# Introduction

This project implements the binary Hopfield model to store and retrieve patterns representing letters (A, C, K, T, W) using a 10 × 10 grid. The Hopfield model is a recurrent neural network characterized by its binary state, designed to work with pattern recognition and associative memory. This report details the implementation of the model, the steps taken to add noise to patterns, and the convergence of the network using asynchronous updates until stability. The main goal of this project is to evaluate the network's ability to recognize noisy patterns and converge to the original states. *The code is also heavily commented on and contains information about the process.*

## Code and libraries used

The code is written in python3. The libraries; numpy, matplotlib and random are needed to run the code. It first shows the desired letter grids(memories) to be retrieved. Then all five letters’ epochs are shown with three different sigma values. After each plot is shown, it needs to be closed to show the next plot. So there are 15 windows to be closed to arrive at the last plot window.

## Step 1: Data generation

We first define five binary patterns corresponding to the letters A, C, K, T, and W. These patterns are represented on a 10×10 grid, where light pixels are denoted by 1 and dark pixels by −1. Each letter is converted into a vector form by flattening the 10×10 grid into a 100 element vector: . The grids are constructed in the code and visualized using matplotlib, as shown in the figure below.A black and white symbol

Description automatically generated

## Step 2: Weight Matrix Calculation

## The next step is to calculate the weight matrix that stores each pattern. For each letter, the weight matrix is computed using the Hebbian learning rule: , where is the outer product of the pattern vector with itself, yielding a matrix where each element is . That is, each connection between two neurons i and j are only a product of the two’s current values. The diagonal is subtracted to avoid getting self-loops during runs. This step generates a weight matrix for each letter that will be used in the update process.

## Step 3: Noise Addition

We introduce noise to the patterns by adding zero-mean Gaussian noise with different standard deviations . The noisy pattern is generated as follows:

The noisy vector is then binarized using the np.sign function: . This process ensures that the distorted pattern remains binary, i.e., each element of the noisy vector are either -1 or 1.

## Step 4: Asynchronous Update of Hopfield Network

In this step, the Hopfield network updates asynchronously, meaning that at each iteration, only one neuron is updated at a time. A neuron is randomly selected from the 100 neurons, and its state is updated according to the following rule:

Where is the state of the i’th neuron at time t, is the weight between the neurons i and j. The sign function ensures that the updated state of the neuron is either -1 or 1. This can be understood as if the desired states of a pair of neurons are the same, the weight between them will be positive signed, this means that these two neurons “agree”. The resulting positive weight (or "agreement" weight) is multiplied by the current state of the neuron j, contributing positively to the sum of neuron i and reinforcing into it the current state of the neuron j. Conversely, if the states of the neurons disagree, the weight will be negative, introducing a negative contribution to the sum. The total input to the neuron is the combined effect of all these "agreements" and "disagreements" from other neurons, determining whether the neuron's state remains the same or flips during the update.

The algorithm works by randomly selecting a neuron 100 times per epoch, but it does not guarantee that each neuron will be updated during every epoch. This randomness introduces variability in the convergence process, as some neurons may be updated multiple times while others may not be updated at all.

## Step 5: Results and Convergence

We run the Hopfield network for each letter with three different noise levels and observe the convergence process. Epoch 0 shows the noise added pattern. Each epoch after that are plotted after 100 random pixels are updated. If two subsequent epochs are identical, a false convergence is said to be achieved. But the code specifies the algorithm to run until *true convergence,* that is, the achieved pattern is the same as the stored pattern. Next, all the 15 runs will be shown and comments will be added as necessary to describe behavior.

A black and white image of a person

Description automatically generated

A black and white image of a number

Description automatically generated

A black and white image of a letter

Description automatically generated

A black and white logo

Description automatically generated

A black and white logo

Description automatically generated

A black and white image of a square

Description automatically generated

Upside is a letter C run with sigma set to 1.1. It can be seen that the epochs 3, 4 and 5 are all identical and differ by one pixel from the stored pattern. This is due to the process of selecting the neurons to be updated, it is a random process. If during these epochs the wrong pixel was randomly selected, the other, mostly correct pixels would reinforce the wrong one to flip into black. This happens in epoch 5 to finally achieve the stored C pattern.

A black and white logo

Description automatically generated

A black and white symbols

Description automatically generated

A black and white image of a letter k

Description automatically generated

A white letter on a black background

Description automatically generated

A black and white logo

Description automatically generated

A black and white image of a letter

Description automatically generated

A black and white image of a black and white image of a black and white image of a black and white image of a black and white image of a black and white image of a black and

Description automatically generated

A black and white image of a sign

Description automatically generated

A black and white image of a number

Description automatically generated

This sums up the specified 15 runs with 5 letters and 3 weight values. It is seen that the stored states are achieved in each run.

## Step 6: Special Cases and Some Observations

a.During one of the runs with sigma=1.1 an inverse image was observed. This happens due to the fact that the weights are symmetrical in the weight matrix, i.e., , so if the noise is high enough, and the first randomly selected neurons on the first epoch are not diverse enough, the model converges to the inverse of the stored pattern. This was demonstrated with setting sigma to an arbitrarily large value:

A group of black and white symbols

Description automatically generated

This can be fixed by storing the inverses of the stored patterns in another list, and checking if after the 100 epoch cap the achieved pattern is in this list. And if so simply inverting 1’s to -1’s and vice versa would retrieve the desired pattern.

b.