB image conversion is now deprecated. For RGBA images, please use rg
  two = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/2.png','w')).copy()
/content/hopfield.py:109: FutureWarning: Non RGB image conversion is now deprecated. For RGBA images, please use rg
  three = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/3.png','w')).copy()
/content/hopfield.py:110: FutureWarning: Non RGB image conversion is now deprecated. For RGBA images, please use rg
  four = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/4.png','w')).copy()
/content/hopfield.py:111: FutureWarning: Non RGB image conversion is now deprecated. For RGBA images, please use rg
  six = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/6.png','w')).copy()
/content/hopfield.py:112: FutureWarning: Non RGB image conversion is now deprecated. For RGBA images, please use rg
  dash = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/-.png','w')).copy()
/content/hopfield.py:113: FutureWarning: Non RGB image conversion is now deprecated. For RGBA images, please use rg
  nine = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/9.png','w')).copy()
Start to data preprocessing...
<Figure size 640x480 with 0 Axes>
<Figure size 1500x1000 with 32 Axes>
```

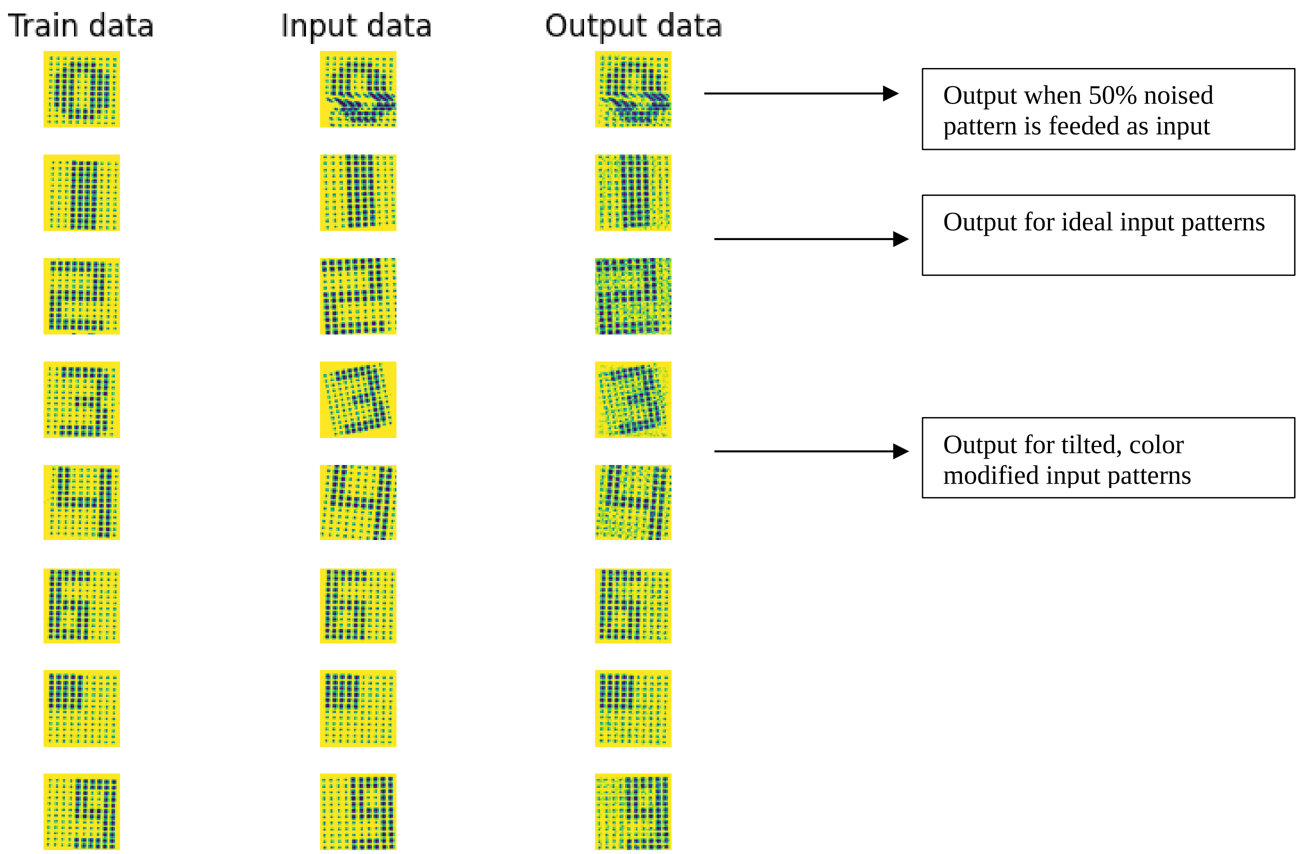
✓ 2m 59s completed at 12:26 AM

Time taken for entire process ~ 3 mins.

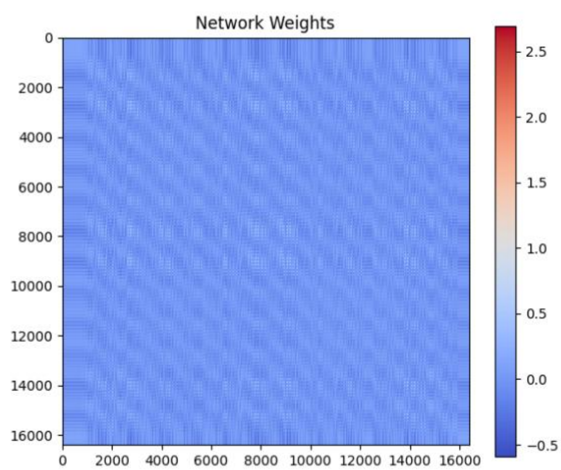
3. Asynchronous learning:

In Asynchronous learning, only one unit is updated at a time. This unit can be picked at random, or a pre-defined order can be imposed from the very beginning. Both the cases of noisy and clean input patterns are executed together, and the result is as below –

i) Learning results with given dotted input patterns –

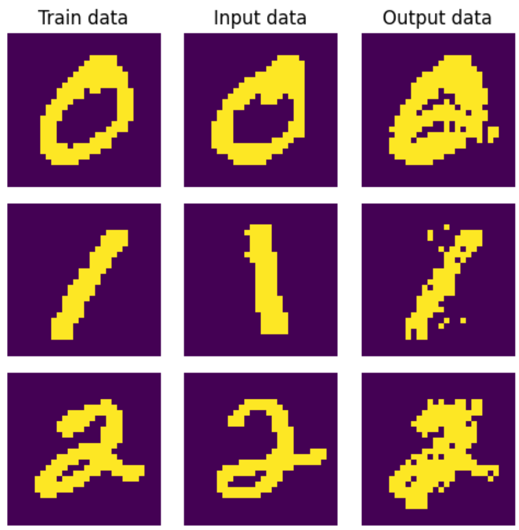


Weight matrix for above –

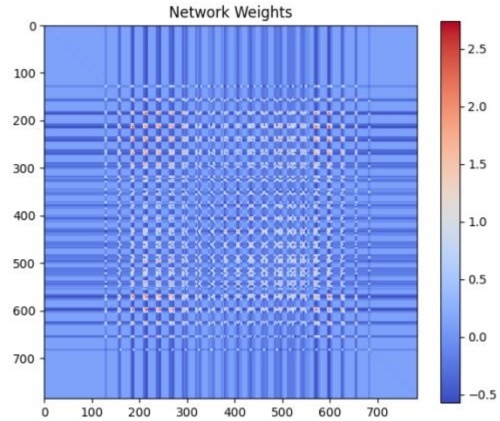


For the dotted input patterns, there was a slight variance in recalled or output image due to number of dots not being able to be recognized, fit into the pixels. So, another case is taken, where input patterns are not in dotted format but instead in thick patterns.

ii) Learning results with varied input pattern for better recalled output –



Weight matrix for above –



CORRESPONDING CODE EXPLANATION FOR ASYNCHRONOUS LEARNING PROCEDURE SHOWN ABOVE:

train_mnist.py × result_mnist.png weights.png

```python
import numpy as np
from matplotlib import pyplot as plt
from skimage.filters import threshold_mean
import network
from keras.datasets import mnist
from skimage.color import rgb2gray
from skimage.transform import resize
import matplotlib.image as img


# Utils
def reshape(data):
    dim = int(np.sqrt(len(data)))
    data = np.reshape(data, (dim, dim))
    return data

def plot(data, test, predicted, figsize=(3, 3)):
    data = [reshape(d) for d in data]
    test = [reshape(d) for d in test]
    predicted = [reshape(d) for d in predicted]

    fig, axarr = plt.subplots(len(data), 3, figsize=figsize)
    for i in range(len(data)):
        if i==0:
            axarr[i, 0].set_title('Train data')
            axarr[i, 1].set_title("Input data")
            axarr[i, 2].set_title('Output data')

        axarr[i, 0].imshow(data[i])
        axarr[i, 0].axis('off')
        axarr[i, 1].imshow(test[i])
        axarr[i, 1].axis('off')
        axarr[i, 2].imshow(predicted[i])
        axarr[i, 2].axis('off')

    plt.tight_layout()
    plt.savefig("result_mnist.png")
    plt.show()
```

Reshaping given data

Plotting train, input and output data

train_mnist.py × result_mnist.png weights.png

```python
def preprocessing(img):
    print(img.shape)
    w, h = img.shape
    # Thresholding
    thresh = threshold_mean(img)
    binary = img > thresh
    shift = 2*(binary*1)-1 # Boolian to int

    # Reshape
    flatten = np.reshape(shift, (w*h))
    return flatten

def preprocessing1(img, w=128, h=128):
    # Resize image
    print(img.shape)
    img = resize(img, (w,h), mode='reflect')
    print(img.shape)
    # Thresholding
    thresh = threshold_mean(img)
    binary = img > thresh
    shift = 2*(binary*1)-1 # Boolian to int
    print(shift.shape, (w*h))
    # Reshape
    flatten = np.reshape(shift, (w*h))
    return flatten

def main():
    # Load data
    zero = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/0.png','w')).copy()
    one = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/1.png','w')).copy()
    two = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/2.png','w')).copy()
    three = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/3.png','w')).copy()
    four = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/4.png','w')).copy()
    six = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/6.png','w')).copy()
    dash = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/-.png','w')).copy()
    nine = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/9.png','w')).copy()

    zerooo = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/zeroooo.JPEG','w')).copy()
    onee = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/test/one.JPEG','w')).copy()
```

Preprocessing the data by adjusting size of pixels, b/w contrast

Loading our input patterns

train_mnist.py × result_mnist.png weights.png

```python
    dash = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/-.png','w')).copy()
    nine = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/9.png','w')).copy()

    zerooo = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/zeroooo.JPEG','w')).copy()
    onee = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/test/one.JPEG','w')).copy()
    twoo = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/test/twoo.JPEG','w')).copy()
    threee = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/test/threee.JPEG','w')).copy()
    fourrr = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/test/four_test.jpg','w')).copy()
    sixx = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/test/Sixx.jpg','w')).copy()
    dashh = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/-.png','w')).copy()
    ninee = rgb2gray(img.imread('/content/drive/MyDrive/ass 3 nn/test/9.jpg','w')).copy()


    x_train = [zero,one,two,three, four, six, dash,nine]
    x_test = [zerooo,onee,twoo,threee,fourrr,sixx,dashh,ninee]


    data = []

    # Preprocessing
    print("Start to data preprocessing...")
    data = [preprocessing1(d) for d in x_train]

    # Create Hopfield Network Model
    model = network.HopfieldNetwork()
    model.train_weights(data)

    # Make test datalist
    test = []
    test = [preprocessing1(d) for d in x_test]

    predicted = model.predict(test, threshold=100, asyn=True)
    print("Show prediction results...")
    plot(data, test, predicted, figsize=(5, 5))
    print("Show network weights matrix...")
    model.plot_weights()

if __name__ == '__main__':
    main()
```

Loading error patterns

Train, test the patterns

Create the Hopfield model

Setting “asynchronous” to true

Script output in command window –

\+ Code + Text RAM Disk Editing

```
(312, 312)
(128, 128)
(128, 128) 16384
(312, 312)
(128, 128)
(128, 128) 16384
Start to train weights...
tcmalloc: large alloc 2147483648 bytes == 0x55dbeaae4000 @  0x7f92db982001 0x7f92d92d11af 0x7f92d9327c23 0x7f92d9328a87 0x7f92d93ca823 0x5
  0% 0/8 [00:00<?, ?it/s]tcmalloc: large alloc 2147483648 bytes == 0x55dc6aae4000 @  0x7f92db9801e7 0x7f92d92d10ce 0x7f92d9327cf5 0x7f92d9
 12% 1/8 [00:01<00:08,  1.21s/it]tcmalloc: large alloc 2147483648 bytes == 0x55dc6aae4000 @  0x7f92db9801e7 0x7f92d92d10ce 0x7f92d9327cf5
 25% 2/8 [00:02<00:06,  1.02s/it]tcmalloc: large alloc 2147483648 bytes == 0x55dc6aae4000 @  0x7f92db9801e7 0x7f92d92d10ce 0x7f92d9327cf5
 38% 3/8 [00:02<00:04,  1.04it/s]tcmalloc: large alloc 2147483648 bytes == 0x55dc6aae4000 @  0x7f92db9801e7 0x7f92d92d10ce 0x7f92d9327cf5
 50% 4/8 [00:03<00:03,  1.06it/s]tcmalloc: large alloc 2147483648 bytes == 0x55dc6aae4000 @  0x7f92db9801e7 0x7f92d92d10ce 0x7f92d9327cf5
100% 8/8 [00:07<00:00,  1.08it/s]
(312, 312)
(128, 128)
(128, 128) 16384
(294, 294)
(128, 128)
(128, 128) 16384
(286, 287)
(128, 128)
(128, 128) 16384
(2513, 2653)
(128, 128)
(128, 128) 16384
(287, 287)
(128, 128)
(128, 128) 16384
(312, 312)
(128, 128)
(128, 128) 16384
(312, 312)
(128, 128)
(128, 128) 16384
(298, 298)
(128, 128)
(128, 128) 16384
Start to predict...
100% 8/8 [00:39<00:00,  4.94s/it]
Show prediction results...
<Figure size 500x500 with 24 Axes>
Show network weights matrix...
<Figure size 600x500 with 2 Axes>
```

✓ 1m 8s completed at 10:30 AM

Time taken for entire process ~ 1min

The core script for the network is as below:

network.py

```python

import numpy as np
from matplotlib import pyplot as plt
import matplotlib.cm as cm
from tqdm import tqdm

class HopfieldNetwork(object):
    def train_weights(self, train_data):
        print("Start to train weights...")
        num_data =  len(train_data)
        self.num_neuron = train_data[0].shape[0]

        # initialize weights
        W = np.zeros((self.num_neuron, self.num_neuron))
        rho = np.sum([np.sum(t) for t in train_data]) / (num_data*self.num_neuron)

        # Hebb rule
        for i in tqdm(range(num_data)):
            t = train_data[i] - rho
            W += np.outer(t, t)

        # Make diagonal element of W into 0
        diagW = np.diag(np.diag(W))
        W = W - diagW
        W /= num_data

        self.W = W

    def predict(self, data, num_iter=20, threshold=0, asyn=False):
        print("Start to predict...")
        self.num_iter = num_iter
        self.threshold = threshold
        self.asyn = asyn

        # Copy to avoid call by reference
        copied_data = np.copy(data)

        # Define predict list
        predicted = []
```

1m 8s completed at 10:30 AM

Comment Share R

RAM Disk Editing

network.py ×

```python
        # Define predict list
        predicted = []
        for i in tqdm(range(len(data))):
            predicted.append(self._run(copied_data[i]))
        return predicted

    def _run(self, init_s):
        if self.asyn==False:
            """
            Synchronous update
            """
            # Compute initial state energy
            s = init_s

            e = self.energy(s)

            # Iteration
            for i in range(self.num_iter):
                # Update s
                s = np.sign(self.W @ s - self.threshold)
                # Compute new state energy
                e_new = self.energy(s)

                # s is converged
                if e == e_new:
                    return s
                # Update energy
                e = e_new
            return s
        else:
            """
            Asynchronous update
            """
            # Compute initial state energy
            s = init_s
            e = self.energy(s)

            # Iteration
            for i in range(self.num_iter):
```

✓ 1m 8s completed at 10:30 AM

Comment Share R

RAM Disk Editing

network.py

```python
            return s
        else:
            """
            Asynchronous update
            """
            # Compute initial state energy
            s = init_s
            e = self.energy(s)

            # Iteration
            for i in range(self.num_iter):
                for j in range(100):
                    # Select random neuron
                    idx = np.random.randint(0, self.num_neuron)
                    # Update s
                    s[idx] = np.sign(self.W[idx].T @ s - self.threshold)

                # Compute new state energy
                e_new = self.energy(s)

                # s is converged
                if e == e_new:
                    return s
                # Update energy
                e = e_new
            return s


    def energy(self, s):
        return -0.5 * s @ self.W @ s + np.sum(s * self.threshold)

    def plot_weights(self):
        plt.figure(figsize=(6, 5))
        w_mat = plt.imshow(self.W, cmap=cm.coolwarm)
        plt.colorbar(w_mat)
        plt.title("Network Weights")
        plt.tight_layout()
        plt.savefig("weights.png")
        plt.show()
```

✓ 1m 8s completed at 10:30 AM

So overall, by computer simulations, we find that the projection rule in synchronous mode maintains a high noise tolerance. But Hopfield model performs the best when the net is operated asynchronously. In asynchronous mode, the patterns are recalled when the input pattern is thick pixels rather than dotted patterns. With respect to processing times for executions, the asynchronous mode is faster than the synchronous.