# DHN_Zamani

September 4, 2024

`**Zahra Zamani --- University of Tehran 2020**`

**Discrete HopField Network**

*Image Recovery Application*

## 1 Brief Introduction:

The Discrete Hopfield Network (DHN) is a type of recurrent neural network, named after John Hopfield in 1982, that is used for associative memory and optimization tasks. It consists of a set of interconnected neurons, each of which can take on only two discrete states: 0 or 1.

Hopfield's paper, **"Neural networks and physical systems with emergent collective computational abilities,"** marked a significant turning point in the development of neural networks.

## 2 Basics:

The DHN operates on the principle of **energy minimization**. Each neuron in the network has an associated energy value, and the network's goal is to find a configuration of neuron states that minimizes the total energy of the system. This is achieved through a process of iterative updates, where the state of each neuron is updated based on the states of its connected neighbors.

## 3 Image Memorization Application:

To remember an image using a DHN, the image is first converted into a binary pattern, where each pixel is represented by a 0 or 1. This pattern is then used to train the network by setting the weights between neurons. The weights are set such that the desired pattern corresponds to a minimum energy state of the network.

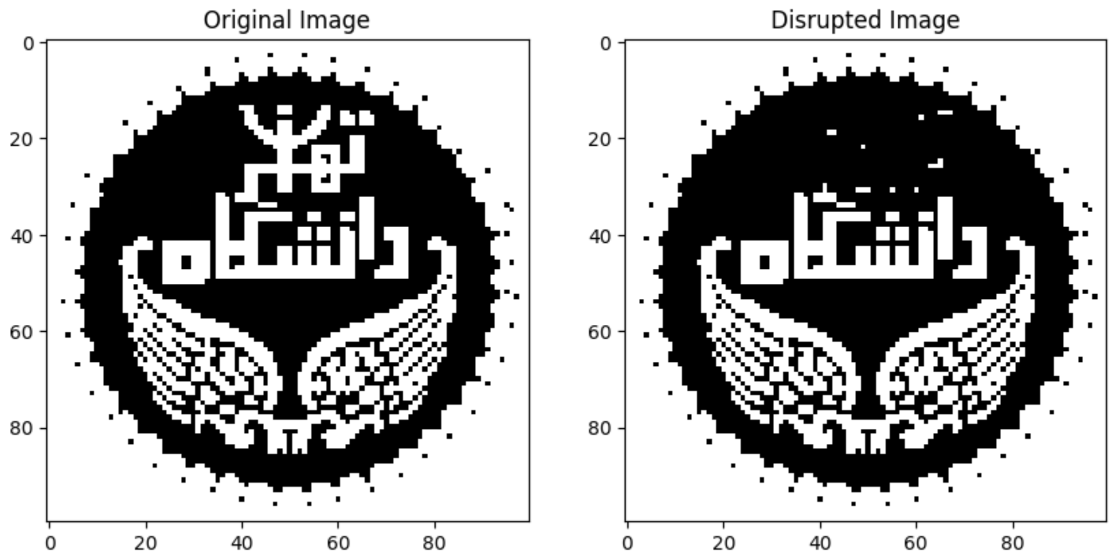
Once the network is trained, it can be used to recall the image by providing it with a noisy or incomplete version of the original pattern. The network will then iteratively update the neuron states until it reaches a stable configuration that corresponds to the stored pattern.

## 4 Implementation:

---

[8]:
```
import numpy as np
from PIL import Image
import os
```

```python
import matplotlib.pyplot as plt

# Set the threshold value
threshold = 180

# path = os.getcwd() # --> Local Path
path = '/content'  # Colab Path

path_train = path+ '/UT_Train.png'
path_test  = path+ '/UT_Test.png'

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.float64)
    x[img_train_array > threshold] = 1
    x[x==0] = -1

    return x

# Read images
x = read_binarize_img(path_train)
y = read_binarize_img(path_test)

# Normalize the image data to the range of 0 to 1
x_normalized = np.clip(x, 0, 1)
y_normalized = np.clip(y, 0, 1)

# Plot images
plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(np.repeat(x_normalized[:,:,np.newaxis], repeats=3, axis=2))
plt.title('Original Image')
plt.subplot(1, 2, 2)
plt.imshow(np.repeat(y_normalized[:,:,np.newaxis], repeats=3, axis=2))
plt.title('Disrupted Image')
plt.show()
```

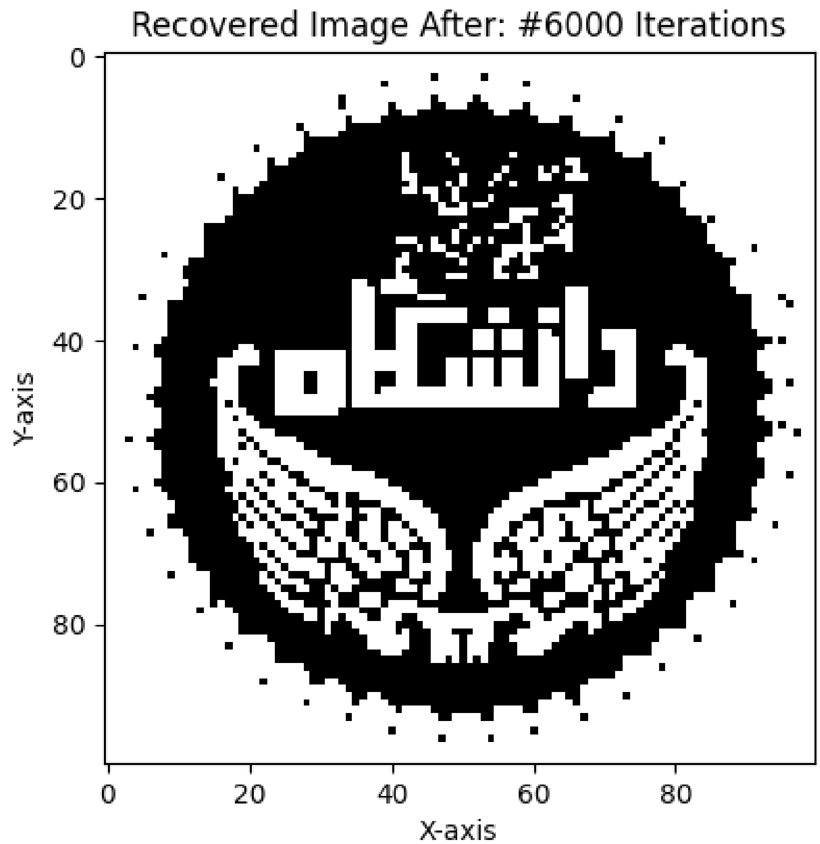


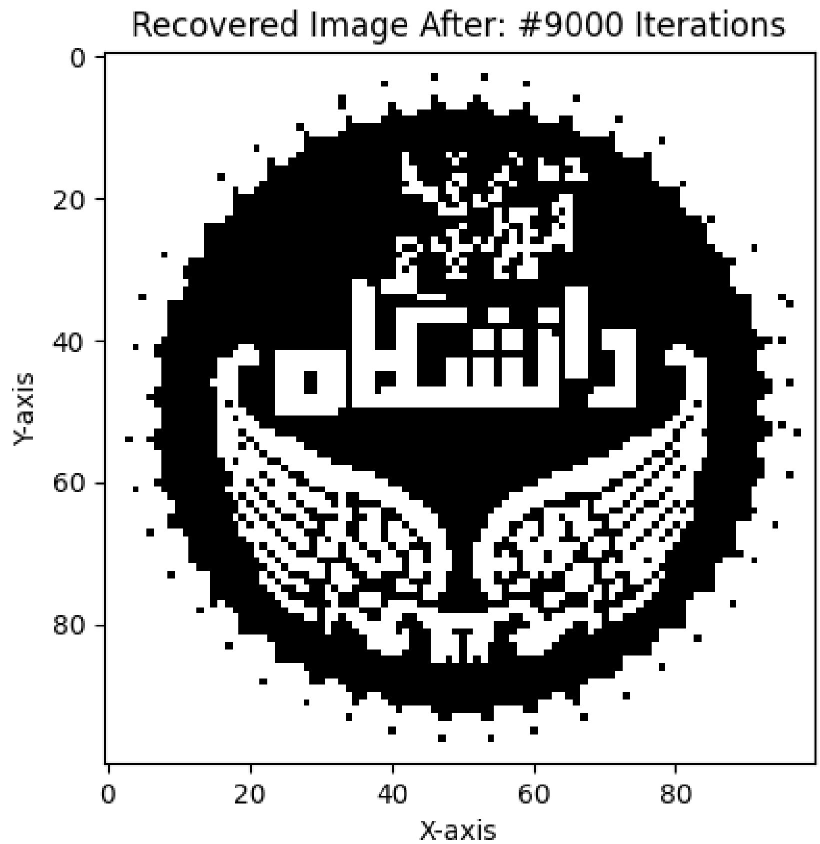

[9]:
```python
x = x.reshape(1, 10000)
y = y.reshape(1, 10000)
# Create the weights matrix
w  = w = np.matmul(np.transpose(2*x-1),2*x-1) - np.eye(len(x[0,:]))
# - 2*x-1: Converts the binary image (x) with values -1 and 1 into a range of -1 to 1.
# - np.transpose(2*x-1): Transposes the training image,
#   essentially creating connections between each pair of neurons

# - np.matmul(np.transpose(2*x-1),2*x-1):
# Performs matrix multiplication between the transposed and original training image,
# capturing the co-occurrence of active neurons in the image.
# High co-occurrence strengthens the connection (weight) between those neurons.

# - np.eye(len(x[0,:])): Subtracts the identity matrix to prevent self-connections from influencing the network update process.



# - Iterative Updates:
#   This section simulates the DHN's iterative process.
#   It starts with a noisy or incomplete version of the training image (y).
#   Here, y is initialized as zeros.
theta = 0.5
y_in = 0
for iter in range(30001):
    i = np.random.choice(10000,1,replace = False)
```

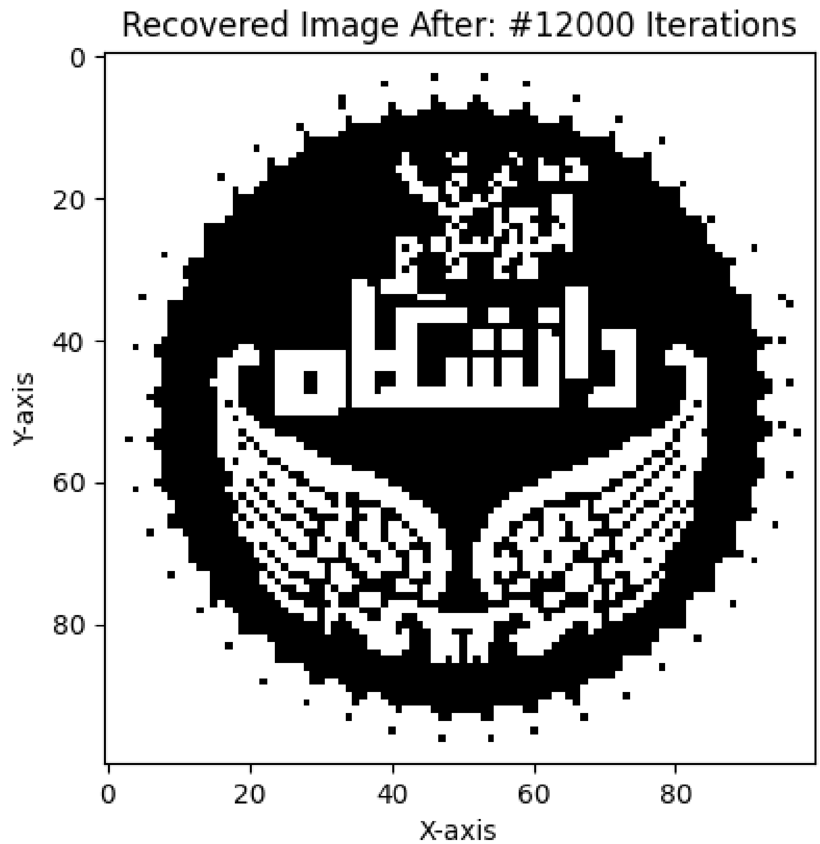
```
    # - Random Update: The code randomly selects a single neuron (i) from the␣
↪image (y)
    # - It calculates the local field (y_in) at the chosen neuron.
    #  This represents the weighted sum of all connected neurons' states in the␣
↪network.
    #  The formula used is:
    y_in = x[0][i] + np.matmul(y[0,:],w[:,i].reshape(10000,1))

    # Based on the local field value (y_in),
    # the state of the selected neuron is updated using a threshold (theta set␣
↪to 0.5 in this case)
    if(y_in>theta):
        y[0,i] = 1
    if(y_in<theta):
        y[0,i] = 0
    if iter % 3000 == 0: # Visualize the Result every 1k Iter:
        t = y.reshape(100, 100)
        t_normalized= np.clip(t,0,1)
        plt.imshow(np.repeat(t_normalized[:,:,np.newaxis], repeats=3, axis=2))
        plt.title(f'Recovered Image After: #{iter} Iterations')
        plt.xlabel('X-axis')
        plt.ylabel('Y-axis')
        plt.show()
```


Recovered Image After: #0 Iterations
Y-axis
X-axis
0
20
40
60
80
0
20
40
60
80


Recovered Image After: #3000 Iterations
Y-axis
X-axis
0
20
40
60
80
0
20
40
60
80


Recovered Image After: #6000 Iterations
Y-axis
X-axis
0
20
40
60
80

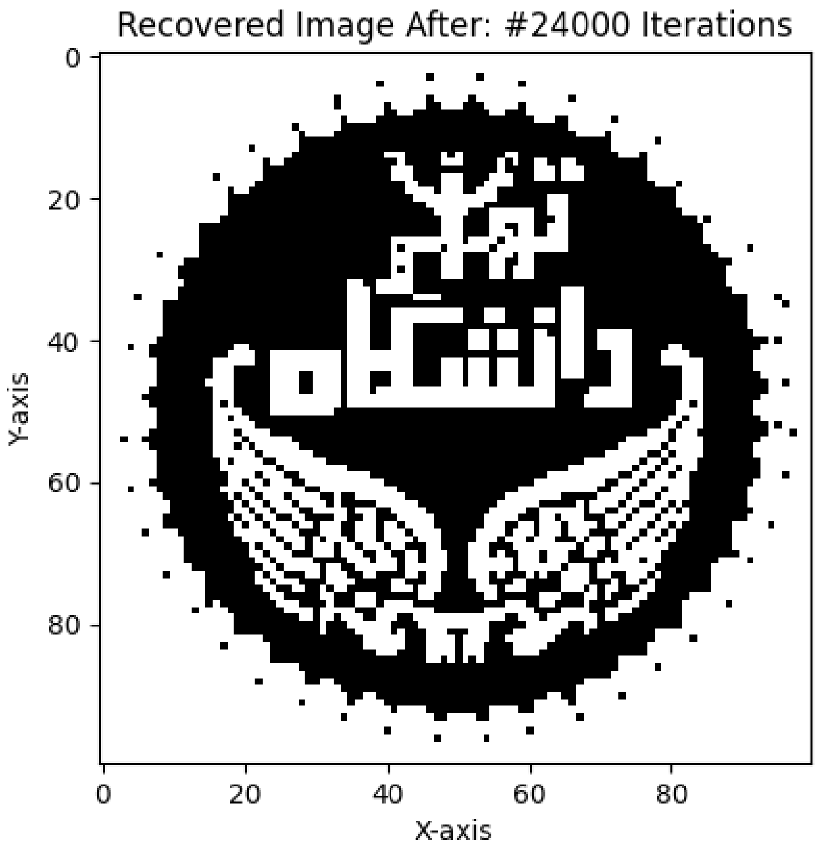

Recovered Image After: #9000 Iterations
Y-axis
X-axis
0
20
40
60
80
0
20
40
60
80


Recovered Image After: #12000 Iterations
Y-axis
X-axis
0
20
40
60
80
0
20
40
60
80

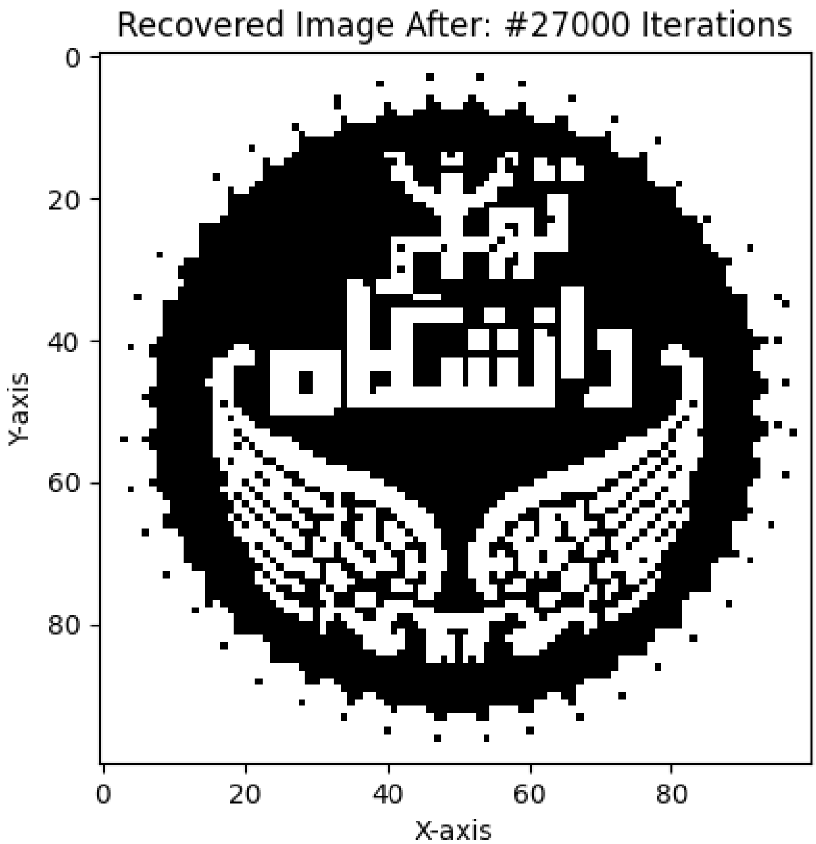
Recovered Image After: #15000 Iterations
Y-axis
X-axis
0
20
40
60
80


Recovered Image After: #18000 Iterations
Y-axis
X-axis
0
20
40
60
80
0
20
40
60
80

Recovered Image After: #21000 Iterations
Y-axis
X-axis
0
20
40
60
80



Recovered Image After: #24000 Iterations
Y-axis
X-axis
0
20
40
60
80


Recovered Image After: #27000 Iterations
0
20
40
60
80
Y-axis
0
20
40
60
80
X-axis


Recovered Image After: #30000 Iterations
Y-axis
0
20
40
60
80
X-axis
0
20
40
60
80