# 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. The letters had to be set to be big enough to touch both the side and vertical edges of the grid, so to prevent white letters blending into white background, the background was finally set to black, the reason is also mentioned in the last paragraph of step 6.

## 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 white pixelated letter k

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, which then are simply summed up to generate a single weight matrix that stores all the five figures’ patterns. This can be shown as follows

## 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 white and black screen with a black background

Description automatically generated

A screenshot of a video game

Description automatically generated

A white letter on a black background

Description automatically generated

All the letter A runs achieve the stored A letter from the single weight matrix effectively. Also the epochs are clearly visualized with the pixels correcting each epoch by using weights and values of other, majority, correct pixels. In the epochs 2 and 3 of the Sigma=1.1 case we have identical patterns. This is due to the nature of the update rule, it simply did not pick the missing pixels in first two epochs, the random process of picking pixels has the disadvantage of slowing down true convergence and showing false convergence. But is employed along with asynchronous updates to have a reliable but slower process, and is easier to see the changes along the runs.

A black and white screen with white text

Description automatically generated

A black and white pixelated image

Description automatically generated

A black and white screen with white text

Description automatically generated

A screenshot of a computer screen

Description automatically generated

A black and white pixelated image

Description automatically generated

A white text on a black background

Description automatically generated

Letter K with Sigma=1.1 can be seen to have stalled at two distinct, non-stored states.

A screen shot of a black and white letter

Description automatically generated

A black and white image of a square object

Description automatically generated

A white letter t on a black background

Description automatically generated

A screen shot of a computer screen

Description automatically generated

A screenshot of a video game

Description automatically generated

A screen shot of a video game

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. This was expected because the number of states stored are much less then the expected capacity of the Hopfield network:

## Step 6: Special Cases and Some Observations

During one of the runs with sigma=1.1 an inverse image was observed. This happens because 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 sometimes converges to the inverse of the stored pattern. This case was skipped over at first but later is demonstrated with setting sigma to an arbitrarily large value: A black and white screen shot

Description automatically generated

This is because the Hopfield network in this code can converge to either the original pattern or its inverse, as both are stable states of the network. This happens because the weight matrix calculation (create\_weight\_matrix) uses outer products, which means that flipping all signs in a pattern (-v instead of v) results in the same contribution to the weights: W += np.outer(pattern, pattern). When you multiply -v by -v, you get the same result as multiplying v by v. This means both the pattern and its inverse are stable states of the network. 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.

Another thing to mention is that during initial implementations, the network exhibited a persistent spurious attractor state that emerged regardless of the input pattern or noise level. This unwanted stable state appeared when using thinner letter representations in the training patterns. Interestingly, the issue was resolved by modifying the letter patterns to use thicker strokes that also use the edge pixels(**that is the reason why a black background was finally opted for**), suggesting that the original pattern geometries created an unintended strong local minimum in the network's energy landscape. This demonstrates how the choice of training pattern representations can significantly impact the network's behavior and stability, potentially creating artificial attractors that compete with the desired stored patterns. This phenomenon occurs because Hopfield networks can develop stable states that are not part of the training set but emerge from linear combinations or correlations between the stored patterns. When the letters were thinner, they likely had a particular spatial correlation structure that created a very strong spurious attractor. The thicker letters presumably reduced these unwanted correlations, leading to more robust pattern storage and retrieval. The spurious state looked like this: A black square with white squares

Description automatically generated with medium confidence

A black and white logo

Description automatically generated

The letters initially used that produced the spurious state. It can be observed that the spurious state is likely to be the “average” of the five shapes used.

## Conclusion

The project successfully implemented a binary Hopfield model, storing five distinct letter patterns in a single weight matrix. Using asynchronous updates, the model demonstrated reliable convergence to the correct patterns even under varying levels of Gaussian noise. Noise levels ranging from σ=0.5 to σ=1.1 were applied, and the network was able to recover the original patterns with high accuracy, despite occasional occurrences of inverse images or spurious states. The results confirmed the model's capacity to handle pattern distortion and reinforced its effectiveness in associative memory tasks.