CS786A

Computational Cognitive Science

Course Instructor :

Dr. Nisheeth Srivastava

Submitted By :

Akshat Gupta (200085)

Indian Institute of Technology, Kanpur

August 28, 2024

1 Can you write you own Hopfield network model that works more or less like the one simulated above?

The implementation of my Hopfield network can be found in the file *network.py*. It includes implementation of two different rules for weight calculation - **Hebb's Rule** and **Pseudo-Inverse Rule** [1]. It also includes functions for **synchronous** and **asynchronous** update of states.

Hebb's Rule: It calculates the weight matrix for the Hopfield Network by computing the **outer product** of the input patterns. Given a set of patterns $\xi^1, \xi^2, ..., \xi^p$, each of length N, where ξ_i^k represents the state of the *i-th* neuron in the *k-th* pattern, the weight w_{ij} between neuron i and neuron j is calculated as

$$w_{ij} = \frac{1}{N} \sum_{k=1}^{p} \xi_i^k \xi_j^k$$

Pseudo-Inverse Rule: The goal of the pseudo-inverse rule is to set the weights in such a way that the network can store a set of patterns $\{\xi^1, \xi^2, ..., \xi^p\}$ as **stable states**. This helps the network converge back to the correct pattern, even if the input is slightly noisy. The weights are calculated as follows

$$w_{ij}^{\nu} = \frac{1}{N} \sum_{\nu=1}^{p} \sum_{\mu=1}^{p} \xi_i^{\nu} (Q^{-1})^{\nu\mu} \xi_j^{\mu}$$

where $Q = \frac{1}{N} \sum_{k=1}^{n} \xi_k^{\nu} \xi_k^{\mu}$, and p is the total number of patterns. **The pseudo-inverse rule is much better at remembering a higher number of patterns as compared to the hebbian rule.**

The file *patternLib.py* includes various functions helping with **pattern generation** and **computing overlaps**. Finally, the file *vis.py* includes functions which provide facilities to **plot patterns, overlaps, and state evolution** under hopfield networks. The notebook *demo.ipynb* contains a dummy example showcasing a brief walkthrough of the Hopfield Network.

NOTE: Unless specified, the hopfield network will execute hebb's rule and synchronous updates as default.

2 Run the model with different parameters to figure out how the model's capacity to retrieve the correct pattern in response to a cue deteriorates as a function of (a) the informativeness of the cue (b) the number of other patterns stored in the network (c) the size of the network

All the results are present in the notebook *param_tuning.ipynb*.

2.1 Informativeness of Cue

As a general rule, the model's capacity to retrieve a pattern is **directly proportional** to the cue's informativeness. For example, when starting with a **49 neuron model storing 5 different patterns**, the model is able to retrieve a pattern with **100%** overlap when only 1 neuron is disturbed. However, as we disturb more neurons like **20**, the overlap falls to **35%**. As we increase the disturbance even more, the model isn't able to recover a similar pattern at all.

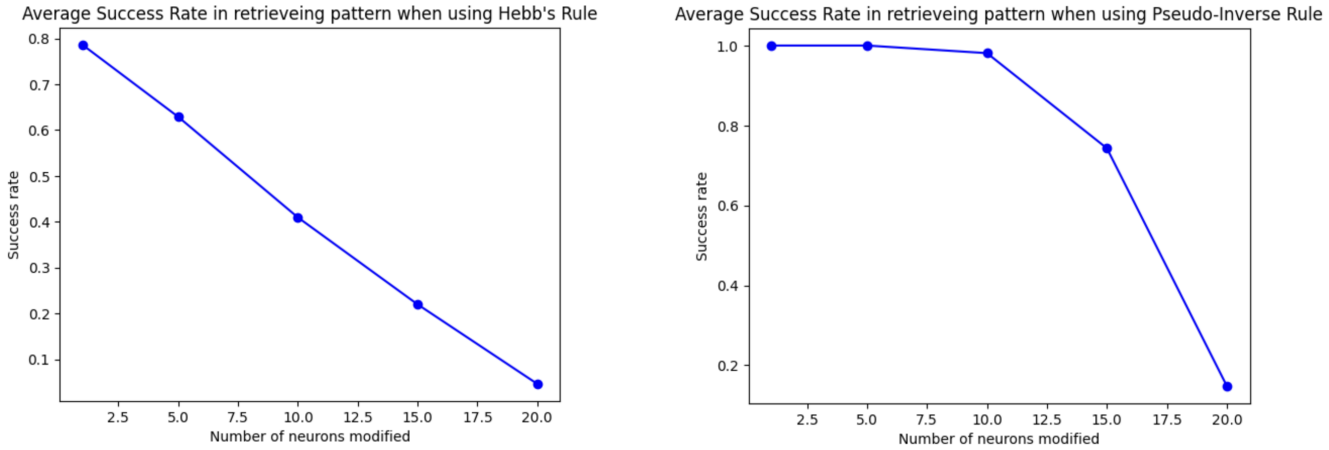


Figure 1: Average success rate vs Number of neurons modified

From the above two graphics, we can see that there is an almost **linear decrease** in success rate for a hopfield network using **hebb's rule**. It's success rate falls from about **80%** with a cue with just a **single bit-flip** to almost **40%** with **10 bit flips**. While using **pseudo-inverse rule**, the model is pretty **successful till 10 bit-flips**, but falls off after that.

2.2 Number of other patterns stored in the network

As a general rule, the model's capacity to retrieve a pattern is **inversely proportional** to the number of patterns stored in the network. As an example, using a hopfield network with **49 neurons** with an initial cue with **3** flipped bits, the final overlap drops from **100%** to **47%** as we increase the number of patterns stored from **3** to **15**.

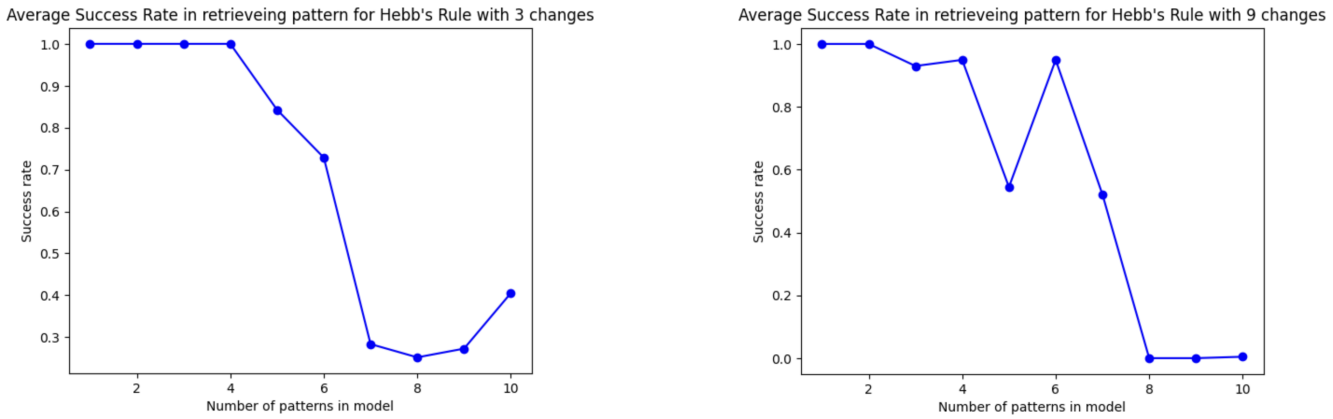
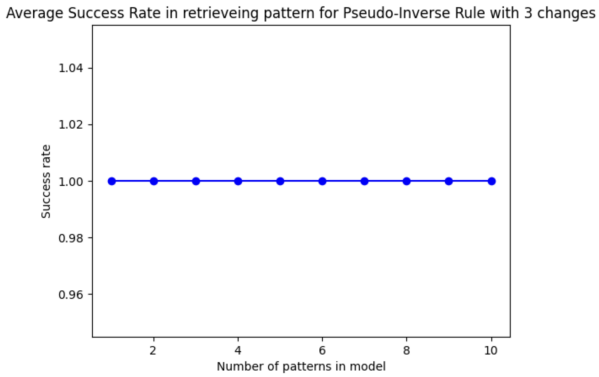


Figure 2: Average success rate vs Number of Patterns in model [Hebb's Rule]



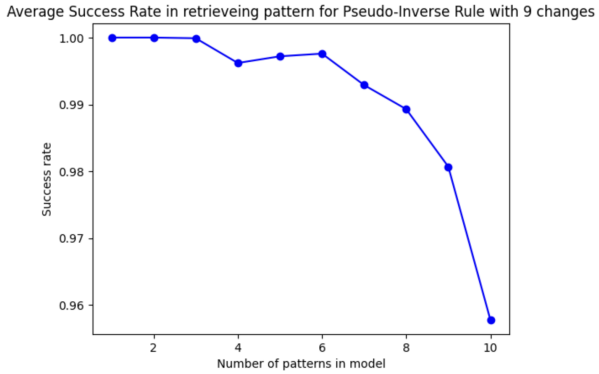
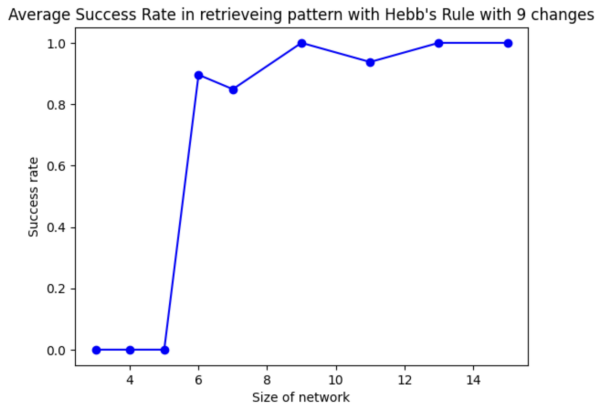


Figure 3: Average success rate vs Number of Patterns in model [Pseudo-Inverse Rule]

From fig. 2, we can see that success rate for Hopfield Network [Hebb's Rule] is stable for a while, and then **drops off drastically** after a certain point, and is almost zero when we reach about 8 patterns. On the other hand, for Hopfield Network [Pseudo-Inverse Rule] the success rate is **very close to 1** as we approach 10 patterns even with an initial cue with 9 bit flips.

2.3 Size of the network

Here again, as a general rule, the model's capacity to retrieve a pattern is **directly proportional** to the network size. As an exercise, increaseing the network size from **(3x3)** to **(15x15)** lead to an increase from an overlap of **0.11** to **1.00**, with 5 patterns stored in the network, and initial bit flip of 3.



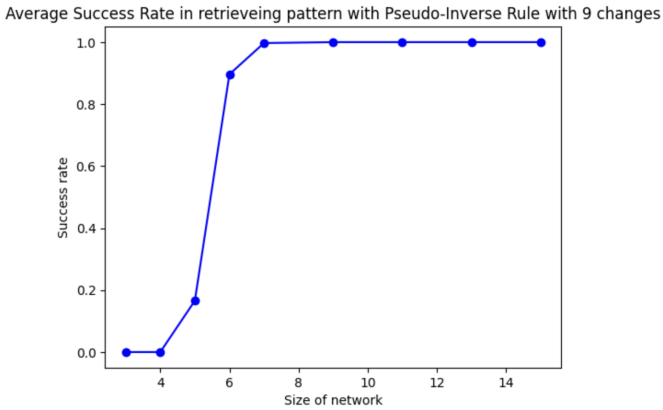


Figure 4: Average success rate vs Network Size

For both version of our Hopfield Network, we can see a **similar increase** in success rate from almost **zero initially** to close to **0.8** as we reach a network of **(6x6)** and almost **1** thereafter. Here, the difference in the two approaches doesn't seem to matter all that much.

NOTE: The average success rate over 10,000 iterations was calculated as follows:

$$\frac{\text{Number of iterations where network evolved to a state with complete overlap within 10 rounds}}{10000}$$

3 Can you write a function that converts MNIST digit data to the sort of patterns used in this simulation?

The code for this conversion can be found in the notebook *main.ipynb*. It involves mapping the pixel values of the MNIST images to -1 and 1 based on a threshold value. The threshold is selected on the basis of balanced binarization using an iterative approach.

```
# scaling factor to normalize pixel values
factor = 2.0 / np.max( X_train )

# create an average pattern for each image label (0 to 9)
def calculate_avg_classes():
    unique_labels = np.unique(y_train)
    avg_patterns = []
    for l in unique_labels:
        class_mean = np.mean(X_train[y_train == l, :], axis=0)
        scaled_mean = factor * class_mean - 1.0
        avg_patterns.append(scaled_mean)
    return np.array(avg_patterns)

patterns = calculate_avg_classes()

# Convert the patterns to {-1,1} matrices, using balanced binarization
low = -1.0
high = 1.0
iters = 10
balance = np.zeros( iters )
threshold = np.zeros( iters )

for i in tqdm( range( iters ) ):
    threshold[ i ] = ( low + high ) / 2
    balance[ i ] = np.mean( np.where( patterns < threshold[ i ], -1, 1 ) )
    if balance[ i ] > -0.7:
        low = threshold[ i ]
    else:
        high = threshold[ i ]

binary_patterns = np.where( patterns < threshold[ -1 ], -1, 1 )

# convert the test data to binarized form
X_test_binarized = np.where( ( factor * X_test - 1 ) < threshold[ -1 ], -1, 1 )
```

4 Can you write an MNIST classifier using the Hopfield network? Can you characterize its performance using F-score, and compare with classical and deep supervised learning methods? Remember that you can always use multiple samples of the same digit even for the Hopfield network classifier. Summarize your sense of the merits and demerits of using a Hopfield network as a classifier.

My implementation of Hopfield network as an MNIST classifier is present in the notebook *main.ipynb*.

The basic strategy included creating an **average binarized image for each digit** from the training data, and then using a **pseudo-inverse** rule-based Hopfield network to store these ten patterns. Now we can classify the test images considering them as input cues, and labelling them with the input pattern with the highest match.

Figure 5: Average Binarized Patterns of each digit

Results:

Digit	Precision	Recall	F1-score
0	0.95	0.68	0.79
1	0.50	0.98	0.67
2	0.85	0.55	0.67
3	0.71	0.70	0.71
4	0.68	0.73	0.70
5	0.54	0.56	0.55
6	0.84	0.70	0.76
7	0.79	0.78	0.79
8	0.81	0.53	0.64
9	0.70	0.69	0.69
Accuracy			**0.70**

We achieved an overall accuracy of **70%** with the F1-scores of each digit ranging between **0.55** and **0.79**. It is difficult for the network to correctly classify digits with similar structures (like 2,

5 and 8) with high accuracy, probably due to the high similarity between their similar states.

Comparison with Classical and Deep Learning

- Right off the bat, classical learning techniques like SVM can reach accuracies of **98.5%**, while CNN techniques can reach accuracies of **99.72%**, both of which are **significantly better** than the performance of our Hopfield Network.
- While supervised learning algorithms are trained by minimizing a specific loss function, the hopfield network model is not trianed in any conventional sense. It rather stores the image patterns implicity in its weight matrix.

Merits of using Hopfield Network

- Since hopfield networks are based on **associative memory**, they can be useful to derive original digits from noisy input.
- They are very **easy to implement**, and doesn't require state-of-the-art techniques or long training times.

Demerits of using Hopfield Network

- Hopfield networks have serious issues with **capacity constraints**. They are not able to store a large number of patterns, as the number of patterns which they can hold is proportional to number of neurons in the model.
- Since they are not built for supervised learning, they **waste the training labels**, and do not learning anything from them.
- It is **tough to extract complex features** from the images using a hopfield network, in contrast to deep learning techniques.

References

[1] Amos Storkey. Increasing the capacity of a hopfield network without sacrificing functionality. In Wulfram Gerstner, Alain Germond, Martin Hasler, and Jean-Daniel Nicoud, editors, *Artificial Neural Networks — ICANN'97*, pages 451–456, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.