

Introduction

This assignment presents an association problem – with the task of recognition, using several patterns. The proposed solution employs a discrete-time Hopfield network to recognize patterns given certain initial conditions.

I have employed Jupyter Notebook, which is a web-based interactive environment that allows you to write and execute code. The platform is especially designed and optimized for data science, machine learning, and data analysis applications.

For implementation, I have made use of Python programming language along with data processing and manipulation libraries such as NumPy, pandas, and Python Image Library (PIL). In addition, for the purpose of data visualization – matplotlib & seaborn libraries have been utilized.

The entire ML workflow has been adopted: dataset generation, model building, model evaluation, model optimization & results. These are discussed in depth in the subsequent sections.

Methodology & Justification

I. Hopfield Network

A Hopfield Network is a type of recurrent artificial neural network that is designed for associative memory and pattern recognition tasks. It is a dynamical system that evolves in continuous or discrete time towards an asymptotically stable state, also known as an attractor.

The network consists of binary/bipolar neurons that are fully connected to each other and are trained using a set of patterns that are stored within the network. During training, it employs the Hebbian learning rule to learn weights as the network transitions towards the stable state. This enables the network to retrieve the stored pattern when presented with a partial or noisy version of the pattern, under the recall phase. Furthermore, the network's stability can be gauged using the energy function which decreases or remains with each update, converging to local minimum.

II. Data Generation & Analysis

i. Clean Patterns

The dataset has been generated using list slicing to form each pattern. There are 8 clean patterns as represented in the problem statement, with each having dimensions of 12x10. The patterns are of grey-scale representation, and have been populated using bipolar values i.e., 1 & -1.

These are then converted to arrays and transformed into associated images for viewing. Pixels with values 1 represent light regions and -1 depicts dark areas. Figure 1. represents 'clean' patterns generated.

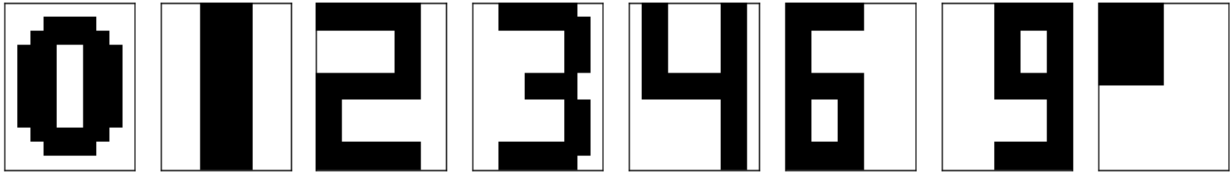


Figure 1.

The table below represents certain characteristics of each pattern that is vital for future reference.

Table 1. Pattern Characteristics

Number of Patterns	Rows per Pattern	Columns per Patterns	Number of Pixels/Neurons
08	12	10	120

Each pattern is independently produced and then stored in a single array to be fed to the network later for training.

ii. Noisy Patterns

As per specifications mentioned in the problem statement, I have generated noisy patterns associated with each clean pattern by introducing random noise. This is achieved by reversing values of 25% of pixels i.e., 30 pixels per pattern.

A similar process is adopted for these patterns, they are collectively stored in a single array for further usage.

Figure 2. depicts one such sample set of noisy patterns generated.

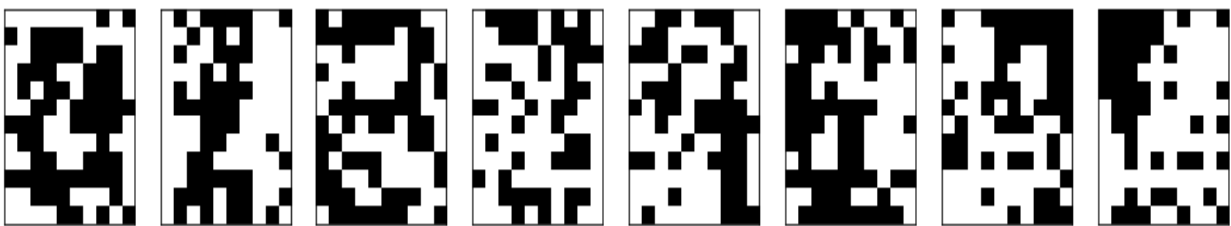


Figure 2.

The dataset has been generated using piece-wise functions to map curvatures of the regions and populated with data points in between the decision surfaces. This was achieved with the aid of python and excel workbook. The excel workbook is imported to the Jupyter environment as a comma separated (csv) file and is then merged with the python generated data points. Figure 1. depicts characteristics of the dataset.

III. Data Preprocessing

Under this section, we look at the preprocessing techniques that were employed to transform the dataset into clean and standardized representation for feeding the network.

I. Bipolar Nature

The output of the neuron, computed using the neuron's activation and a thresholding function, analogous to a neuron's 'firing state.' In this case, -1 and +1. In addition, to aid processing and network performance, the input is also bipolar and represents the same semantics.

```
y_new[y_new >= 0] = 1
y_new[y_new < 0] = -1
```

II. Flattening

Using the *np.flatten()* or *np.reshape(-1)* method of the NumPy library, I have reshaped the original arrays of shape (12,10) to (120,). This enables easier interpretation by the network and allows for faster processing due to its single dimensionality.

```
arr_0 = pattern_0.reshape(-1)
```

IV. Model Building & Evaluation

Under this section, the network is built and evaluated for a number of scenarios. The learning process is unsupervised and based on the principles of correlation i.e., Hebbian learning. In summary, the Hebbian learning rule defines that the weight vector W_i increase in proportion to the product of the input x and the learning signal r .

There are two learning procedures for a Hopfield Network: synchronous and asynchronous learning. The following subsections will evaluate and analyze the network from the perspective of both procedures.

i. Synchronous Learning

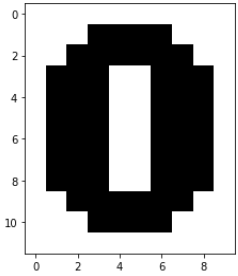
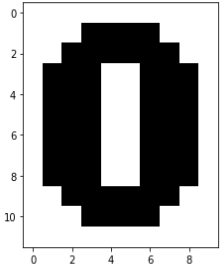
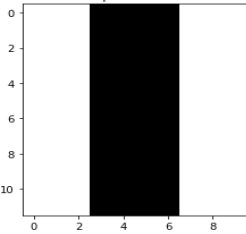
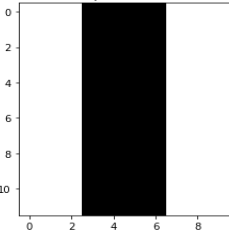
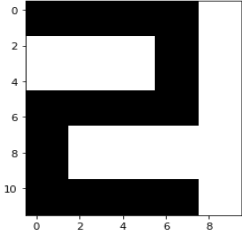
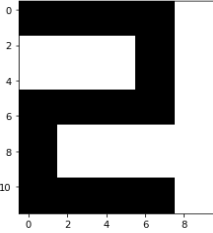
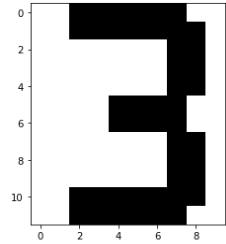
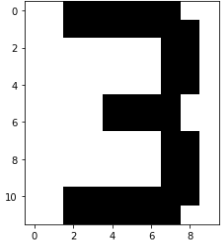
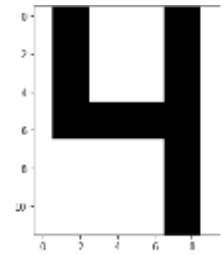
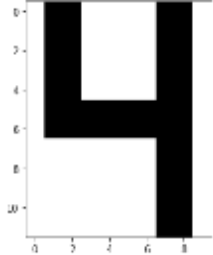
Under synchronous learning, the transition from one state to another is done simultaneously. So, for a given time all elements of the input vector V are updated simultaneously considering the most recent values of changes.

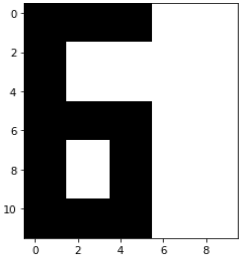
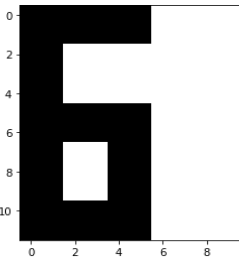
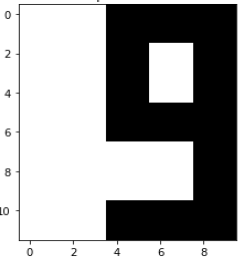
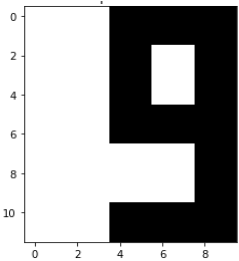
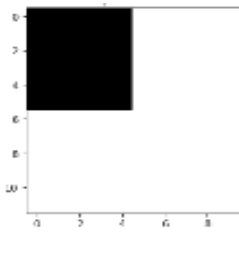
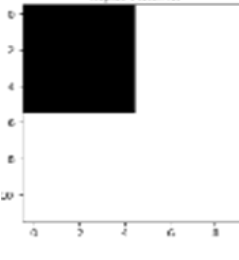
The update rule can be depicted in the form below:

$$v^{k+1} = \Gamma(Wv^k + i - T)$$

The performance of the model is evaluated under the recall phase for all clean patterns. Table 2. represents the metrics recorded:

Table 2. Clean Pattern Recall (Synchronous)

Input Pattern	Associated Pattern	True Pattern	States	Energies
Pattern_0			[1]	[-67.73]
Pattern_1			[1]	[-67.87]
Pattern_2			[1]	[-82.33]
Pattern_3			[1]	[-86.60]
Pattern_4			[1]	[-77.73]

Pattern_6			[1]	[-90.47]
Pattern_9			[1]	[-81.13]
Pattern_dot			[1]	[-66.93]

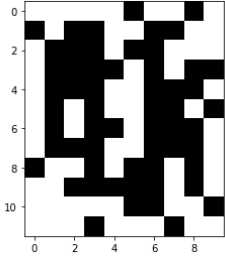
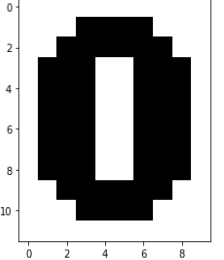
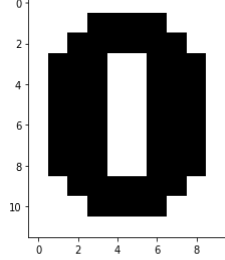
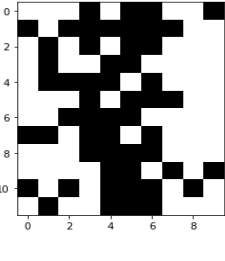
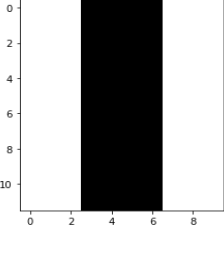
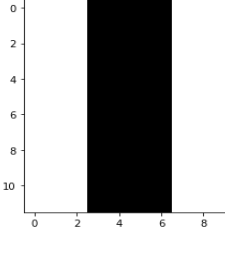
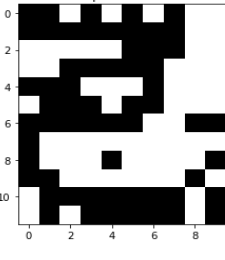
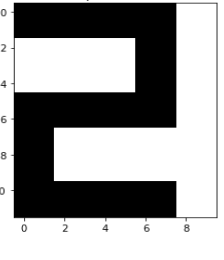
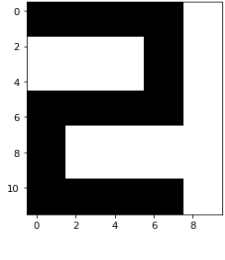
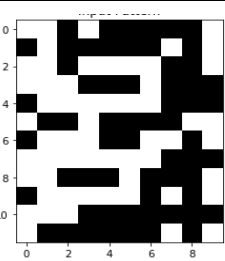
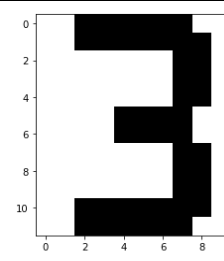
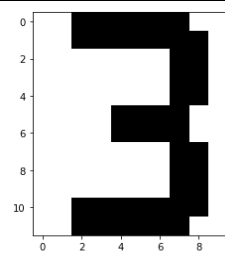
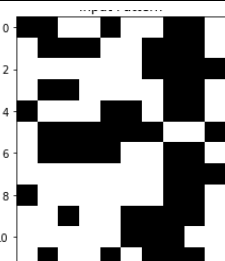
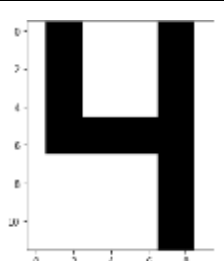
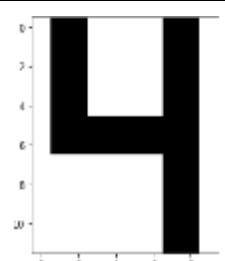
The input patterns are flattened and fed to the network for learning weights for the attractor states. As we can observe, the network is able successfully recall all patterns. We can state the following points about the network:

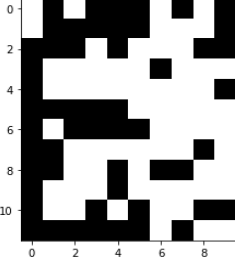
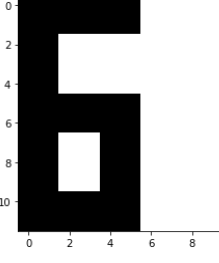
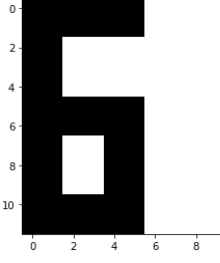
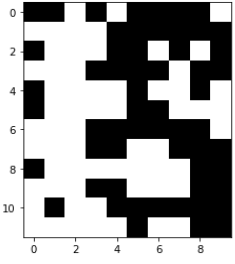
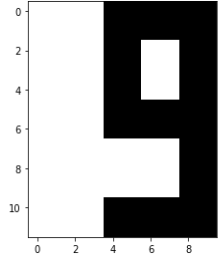
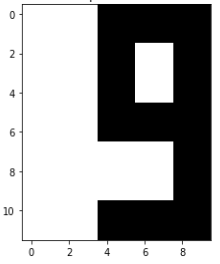
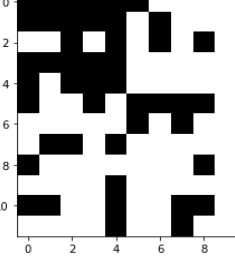
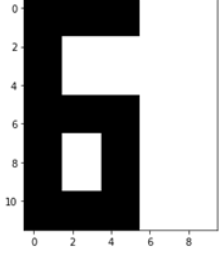
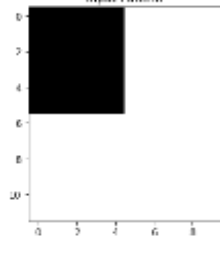
- All patterns have been successfully stored and recalled by the network.
- It takes exactly one iteration per pattern for recall.
- The associated energy function values for each predicted pattern vary and thus represent different convergence/minimum for each pattern.
- The accuracy of the network is 100%.

Now we look at a similar configuration for all noisy patterns i.e., 25% pixels randomly reversed.

Table 3. represents the metrics recorded:

Table 3. Noisy Pattern Recall (Synchronous)

Input Pattern	Associated Pattern	True Pattern	States	Energies
			[3]	[-19.40, -65.73, -67.73]
			[3]	[-17.00, -66.53, -67.87]
			[3]	[-18.93, -81.01, -82.53]
			[3]	[-20.33, -85.20, -86.60]
			[4]	[-21.46, -72.40, -75.10, -77.73]

			[3]	[-22.13, -88.73, -90.47]
			[2]	[-18.87, -81.13]
			[6]	[-17.80, -61.47, -67.73, -80.33, -88.73, -90.47]

We observe the following points about the network's performance:

- All patterns are correctly associated with their clean representations except for the last pattern i.e., pattern_dot, which is incorrectly associated with pattern_6.
- The number of iterations it takes the network to converge to the steady state varies for each pattern, with 2 being the least and 6 being the most.
- We observe that the energy function values decrease after each iteration, thus signaling towards the stability of the network.
- The accuracy of the network is 87.5%.

ii. Asynchronous Learning

Under asynchronous learning, the transition from one state to another is done independently. For a given time, only a single neuron (randomly chosen) is allowed to update its output and only one element of the input vector V is updated considering the most recent values of changes.

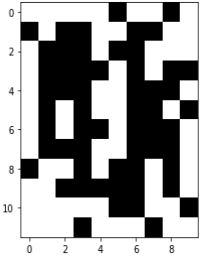
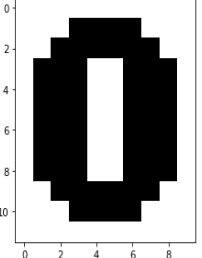
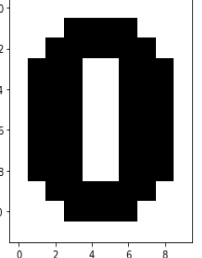
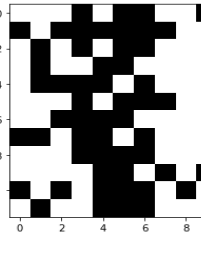
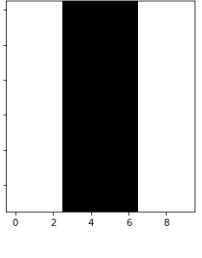
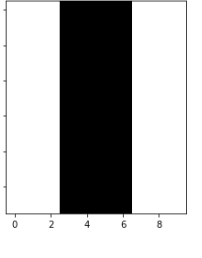
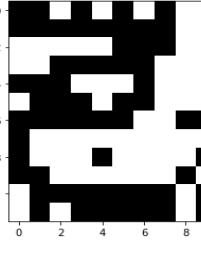
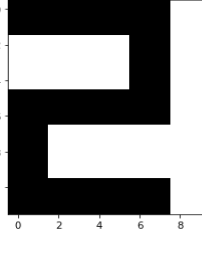
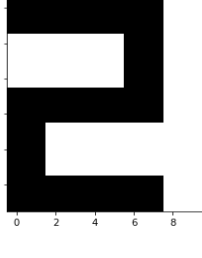
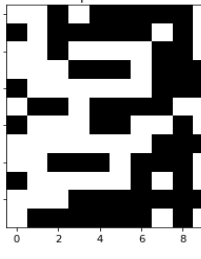
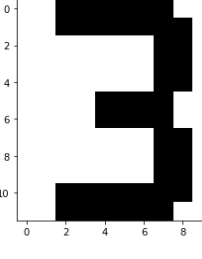
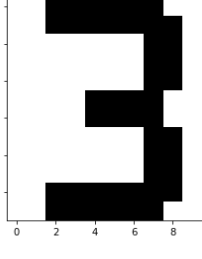
The update rule can be depicted in the form below:

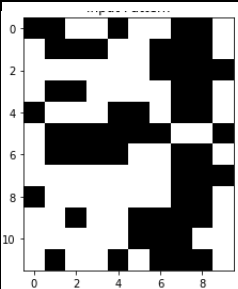
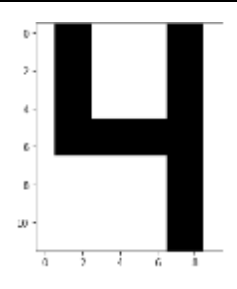
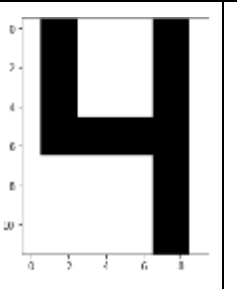
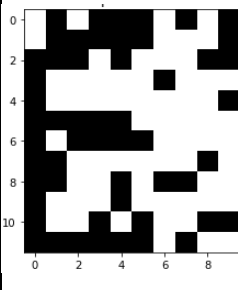
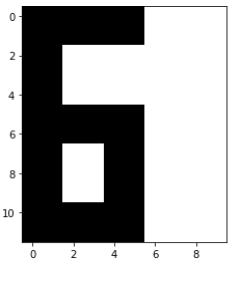
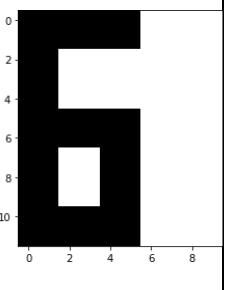
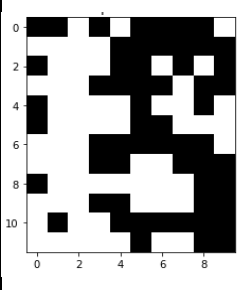
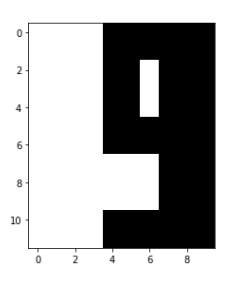
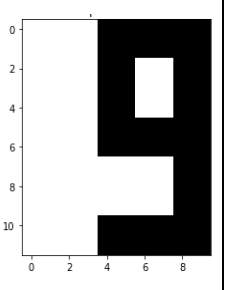
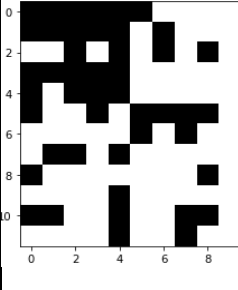
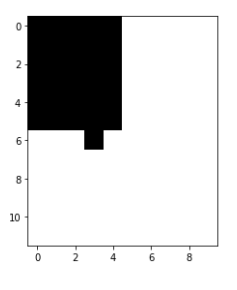
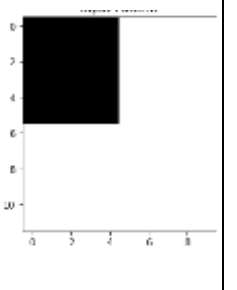
$$v_i^{k+1} = \text{sgn}(w_i^T v^k + i_i + T_i)$$

For clean patterns, the performance of the network is identical to that of the synchronous learning model. All parameter values are the same and patterns can be correctly recalled.

We now observe the characteristics of the asynchronous network for noisy patterns. This is depicted in the table below.

Table 3. Noisy Pattern Recall (Synchronous)

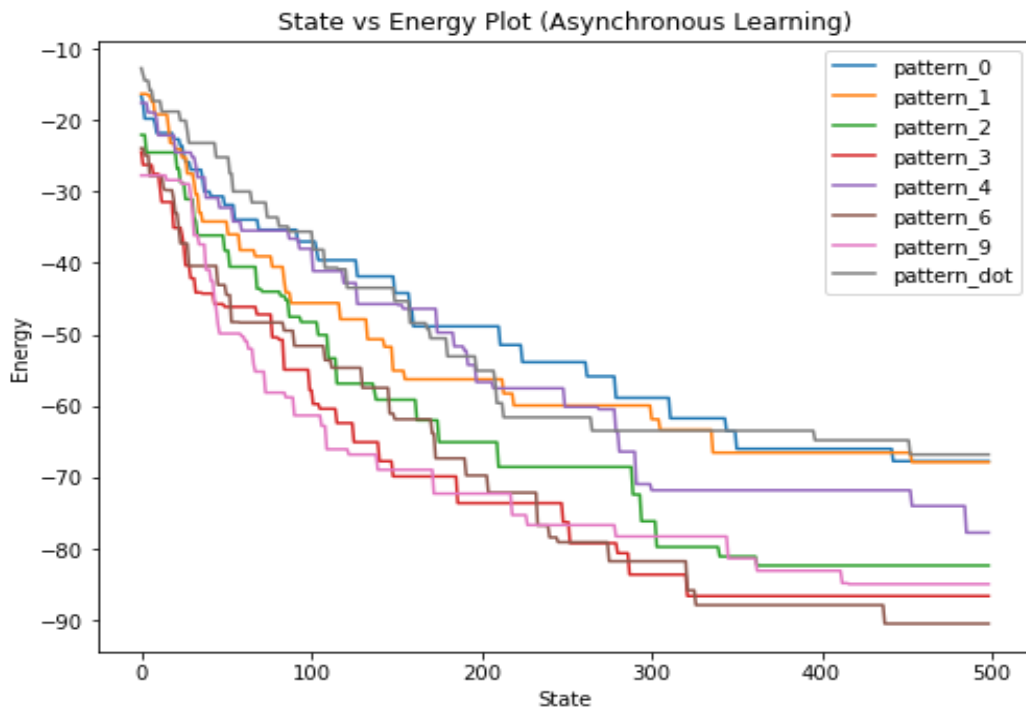
Input Pattern	Associated Pattern	True Pattern	States	Energies
			[500]	[-16.67, -18.4, -19.8, -21.73, -22.6, -24.0, -26.33, -27.33, -28.6, -30.13, -31.93, -33.2, -35.27, -36.4, -37.73, -38.67, -39.6, -42.27, -44.07, -45.6, -48.67, -49.87, -50.0, -51.93, -54.27, -55.8, -57.73, -59.93, -62.4, -64.93, -65.87, -67.73]
			[500]	[-16.53, -16.33, -17.33, -19.2, -21.07, -23.2, -24.07, -25.2, -27.47, -28.67, -30.4, -32.93, -34.2, -36.0, -38.2, -39.07, -40.53, -42.53, -44.13, -45.6, -47.87, -50.67, -51.73, -55.07, -56.27, -58.27, -59.93, -61.87, -63.33, -66.53, -67.87]
			[500]	[-22.07, -24.53, -26.8, -28.73, -28.93, -31.07, -33.6, -34.13, -36.13, -38.27, -40.53, -44.6, -43.6, -44.0, -45.2, -47.53, -48.27, -50.07, -53.33, -56.87, -57.87, -59.13, -62.0, -65.07, -68.53, -72.4, -76.13, -79.73, -81.07, -82.33]
			[500]	[-24.47, -26.27, -27.53, -29.47, -31.47, -35.07, -36.2, -37.6, -40.33, -42.13, -44.27, -44.13, -45.73, -46.13, -47.2, -50.73, -50.4, -54.93, -57.8, -59.73, -60.4, -62.4, -65.07, -67.73, -69.87, -73.6, -76.27, -79.2, -80.6, -83.6, -86.6]

			[500]	[-17.6, -18.93, -19.93, -22.13, -24.53, -25.2, -26.33, -28.0, -30.87, -32.27, -34.2, -35.47, -36.6, -38.0, -41.13, -42.8, -45.87, -45.73, -46.4, -49.73, -51.67, -52.27, -54.27, -56.67, -57.53, -60.47, -60.13, -63.6, -66.4, -70.93, -71.8, -74.0, -77.73]
			[500]	[-23.93, -24.93, -27.8, -29.8, -29.47, -31.07, -33.07, -35.07, -37.27, -40.4, -43.07, -44.47, -48.33, -48.27, -49.47, -51.6, -53.6, -54.67, -57.47, -61.87, -61.07, -63.87, -67.33, -69.73, -72.13, -76.73, -78.4, -79.07, -81.73, -85.8, -87.87, -90.47]
			[500]	[-27.73, -28.4, -28.93, -30.53, -32.47, -36.13, -37.4, -40.93, -42.53, -45.07, -46.4, -50.93, -48.53, -49.87, -50.27, -52.0, -54.13, -55.2, -58.8, -58.13, -61.33, -62.8, -66.8, -66.07, -68.93, -72.27, -75.27, -76.67, -78.27, -81.33, -84.93, -83.07, -84.8]
			[500]	[-12.73, -13.6, -14.47, -15.87, -17.33, -18.8, -20.07, -21.47, -23.2, -25.2, -27.47, -30.0, -31.53, -33.6, -34.87, -35.6, -38.13, -40.67, -40.87, -43.47, -45.33, -48.4, -49.2, -50.53, -53.07, -55.07, -56.6, -59.6, -61.6, -63.47, -64.8, -66.8]

The following observations are deduced:

- The network requires a large number of iterations i.e., 500 as an average value for it to converge to an attractor state.
- All patterns are correctly associated with minimalistic inaccuracy i.e., incorrect pixel or two.
- The network can get stuck at one of the many local minima, for multiple iterations.
- The energy associated with each state successively decreases or remains constant, reaffirming the model's convergence to stability.
- The process is much more time-consuming due to its asynchronous nature.

The line plots for energy associated with each state, for all patterns. The graph below enables a clear representation of this relationship.



V. Conclusion

To conclude, the Hopfield Network was impressive in terms of storing eight patterns and largely successful in retrieving them during the recall phase for both clean and noisy patterns. In addition, this reaffirms the findings of literature that states, 'a network is able to learn $0.15n$ number of patterns, where n is the number of neurons of the network'. Furthermore, we also observed that due to the bipolar nature of the activation function, bipolar input resulted in greater accuracy and improved performance. We were also able to draw a comparative analysis between the two learning approaches: synchronous and asynchronous, with both offering their own set of advantages and disadvantages. Finally, we saw how the network was able to cope with randomly induced noise and was able to converge towards the steady state solutions.

VI. Appendix

```
#Importing requisite libraries
import numpy as np
import pandas as pd
import array
import matplotlib.pyplot as plt
import seaborn as sns
from PIL import Image

#Dataset Generation

#Pattern 0
pattern_0 = np.full(shape=(12, 10), fill_value=1)
pattern_0[1][3:7] = -1
pattern_0[2][2:8] = -1
pattern_0[3][1:4] = pattern_0[3][6:9] = -1
pattern_0[4][1:4] = pattern_0[4][6:9] = -1
pattern_0[5][1:4] = pattern_0[5][6:9] = -1
pattern_0[6][1:4] = pattern_0[6][6:9] = -1
pattern_0[7][1:4] = pattern_0[7][6:9] = -1
pattern_0[8][1:4] = pattern_0[8][6:9] = -1
pattern_0[9][2:8] = -1
pattern_0[10][3:7] = -1

#Pattern 1
pattern_1 = np.full(shape=(12, 10), fill_value=1)
pattern_1[0][3:7] = -1
pattern_1[1][3:7] = -1
pattern_1[2][3:7] = -1
pattern_1[3][3:7] = -1
pattern_1[4][3:7] = -1
pattern_1[5][3:7] = -1
pattern_1[6][3:7] = -1
pattern_1[7][3:7] = -1
pattern_1[8][3:7] = -1
pattern_1[9][3:7] = -1
pattern_1[10][3:7] = -1
pattern_1[11][3:7] = -1

#Pattern 2
pattern_2 = np.full(shape=(12, 10), fill_value=1)
pattern_2[0][0:8] = -1
pattern_2[1][0:8] = -1
pattern_2[2][6:8] = -1
pattern_2[3][6:8] = -1
pattern_2[4][6:8] = -1
```

```

pattern_2[5][0:8] = -1
pattern_2[6][0:8] = -1
pattern_2[7][0:2] = -1
pattern_2[8][0:2] = -1
pattern_2[9][0:2] = -1
pattern_2[10][0:8] = -1
pattern_2[11][0:8] = -1
#Pattern 3
pattern_3 = np.full(shape=(12, 10), fill_value=1)
pattern_3[0][2:8] = -1
pattern_3[1][2:9] = -1
pattern_3[2][7:9] = -1
pattern_3[3][7:9] = -1
pattern_3[4][7:9] = -1
pattern_3[5][4:8] = -1
pattern_3[6][4:8] = -1
pattern_3[7][7:9] = -1
pattern_3[8][7:9] = -1
pattern_3[9][7:9] = -1
pattern_3[10][2:9] = -1
pattern_3[11][2:8] = -1
#Pattern 4
pattern_4 = np.full(shape=(12, 10), fill_value=1)
pattern_4[0][1:3] = pattern_4[0][7:9] = -1
pattern_4[1][1:3] = pattern_4[1][7:9] = -1
pattern_4[2][1:3] = pattern_4[2][7:9] = -1
pattern_4[3][1:3] = pattern_4[3][7:9] = -1
pattern_4[4][1:3] = pattern_4[4][7:9] = -1
pattern_4[5][1:9] = -1
pattern_4[6][1:9] = -1
pattern_4[7][7:9] = -1
pattern_4[8][7:9] = -1
pattern_4[9][7:9] = -1
pattern_4[10][7:9] = -1
pattern_4[11][7:9] = -1
#Pattern 6
pattern_6 = np.full(shape=(12, 10), fill_value=1)
pattern_6[0][0:6] = -1
pattern_6[1][0:6] = -1
pattern_6[2][0:2] = -1
pattern_6[3][0:2] = -1
pattern_6[4][0:2] = -1
pattern_6[5][0:6] = -1
pattern_6[6][0:6] = -1
pattern_6[7][0:2] = pattern_6[7][4:6] = -1

```

```

pattern_6[8][0:2] = pattern_6[8][4:6] = -1
pattern_6[9][0:2] = pattern_6[9][4:6] = -1
pattern_6[10][0:6] = -1
pattern_6[11][0:6] = -1
#Pattern 9
pattern_9 = np.full(shape=(12, 10), fill_value=1)
pattern_9[0][4:] = -1
pattern_9[1][4:] = -1
pattern_9[2][4:6] = pattern_9[2][8:] = -1
pattern_9[3][4:6] = pattern_9[3][8:] = -1
pattern_9[4][4:6] = pattern_9[4][8:] = -1
pattern_9[5][4:] = -1
pattern_9[6][4:] = -1
pattern_9[7][8:] = -1
pattern_9[8][8:] = -1
pattern_9[9][8:] = -1
pattern_9[10][4:] = -1
pattern_9[11][4:] = -1
#Pattern dot
pattern_dot = np.full(shape=(12, 10), fill_value=1)
pattern_dot[0][0:5] = -1
pattern_dot[1][0:5] = -1
pattern_dot[2][0:5] = -1
pattern_dot[3][0:5] = -1
pattern_dot[4][0:5] = -1
pattern_dot[5][0:5] = -1
#Combining individual patterns into an array
arr_0 = pattern_0.reshape(-1)
arr_1 = pattern_1.reshape(-1)
arr_2 = pattern_2.reshape(-1)
arr_3 = pattern_3.reshape(-1)
arr_4 = pattern_4.reshape(-1)
arr_6 = pattern_6.reshape(-1)
arr_9 = pattern_9.reshape(-1)
arr_dot = pattern_dot.reshape(-1)
patterns_bipolar = list()
patterns_bipolar.append(arr_0)
patterns_bipolar.append(arr_1)
patterns_bipolar.append(arr_2)
patterns_bipolar.append(arr_3)
patterns_bipolar.append(arr_4)
patterns_bipolar.append(arr_6)
patterns_bipolar.append(arr_9)
patterns_bipolar.append(arr_dot)
patterns_bipolar = np.array(patterns_bipolar)

```

#Data Visualization

```
def graph_plotter(patterns, height=12, width=10):
    num_of_patterns = len(patterns)
    fig, ax = plt.subplots(1, num_of_patterns, figsize=(15,8))
    for i in range(0,num_of_patterns):
        ax[i].matshow(patterns[i].reshape((height, width)), cmap='gray')
        ax[i].set_xticks([])
        ax[i].set_yticks([])
graph_plotter(patterns_bipolar)

def noise(patterns,n=120, nr_of_flips=30):# 25% of pixels
    corrupted_patterns = list()
    for pattern in patterns:
        array = np.copy(pattern)
        idx_reassignment = np.random.choice(n, nr_of_flips, replace=False)#
        pick no of pixels to flip (without replacement)
        for i in (idx_reassignment):
            if array[i]==-1:
                array[i]=1
            else:
                array[i]=-1
        corrupted_patterns.append(array)
    return corrupted_patterns

noisy_patterns = noise(patterns_bipolar)
graph_plotter(noisy_patterns)
```

#Model Building and Training

```
class HopfieldNetwork():

    def __init__(self,patterns):

        self.patterns = patterns
        self.n = self.patterns.shape[1] #number of neurons
        self.weights = np.zeros((self.n,self.n)) #initializing weights matrix
        self.energies = {'State': [], 'Energy':[]} #energy container to keep
        record of energies of each state

    def training(self):

        self.weights = np.dot(self.patterns.T,self.patterns)/self.n #updating
        weights matrix
        np.fill_diagonal(self.weights,0)
```

```

def recall(self, x, learning = 'syn', max_iterations = 500):
    y = np.copy(x)
    if learning == 'syn': #learning procedure i.e., synchronous or
asynchronous
        output = self.syn_update(y, max_iterations)
    else:
        output = self.asyn_update(y, max_iterations)
    print("Learning Mode: {}".format(learning))
    return output

def asyn_update(self, y, max_iterations):
    for i in range(1,max_iterations):
        temp = y
        self.rand_index = np.random.randint(0,self.n) #select random
neuron
        activation = np.dot(self.weights[self.rand_index,:],
y)#asynchronous update of neurons
        self.compute_energy(y,i) #compute energy for each state
        #threshold function for binary state change
        if activation < 0:
            y[self.rand_index] = -1
        else:
            y[self.rand_index] = 1
    return y

def syn_update(self, y, max_iterations):
    for i in range(1,max_iterations):
        y_new = np.dot(self.weights, y) #synchronous update of neurons
        self.compute_energy(y,i) #compute energy for each state
        #threshold function for binary state change
        y_new[y_new >= 0] = 1
        y_new[y_new < 0] = -1
        if np.array_equal(y_new, y):
            break
        else:
            y=y_new
    return y

def compute_energy(self, y, iteration):

    self.energy = -0.5*np.dot(np.dot(y.T,self.weights),y)#Energy formula
    self.energies['Energy'].append(self.energy)
    self.energies['State'].append(iteration)

```

```

#Model Evaluation

H_net = HopfieldNetwork(patterns_bipolar)
H_net.training()
y = H_net.recall(noisy_patterns[7], 'asyn')

fig = plt.figure(figsize=(12, 10))

fig.add_subplot(1, 3, 1)
plt.imshow(noisy_patterns[7].reshape((12, 10)), cmap='gray')
plt.title("Input Pattern")

fig.add_subplot(1, 3, 2)
plt.imshow(y.reshape((12, 10)), cmap='gray')
plt.title("Associated Pattern")

fig.add_subplot(1, 3, 3)
plt.imshow(patterns_bipolar[7].reshape((12, 10)), cmap='gray')
plt.title("True Pattern")

print(H_net.energies)

```