Neuroscience

Hopfield Network Simulation

Prajwal Singh
2020A7PS0192H

Birla Institute of Information and Technology, Pilani

Context

1. Introduction
2. Hopfield Logic
3. Methodology
    a. Dataset Preparation
    b. Reading and Writing PBM Files
    c. Corruption methodology
        i. Flipping data
        ii. Cropping data
    d. Convergence
    e. Task 1
    f. Task 2
    g. Task 3
4. Appendix
5. References

1. Introduction

The objective of this report is to document the implementation and experimentation of a Hopfield Network, a type of recurrent neural network, as part of a programming assignment. In this assignment, we set out to create a Hopfield Network from scratch, train it with a dataset of 25 60x60 pixel black and white images in the PBM (Portable BitMap) format, and conduct various experiments to analyse its behaviour.

Hopfield Networks are a form of associative memory networks that have proven to be powerful tools for pattern recognition and retrieval tasks. Our goal was to implement this network and investigate its performance under different conditions, including corrupted memories and various update methods.

Throughout our experiments, we estimated several key quantities and curves:

Convergence Rate: We tracked the fraction of initializations that converged to the correct uncorrupted memory for each corruption level 'p'. This allowed us to assess the network's robustness to different levels of memory corruption.

Update Steps Histogram: For experiments where initialization converged to the correct uncorrupted memory, we created histograms to visualize the distribution of the number of update steps required for convergence. This analysis provided insights into the convergence dynamics.

2. Hopfield Logic

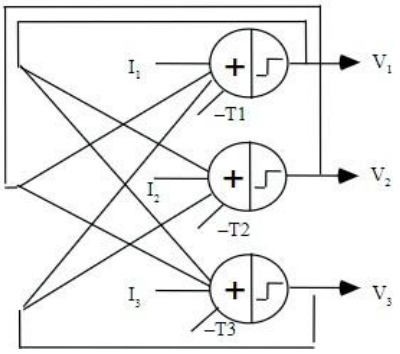
Hopfield network works as Recurrent neural network to collect and retrieve memory in a manner similar to that of the Human Brain. Here, a neuron is either on or off the situation. The state of a neuron(on +1 or off 0) will be restored, relying on the input it receives from the other neuron. A Hopfield network is at first prepared to store various patterns or memories. Afterward, it is ready to recognize any of the learned patterns by uncovering partial or even some corrupted data about that pattern, i.e., it eventually settles down and restores the closest pattern. Thus, similar to the human brain, the Hopfield model has stability in pattern recognition.

Function:

Net function $u_i = \sum_{i=j} w_{ij} v_j + I_i$

$I_i = u_i / R_i$

Output: $v_i = \begin{cases} 1 & u_i \geq 0 \\ -1 & u_i < 0. \end{cases}$



Wij: Weight Matrix of trained Hopfield network

Vj: Flattened input of target image.

U: Threshold value, taken as 0 in the above figure

Synchronously:

In this approach, the update of all the nodes taking place simultaneously at each time.

Asynchronously:

In this approach, at each point of time, update one node chosen randomly or according to some rule. Asynchronous updating is more biologically realistic.

3. Methodology

a. Dataset Preparation

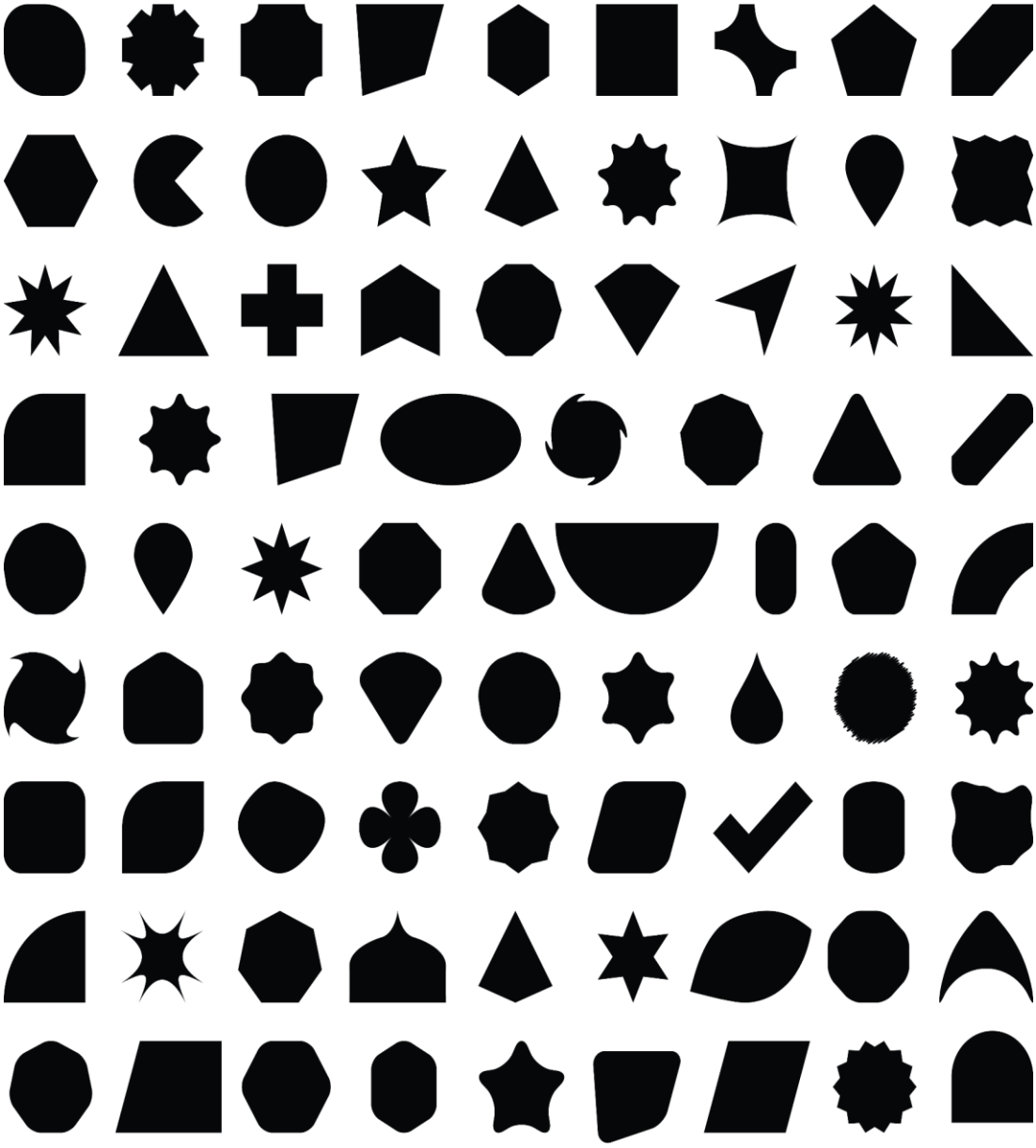
We initially collected similar-looking images, which, in our case, we took of different shapes. The data collected was in the form of ".PNG" files, which is a commonly available format.

We further converted them into the required format of 60x60 and ".PBM" format using a Python script.

```
from pathlib import Path
from PIL import Image
ASCII_BITS = '0', '1'
input_directory = Path('./pics')
output_directory = Path('./workpics')
output_directory.mkdir(exist_ok=True)
target_size = (60, 60)

for imagepath in input_directory.glob('*.png'):
    img = Image.open(imagepath).convert('1')
    img = img.resize(target_size, Image.ANTIALIAS)
    width, height = img.size
    data = [ASCII_BITS[bool(val)] for val in img.getdata()]
    data = [data[offset: offset+width] for offset in range(0, width*height,
width)]
    output_pbm_path = output_directory / imagepath.with_suffix('.pbm').name

    with open(output_pbm_path, 'w') as file:
        file.write('P1\n')
        file.write(f'# Conversion of {imagepath} to PBM format\n')
        file.write(f'{width} {height}\n')
        for row in data:
            file.write(' '.join(row) + '\n')
    print(f'Conversion of {imagepath} to PBM completed.')

print('All conversions finished.')
```

The above codes reads the data from the "pics" folder, which contains all the png files, and then resizes them to 60x60 pixels and further converts them into ".PBM" format.

Dataset images



b. Reading and Writing PBM files.

The following function, “load_training_data”, is used to read all the PBM images and store them after flattening them into a 1D array from a 2D array.

Finally, “training_data” is a 2D array, which has dimension of 25x3600 and each row stores flattened data of the image which is 60x60

```
def load_training_data(folder_path):
    training_data = []
    for pbm_file in os.listdir(folder_path):
        if pbm_file.endswith(".pbm"):
            file_path = os.path.join(folder_path, pbm_file)
            with Image.open(file_path) as img:
                img = img.resize((60, 60), Image.LANCZOS)
                img = img.convert('L')
                binary_vector = np.array(img).flatten()
                binary_vector[binary_vector < 128] = 0
                binary_vector[binary_vector >= 128] = 1
                training_data.append(binary_vector)
    return training_data
```

c. Corruption Methodology

We are given two corruption methodoly which are as follows:

I. Flipping data with probability.
II. Cropping data.

Flipping data with probability:

In this method, we randomly choose one of our images from the dataset and flip the digits (since the file is in PBM format, the data is constrained to either 1 or 0), i.e, 1 to 0 and vice versa with a probability, which for the task 2 was taken as 0.3.

```
def corrupt_initial_state(initial_state, corruption_prob):
    corrupted_state = initial_state.copy()
    for i in range(len(initial_state)):
        if random.random() < corruption_prob:
            corrupted_state[i] = 1 - corrupted_state[i]
    return corrupted_state
```

Cropping data

In this method, we start off by selecting an image randomly like in the case of Flipping data with probability, but this time we crop the data from 60x60 to 40x40 and fill the cropped out area with either 1 or 0.

```
def crop_memory(memory, image_size, box_size, fill_value=0):
    memory = memory.reshape(image_size)
    cropped_memory = np.full(image_size, fill_value)
    start_index = (image_size[0] - box_size[0]) // 2
    end_index = start_index + box_size[0]
    cropped_memory[start_index:end_index, start_index:end_index] = \
        memory[start_index:end_index, start_index:end_index]
    return cropped_memory.flatten()
```

D. Convergence

In the context of our Hopfield Network experiments, "convergence" represents a pivotal concept in understanding the network's ability to retrieve stored memories from corrupted or partially altered inputs. Convergence refers to the state when the network's dynamics stabilise, and the output settles into a pattern that closely matches one of the stored memories.

As we systematically varied the corruption levels ('p') of the memories during initialisation, observing convergence allowed us to gauge the network's resilience and pattern recognition capabilities. A high convergence rate at lower corruption levels indicates the network's robustness in recognising and correcting minor distortions in input patterns. On the other hand, lower convergence rates at higher corruption levels shed light on the network's limitations when dealing with severely corrupted or incomplete data.

Through these convergence experiments, we aim to decipher the intricacies of Hopfield Networks in pattern recognition and retrieval, offering valuable insights into their real-world applications, such as error correction and associative memory tasks.

E. Task 1

Task 1, we took our first steps in preparing our Hopfield Network for action. This phase involved two crucial elements: memory preparation and network training.

Firstly, we carefully curated a dataset comprising 25 images, each measuring 60x60 pixels, and converted them into the PBM format. These images, whether sourced from the web or created by us, were resized uniformly and represented in a binary fashion. Essentially, they served as memories that our Hopfield Network would rely on.

With our memory bank established, we proceeded to implement the Hebbian Learning rule, a classical method for imprinting memories into the network. This rule adjusted the weights of connections between neurons based on the interconnections observed within the stored memories. Consequently, our network became adept at storing and subsequently recalling these patterns.

```
def initialize_weight_matrix(neurons_num, training_data):
    # Initialize the weight matrix with zeros
    weight_matrix = np.zeros((neurons_num, neurons_num))

    # Initialize the weight matrix using Hebbian Learning
    for pattern in training_data:
        pattern = pattern.reshape((-1, 1))
        weight_matrix = weight_matrix + np.dot(pattern, pattern.T)

    weight_matrix /= len(training_data)

    return weight_matrix
```

Here, The training data is the output of the code snippet provided in the “Reading and Writing PBM files” section, and the following function “initialize_weight_matrix” implements the hebbian learning rule with a total of 3600 neurons, resulting in the final weight matrix, namely “weight_matrix”.

F. Task 2

In Task 2, we delved into the intricacies of manipulating our stored memories and guiding our network through updates.

To begin, we explored two methods of memory corruption. First, we introduced probabilistic pixel flipping, altering the state of each pixel with a specified probability 'p'. Second, we implemented cropping techniques, preserving a portion of the original memory within a defined bounding box while modifying the surrounding pixels. These techniques simulated real-world scenarios where memories can be partially distorted.

The code snippet for both things is provided in both the index and the "Corruption Methodology" section.

Following memory corruption, we navigated network updates. We employed both the synchronous and asynchronous updates on the two corruption methodologies to analyse the difference.

Employing synchronous updates, we synchronized all neurons, allowing them to adjust their states simultaneously. In contrast, asynchronous updates involved sequential neuron adjustments in a random order. This dual approach enabled us to investigate how our network coped with simultaneous versus sequential information processing.

I. Snippet 1 - Synchronous Update for "Flipped with probability p" data.

```
def sync_simulation(training_data, num_iterations, output_folder, p, save_interval=1):
    neurons_num = training_data[0].shape[0]
    weight_matrix = initialize_weight_matrix(neurons_num, training_data)
    initial_state = random.choice(training_data).copy()
    initial_state = corrupt_initial_state(initial_state, p)

    for i in range(num_iterations):
        activations = np.dot(weight_matrix, initial_state)
        new_state = (activations >= 0.5).astype(int)
        if i % 1 == 0:
            output_path = os.path.join(output_folder, 'sync_states', f'state_{i}.pbm')
            save_network_state_as_pbm(initial_state, output_path, (60, 60))
        initial_state = new_state

    print("Synchronous Hopfield Network updates completed.")
```

II. Snippet 2 - Synchronous Update for “cropped” data

```
def synchronous_simulation_cropped(training_data, num_iterations, output_folder):
    neurons_num = training_data[0].shape[0]
    weight_matrix = initialize_weight_matrix(neurons_num, training_data)

    initial_state = random.choice(training_data).copy()
    initial_state = crop_memory(initial_state, (60, 60), (40, 40), fill_value=0)

    for i in range(num_iterations):
        neuron_index = random.randint(0, neurons_num - 1)
        activation = np.dot(weight_matrix[neuron_index, :], initial_state)
        new_state = 1 if activation >= 0.5 else 0
        initial_state[neuron_index] = new_state

        if i % 500 == 0:
            output_path = os.path.join(output_folder, 'sync_cropped', f'state_{i}.pbm')
            save_network_state_as_pbm(initial_state, output_path, (60, 60))
    print("Synchronous Hopfield Network updates with cropping completed.")
```

III. Snippet 3 - Asynchronous Update for “Flipped with probability p” data.\

```
def async_simulation(training_data, num_iterations, output_folder, p):
    neurons_num = training_data[0].shape[0]  #3600

    weight_matrix = initialize_weight_matrix(neurons_num, training_data)
    initial_state = random.choice(training_data).copy()
    initial_state = corrupt_initial_state(initial_state, p)

    for i in range(num_iterations):
        neuron_index = random.randint(0, neurons_num - 1)
        activation = np.dot(weight_matrix[neuron_index, :], initial_state)
        new_state = 1 if activation >= 0.5 else 0

        if i % 500 == 0:
            output_path = os.path.join(output_folder, 'async_states',f'state_{i}.pbm')
            save_network_state_as_pbm(initial_state, output_path, (60, 60))

        initial_state[neuron_index] = new_state

    print("Hopfield Network updates completed ASYNCRONOUSLY.")
```

IV. Snippet 4 - Asynchronous Update for “Cropped” data.

```
def asynchronous_simulation_cropped(training_data, num_iterations, output_folder):
    neurons_num = training_data[0].shape[0]
    weight_matrix = initialize_weight_matrix(neurons_num, training_data)

    initial_state = random.choice(training_data).copy()
    initial_state = crop_memory(initial_state, (60, 60), (40, 40), fill_value=0)

    for i in range(num_iterations):
        neuron_index = random.randint(0, neurons_num - 1)
        activation = np.dot(weight_matrix[neuron_index, :], initial_state)
        new_state = 1 if activation >= 0.5 else 0

        if i % 500 == 0:
            output_path = os.path.join(output_folder, "async_cropped",f'state_{i}.pbm')
            save_network_state_as_pbm(initial_state, output_path, (60, 60))

        initial_state[neuron_index] = new_state

    print("Asynchronous Hopfield Network updates with cropping completed.")
```

G. Task 3

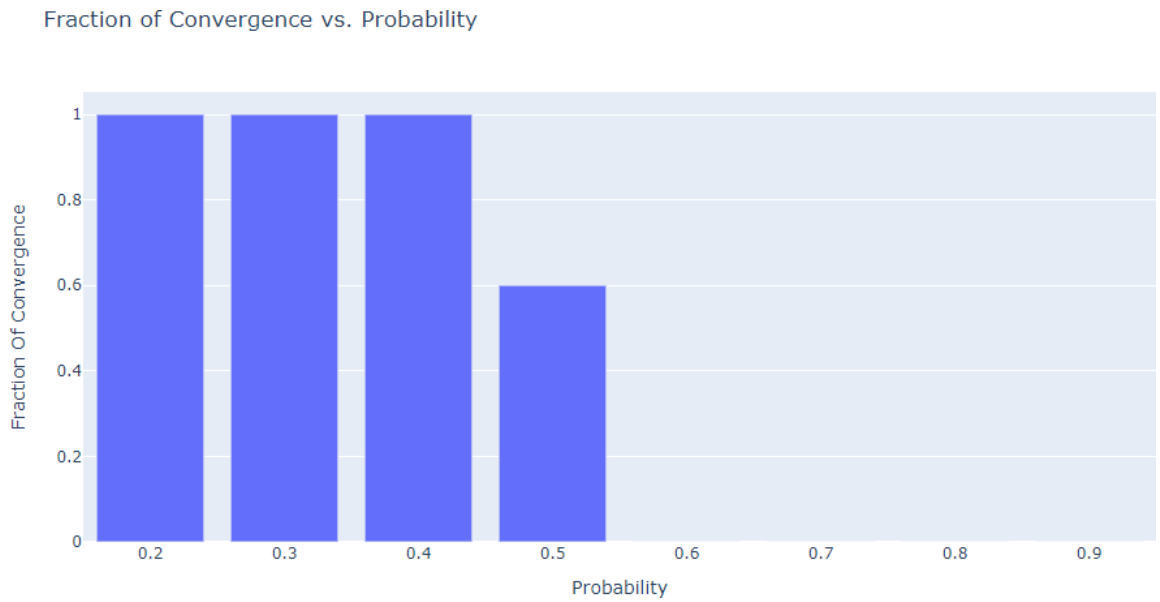
This task entails conducting 20 iterations with varying initializations for a set of probability values while monitoring convergence using synchronous updates. We record the frequency at which the network successfully converges to the original pattern and note the number of iterations needed for convergence. In this context, we assume a learning rate (α) of 0.000001. The obtained results are as follows:

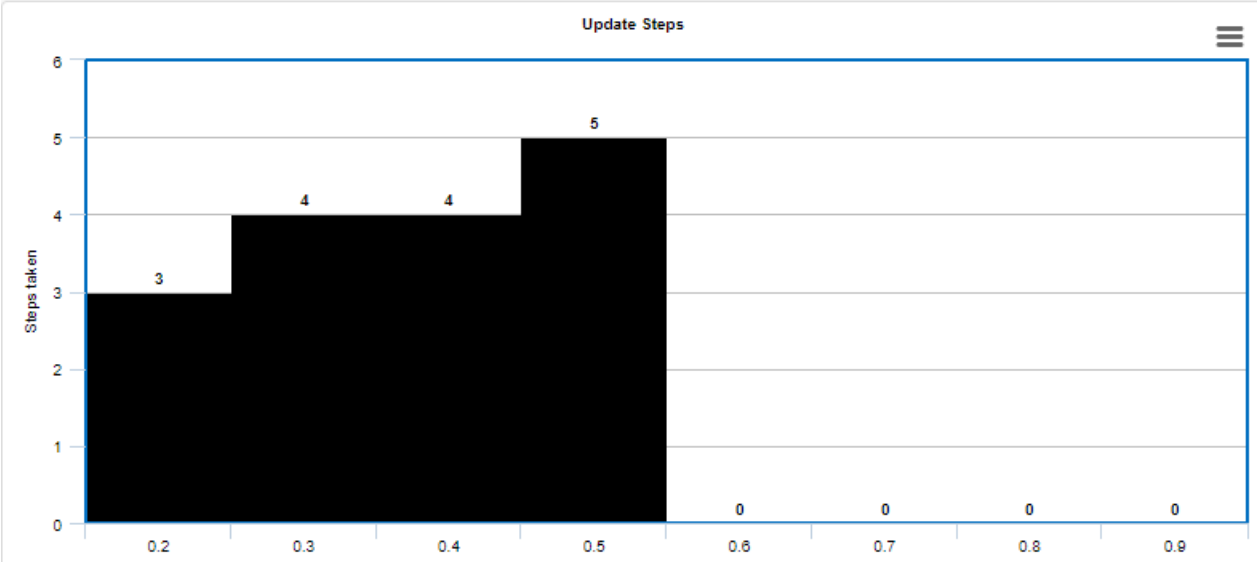
Probability	Fraction_Of_Convergecne
0.2	1.0
0.3	1.0
0.4	1.0
0.5	0.6
0.6	0.6
0.7	0.0
0.8	0.0
0.9	0.0

Hare, The probability refers to the flipping probability that each pixel has of changing it's value from 0 to 1 or from 1 to 0.

It can be observed that the convergence is quite accurate for the probabilities of 0.2, 0.3 and 0.4 but starts to break at higher possibilities. This could be due to the fact that the model is not trained on a big enough dataset.

Below are some more graphs of the data.

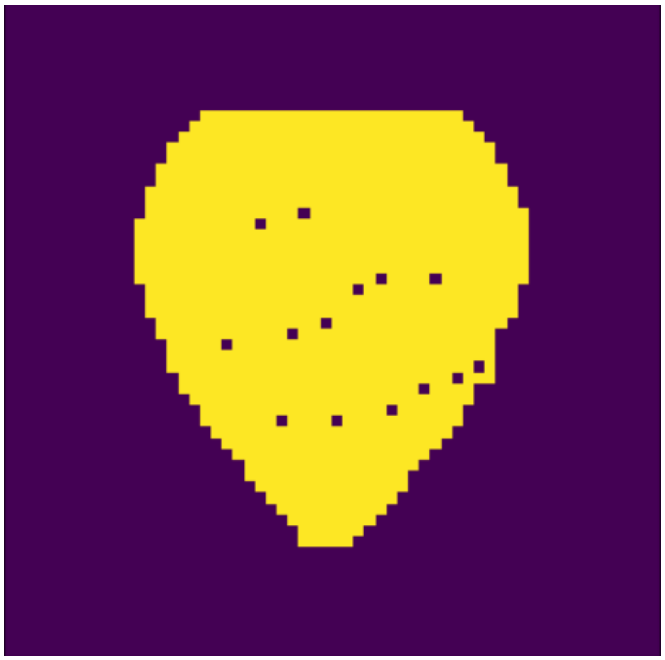




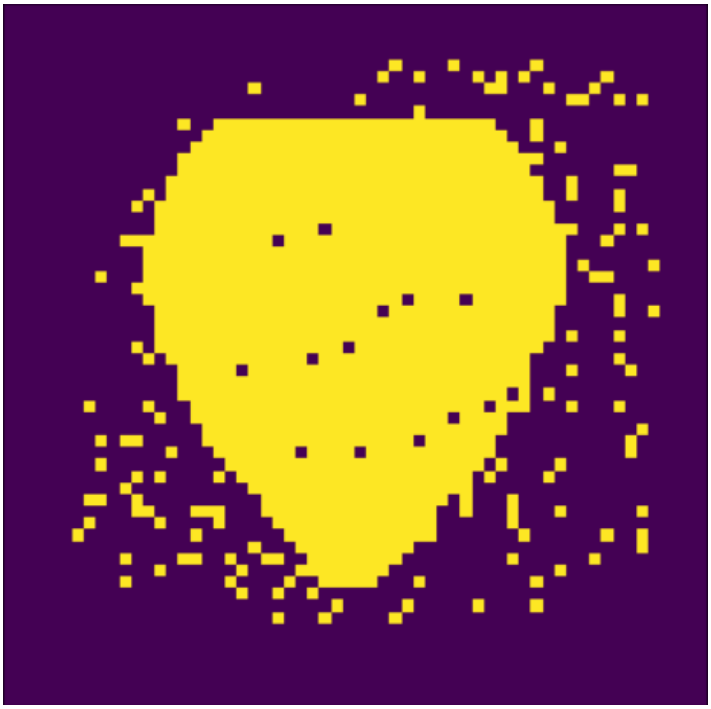
4. Examples – Code Demo

a. Asynchronous Updates

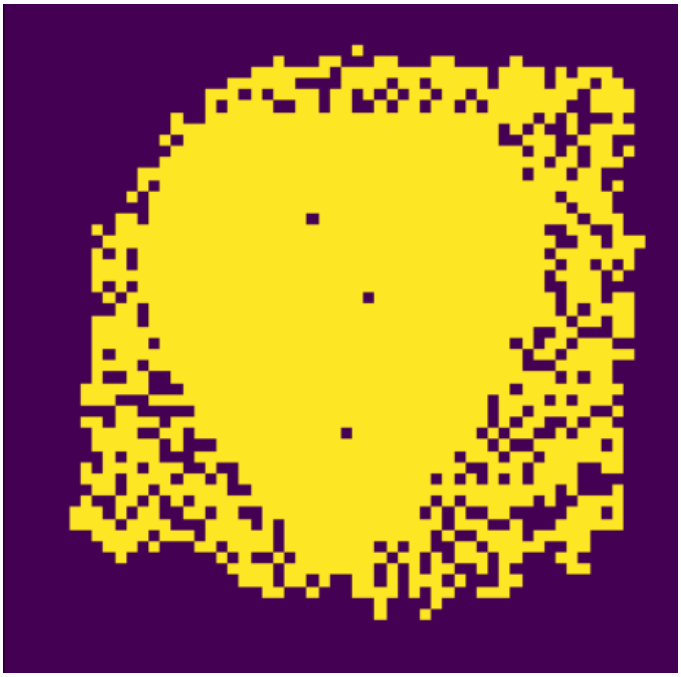
Here, we have examples of our code working for Asynchronous updates



- We Start off with the image which Is cropped to 55% of it's original size, along with flipping with a probability of 0.6 in the edge regions.



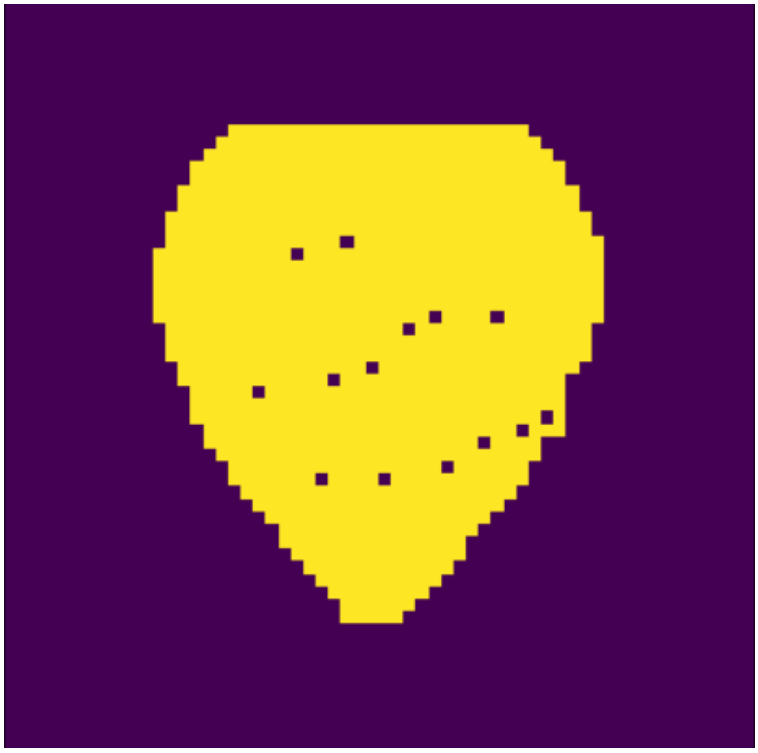
On iteration number 50, This image is observe, which can be seen converging to some predefined state.



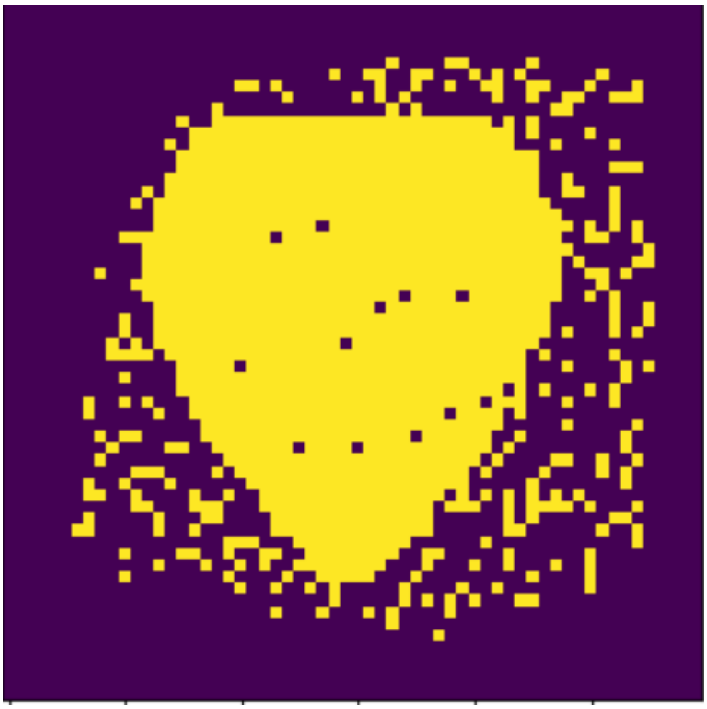
On iteration number 150, The above state of the image is observed, where the predefined state is becoming much more apparent, and on subsequent increase in iterations, very less change is observed and the image finally settles at this state, making it more efficient to consider this the final state, which is somewhat accurate.

b. Synchronous Updates

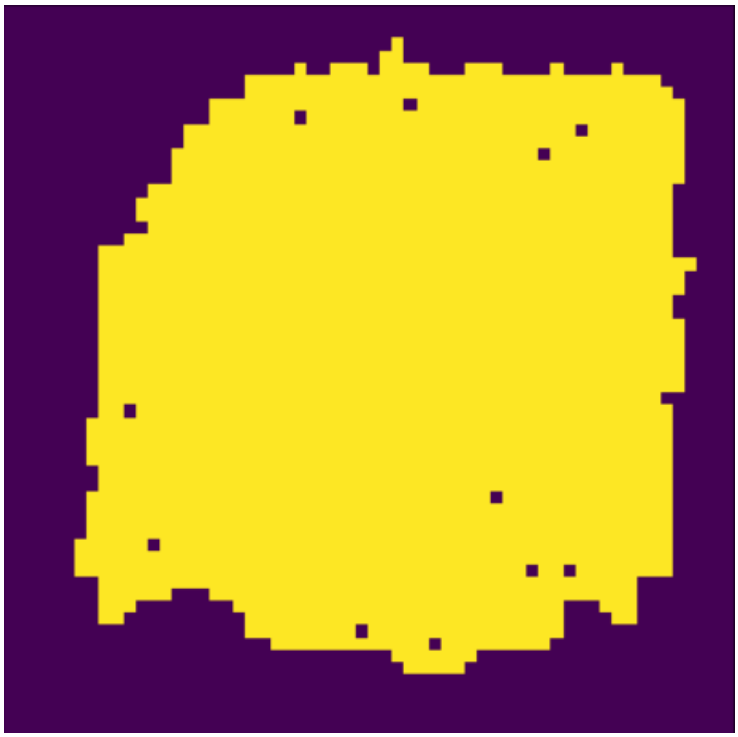
Here we have examples of the same figure but the updates are being done in a synchronous manner, making us expect that the final state would we reached in a much faster.



We are once again starting off with the same image and same corruption methodology, where the image is reduced to 55% of it's original size, along with flipping with a probability of 0.6 in the edge regions.



On just the 5th iteration, we observe that the image is starting to converge to a predefine trained state and in a much faster manner. To contrast, Asynchronous method took around 100 iterations to reach a state which is similar the above state.



This is the state of the image at iteration 10. We observe that the image has completed the convergence and increase in iterations doesn't give any more observable changes to the same.

Comparing, synchronous updates with that of asynchronous updates we observe that synchronous updates are much faster and also reach closer to the final state.

- This goes on to highlight that using synchronous method gives the results in a much more efficient manner.

5. Appendix

Code for file conversion:

```
from pathlib import Path
from PIL import Image

ASCII_BITS = '0', '1'

# Specify the directory containing the PNG images (input) and the output
directory for PBM files.
input_directory = Path('./pics')  # Assuming "pics" is the folder
containing PNGs.
output_directory = Path('./workpics')  # Specify the output directory.

# Create the output directory if it doesn't exist.
output_directory.mkdir(exist_ok=True)

# Specify the target size for resizing.
target_size = (60, 60)

# Iterate through all PNG files in the input directory.
for imagepath in input_directory.glob('*.png'):
```

```
    # Open and resize the image to the target size.
    img = Image.open(imagepath).convert('1')  # Convert image to bitmap.
    img = img.resize(target_size, Image.ANTIALIAS)  # Resize to 60x60
pixels.
    width, height = img.size

    # Convert image data to a list of ASCII bits.
    data = [ASCII_BITS[bool(val)] for val in img.getdata()]
    # Convert that to a 2D list (list of character lists)
    data = [data[offset: offset+width] for offset in range(0, width*height,
width)]

    # Create an output PBM file path in the output directory.
    output_pbm_path = output_directory / imagepath.with_suffix('.pbm').name

    with open(output_pbm_path, 'w') as file:
        file.write('P1\n')
        file.write(f'# Conversion of {imagepath} to PBM format\n')
        file.write(f'{width} {height}\n')
        for row in data:
            file.write(' '.join(row) + '\n')

    print(f'Conversion of {imagepath} to PBM completed.')

print('All conversions finished.')
```

Code for Hopfield Simulation

```
import numpy as np
import pandas as pd
from PIL import Image
import os
import random
import matplotlib.pyplot as plt
import plotly.express as px

def crop_memory(memory, image_size, box_size, fill_value=0):
    memory = memory.reshape(image_size)
    cropped_memory = np.full(image_size, fill_value)
    start_index = (image_size[0] - box_size[0]) // 2
```

```
    end_index = start_index + box_size[0]
    cropped_memory[start_index:end_index, start_index:end_index] = \
        memory[start_index:end_index, start_index:end_index]
    return cropped_memory.flatten()


def initialize_weight_matrix(neurons_num, training_data):
    # Initialize the weight matrix with zeros
    weight_matrix = np.zeros((neurons_num, neurons_num))

    # Initialize the weight matrix using Hebbian Learning
    for pattern in training_data:
        pattern = pattern.reshape((-1, 1))
        weight_matrix = weight_matrix + np.dot(pattern, pattern.T)

    weight_matrix /= len(training_data)
    return weight_matrix


def load_training_data(folder_path):
    # This is an empty list which we will use to store the training
patterns (PBM images as binary vectors)
    training_data = []

    # Iterating over all the files in our workpics folder
    for pbm_file in os.listdir(folder_path):
        if pbm_file.endswith(".pbm"):
            # Check if the file extension is correct
            file_path = os.path.join(folder_path, pbm_file)

            # Opening and adding it to the training dataset list
            with Image.open(file_path) as img:
                img = img.resize((60, 60), Image.LANCZOS)
                img = img.convert('L')

                binary_vector = np.array(img).flatten()

                # To make the image into "light" and "dark" pattern.
(Threshold = 128)
                binary_vector[binary_vector < 128] = 0
                binary_vector[binary_vector >= 128] = 1
```

```
                # Finally append the vector to the training data set.
                training_data.append(binary_vector)
    return training_data




def save_network_state_as_pbm(state, file_path, image_size):
    binary_state = state.astype(np.uint8) * 255
    binary_state = binary_state.reshape(image_size)
    img = Image.fromarray(binary_state, mode='L')
    img.save(file_path)


def corrupt_initial_state(initial_state, corruption_prob):
    corrupted_state = initial_state.copy()
    for i in range(len(initial_state)):
        if random.random() < corruption_prob:
            corrupted_state[i] = 1 - corrupted_state[i]
    return corrupted_state


def threshold_function(activation):
    # Apply a step function to threshold the activation
    return 1 if activation >= 0 else 0



def sync_simulation(training_data, num_iterations, output_folder, p,
save_interval=1):
    neurons_num = training_data[0].shape[0]

    # Initialize weight matrix (you need to implement this)
    weight_matrix = initialize_weight_matrix(neurons_num, training_data)

    # Choose a random initial state
    initial_state = random.choice(training_data).copy()

    # Corrupt the initial state with probability p (you need to implement
```

```
this)
    initial_state = corrupt_initial_state(initial_state, p)

    # Perform Hopfield Network updates and save states midway
    for i in range(num_iterations):
        activations = np.dot(weight_matrix, initial_state)

        # Apply your threshold function here
        new_state = np.where(activations >= 0.5, 1, 0)

        # Save the current state as a PBM image at specified intervals
        if i % 1 == 0:
            output_path = os.path.join(output_folder, 'sync_states',
f'state_{i}.pbm')
            save_network_state_as_pbm(initial_state, output_path, (60, 60))

        # Update the state for the next iteration
        initial_state = new_state

    print("Synchronous Hopfield Network updates completed.")



def synchronous_simulation_cropped(training_data, num_iterations,
output_folder):
    neurons_num = training_data[0].shape[0]

    # Initialize the weight matrix
    weight_matrix = initialize_weight_matrix(neurons_num, training_data)

    # Create an initial state for the Hopfield Network
    initial_state = random.choice(training_data).copy()

    initial_state = crop_memory(initial_state, (60, 60), (40, 40),
fill_value=0)

    # Perform synchronous Hopfield Network updates with cropping and save
states midway
    for i in range(num_iterations):
        neuron_index = random.randint(0, neurons_num - 1)
        activation = np.dot(weight_matrix[neuron_index, :], initial_state)
```

```
        new_state = 1 if activation >= 0.5 else 0

        # Update the selected neuron's state
        initial_state[neuron_index] = new_state

        # Save the current state as a PBM image midway
        if i % 500 == 0:
            output_path = os.path.join(output_folder, 'sync_cropped',
f'state_{i}.pbm')
            save_network_state_as_pbm(initial_state, output_path, (60, 60))

    print("Synchronous Hopfield Network updates with cropping completed.")



def async_simulation(training_data, num_iterations, output_folder, p):
    neurons_num = training_data[0].shape[0]  #3600

    weight_matrix = initialize_weight_matrix(neurons_num, training_data)
    initial_state = random.choice(training_data).copy()

    ##corrupt
    initial_state = corrupt_initial_state(initial_state, p)

    for i in range(num_iterations):
        neuron_index = random.randint(0, neurons_num - 1)
        activation = np.dot(weight_matrix[neuron_index, :], initial_state)
        new_state = 1 if activation >= 0.5 else 0

        if i % 500 == 0:
            output_path = os.path.join(output_folder,
'async_states',f'state_{i}.pbm')
            save_network_state_as_pbm(initial_state, output_path, (60, 60))

        initial_state[neuron_index] = new_state


    print("Hopfield Network updates completed ASYNCRONOUSLY.")
```

```
def asynchronous_simulation_cropped(training_data, num_iterations,
output_folder):
    neurons_num = training_data[0].shape[0]

    # Initialize the weight matrix
    weight_matrix = initialize_weight_matrix(neurons_num, training_data)

    initial_state = random.choice(training_data).copy()

    # Create an initial state for the Hopfield Network
    initial_state = crop_memory(initial_state, (60, 60), (40, 40),
fill_value=0)

    # Perform asynchronous Hopfield Network updates with cropping and save
states midway
    for i in range(num_iterations):
        neuron_index = random.randint(0, neurons_num - 1)
        activation = np.dot(weight_matrix[neuron_index, :], initial_state)
        new_state = 1 if activation >= 0.5 else 0

        # Save the current state as a PBM image midway
        if i % 500 == 0:
            output_path = os.path.join(output_folder,
"async_cropped",f'state_{i}.pbm')
            save_network_state_as_pbm(initial_state, output_path, (60, 60))

        # Update the selected neuron's state asynchronously
        initial_state[neuron_index] = new_state

    print("Asynchronous Hopfield Network updates with cropping completed.")

# Path to the PBM pictures
folderPath = "workpics"

# Number of iterations for updating the network
numIterations = 100

#Corruption Prob
p = 0.2

# Create an output folder for saving the states
```

```
outputFolder = "network_states"
os.makedirs(outputFolder, exist_ok=True)
os.makedirs(os.path.join(outputFolder, 'async_states'), exist_ok=True)
os.makedirs(os.path.join(outputFolder, 'sync_states'), exist_ok=True)
os.makedirs(os.path.join(outputFolder, 'sync_cropped'), exist_ok=True)
os.makedirs(os.path.join(outputFolder, 'async_cropped'), exist_ok=True)

trainingData = load_training_data(folderPath)

sync_simulation(trainingData, numIterations, outputFolder, p)
synchronous_simulation_cropped(trainingData, numIterations, outputFolder)
asynchronous_simulation_cropped(trainingData, numIterations, outputFolder)

async_simulation(trainingData, numIterations, outputFolder, p)

def display_image(i):
    path = f"./network_states/async_states/state_{i}.pbm"
    # Open the image file
    img = Image.open(path)

    # Display the image
    plt.imshow(img)
    plt.show()

display_image(0)
```

6. References

1. Java Point || Hopfield network
2. Geeksforgeeks || Hopfield Neural Network
3. Philadelphia University || Hopefield Networks