Artificial Neural Networks and Deep Architectures, DD2437

Short report on lab assignment 3

Hopfield networks

Filip Husnjak, Ivan Klabucar and Corentin Royer

October 4, 2021

1 Main objectives and scope of the assignment

Our major goals in the assignment were

- to implement a Hopfield network for noise reduction in images,
- to study the proprieties of Hopfield networks using the synchronous and asynchronous update rule,
- to examine the resistance of the Hopfield network to noise and distorsion,
- to study the capacity of the network,
- to modify the training and update algorithms in order to use the network on binary patterns.

2 Methods

We used python for all our experiments. We most heavily relied on numpy and its matrix multiplication especially for the implementation of the Little Model. We also used its np.dot function to compute the dot product of two vectors in the implementation of the asynchronous update rule. The hopfield networks we coded from scratch. We used either VS Code or PyCharm as IDE. We used github for versioning and a simpler sharing of the tasks.

3 Results and discussion

3.1 Convergence and attractors

For this part of our assignment we examined some properties of an 8 unit synchronous Hopfield network or rather an 8 unit *Little Model*. After training it on three patterns:

```
x1=[-1 -1 1 -1 1 -1 -1 1]
x2=[-1 -1 -1 -1 -1 1 -1 -1]
x3=[-1 1 1 -1 -1 1 -1 1]
```

we tested the network by applying the update rule once for each pattern used during training. In all cases the network output remained the same after the update meaning the patterns were indeed stable. Hopfield networks are often used to remove noise from the input patterns, so to test this ability we fed three distored input patterns into our network:

```
x1d=[ 1 -1 1 -1 1 -1 -1 1]
x2d=[ 1 1 -1 -1 -1 1 -1 -1]
x3d=[ 1 1 1 -1 1 1 -1 1]
```

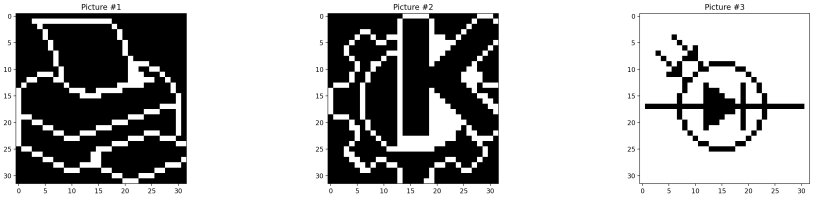
`x1d` has a one bit error, `x2d` and `x3d` have two bit errors. The network was able to recall the right input pattern only for the `x1d` and `x3d` case, while it retrieved `[-1 1 -1 -1 -1 1 -1 -1]` for the `x2d` case. This shows that if we violate some of the assumptions, namely the use of only orthogonal patterns, the Hopfield network won't function perfectly. Still, we can see that even though the network did not retrieve the correct pattern for `x2d` it did manage to get rid of some of the noise as the pattern it retured was closer to the undistorted pattern than `x2d`. The network converged in 2, 2, and 3 steps for `x1d`, `x2d` and `x3d` respectively. We can see that for a network this small in size the convergence is practically instantaneous.

Since `x2d` converged to a pattern different from any pattern in the training dataset that confirms the existance of other other undesirable fixed points in our network. We therefore checked every possible input and in total found 14 attractors in our network. All the attractors we found came in opposite pairs. When we fed very distorted patterns into our network, ones where more than half of the values were flipped, the output was usually the inverted uncorrputed pattern. This makes sense as we just determined that the opposite of an attractor is still an attractor.

3.2 Sequential Update

We continued our experiment with a new 1024 unit Hopfield network trained on a set of three 32x32 grey scale pictures which can be seen on Figure 1.

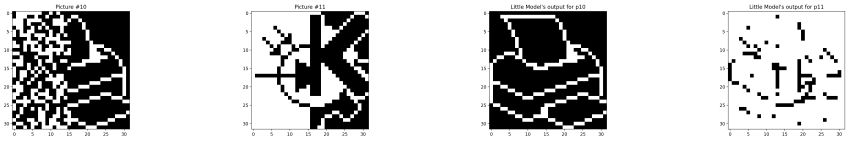
We tested the network's noise reduction capabilities by checking it's output for



(a) (b) (c)

Figure 1

the following corrupted patterns on Figure 2. As we can see 2a is just the first training pattern with right half of the image corrupted by noise while 2b is a mixture of the second and third training pattern.

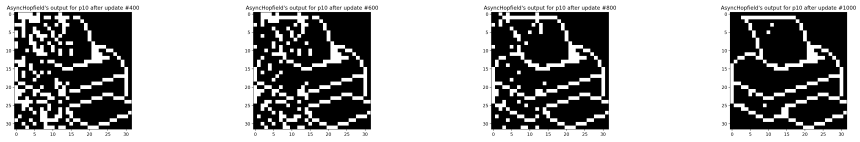


(a) Corrupted p1 (b) Mixture of p2 and p3 (c) Output for 2a (d) Output for 2b

Figure 2

As we can see the network successfully removed the noise from the picture of the printer, but in the case of the other corrupted image it returned a spurious pattern. We can attribute this result to the images being somewhat correlated and the self connections in the matrix not being set to zero. Convergence was reached in 2 and 3 steps respectively.

Using a sequential update rule for the same network proved to be much slower, because for each matrix operation we did in the previous section we now had to do 1024 (32x32) vector operations. On the other hand, the outputs of the network were much more accurate this time. The network always successfully recovered p1 from p10, and p3 from p11 most of the time. Sometimes, because of the randomness, it would return the same spurious pattern as the Little Model. Below on Figure 3 we can see the process of noise removal in action for every 200 updates starting with 400.



(a) (b) (c) (d)

Figure 3

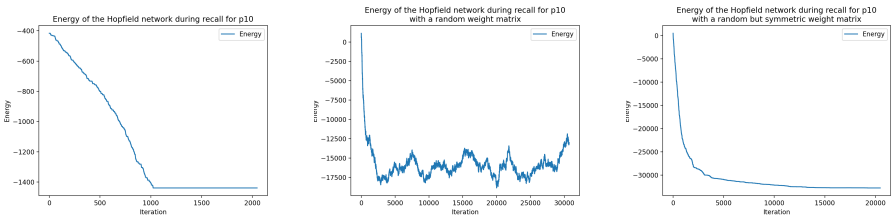
3.3 Energy

The mathematical explanation for the Hopfield network's effectiveness is that after every update the value of the energy function of the outputted pattern is lower or equal to the value of the previous pattern. This means that by simply applying the update rule some number of times, we will always reach a pattern which corresponds to a local minimum of the energy function. In other words, if our training patterns are stored in those local minima we will always reach one of them. We tested this on our Hopfield network and got the following results:

```
Energy at p1:  -1439.390625
Energy at p2:  -1365.640625
Energy at p3:  -1462.25

Energy of the distorted patterns p10 and p11
Energy at p10:  -415.98046875
Energy at p11:  -173.5
```

If we plot the energy of the patterns during the recall for p10 we can see that indeed the energy does decrease from -415.98 to -1439.39. This can be seen on Figure 4a.



(a) (b) (c)

Figure 4: Energies during recall

The assumption for using Hopfield networks this way is that the weight matrix be symmetric. To examine what happens when this assumption is violated we initialized the weight matrix to random numbers from the normal distribution and started the recall for pattern p10. As we can see on Figure 4b the recall never reached convergence, even though the tendency at the beginniing was to reduce the energy function. If we make the random weight matrix symmetric by

doing the following calculation: $w = 0.5 * (w + w^T)$ we can satisfy the mentioned assumption and reach convergence as pictured on Figure 4c. Still, because the matrix was basically random, the pattern we converge to is gibberish, as seen on Figure 5.

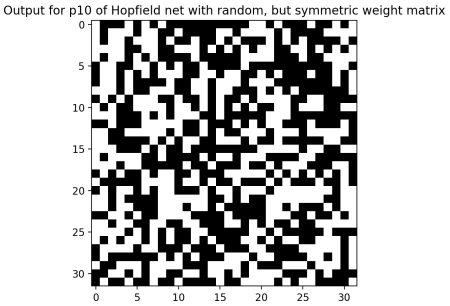


Figure 5

3.4 Distortion Resistance

In this part we studied the resistance of Hopfield networks to distorsion and noise. Most common usage of Hopfield networks is pattern recovery from noisy representations. However if there is too much noise we cannot guarantee that the network will converge to the pattern we used for training since there may be a lot of additional attractors as a result of our weight matrix. Here we used the same patterns as in part 3.2 and both LittleModel and asynchronous updates to compare the results. Patterns p1, p2 and p3 were used for training while the rest of them were generated by randomly by flipping the selected number of units in each pattern to test the network's robustness. We ran the algorithm across 0% to 100% noise and concluded that on average the network manages to recover about 20-30% of the noise depending on the pattern. The results for LittleModel network per pattern are:

- p1 - 26.6%,
- p2 - 33.7%,
- p3 - 19.8%.

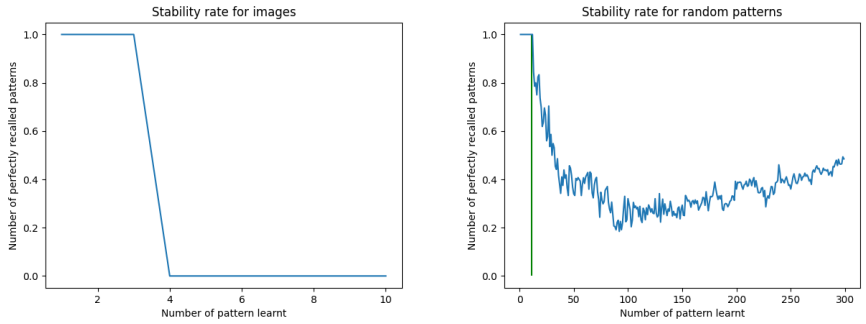
Then we used the asynchronous updates for the comparison and got the following results:

- p1 - 24.6%,
- p2 - 31.8%,
- p3 - 17.3%.

As one can clearly observe the results are similar although LittleModel network can recover a bit more noise. We also concluded that the pattern p2 is by far the most stable attractor, which means that all the other attractors are farther away. Since the network does not always converge to the attractors we used for training it means that there are other attractors as well.

3.5 Capacity

We now want to know more about the capacity of the network. For that we taught more and more patterns to the network. Each time we trained the network, we verified if the previous patterns were still recognized.
The result from this experiment with the images of the dataset is that we can store 3 patterns, when we try to learn 4 or more, the network can not recall all of them (sometimes it recalls some patterns almost perfectly and sometimes it recalls none of them). As we can see in figure 6a, this drop in performance is abrupt, after 3 patterns, the network is unable to recall even one pattern correctly.



(a) (b)

Figure 6: **6a)** Evolution of the number of perfectly recalled patterns with the increasing number of learned patterns. **6b)** Evolution of the number of perfectly recalled patterns for random patterns.

Then we tried to teach random patterns to the network instead of the images. This time, we used a smaller network ($N = 100$) and up to 300 patterns. With this network, we can recall 11 patterns correctly (see figure 6b), the capacity of our network is thus $0.11N$ which is close to the value computed by researchers ($0.138N$). This is probably because images can be correlated so the same nodes are used to recall more patterns and some nodes are not used. With random patterns, it is not the case since every sample is independent.

We can also see that the number of pattern that we can correctly recall decreases after the eleventh learned pattern. However, we can see of figure 6b that after a certain point, it starts to rise again up to 50% of the learned patterns. The

addition of noise to the pattern presented to the network for the recall process tends to degrade the performance of the network: we can not recognize the patterns as well as before (the errors are more common) even though the curve keeps the same shape.

Without self connections, the results are not better. It is actually even worse as the network cannot recognize any pattern anymore (see figure 7).

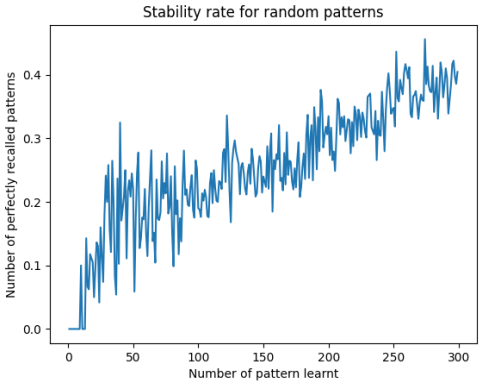


Figure 7: Evolution of the number of perfectly recalled patterns for random patterns without self connections.

The addition of a bias is also a factor that decreases the performances of the network. With a bias of 0.5 we can memorize 6 patterns and we see that the recall rate falls sharply after that. It is a problem because a lot of images are biased in the dataset.

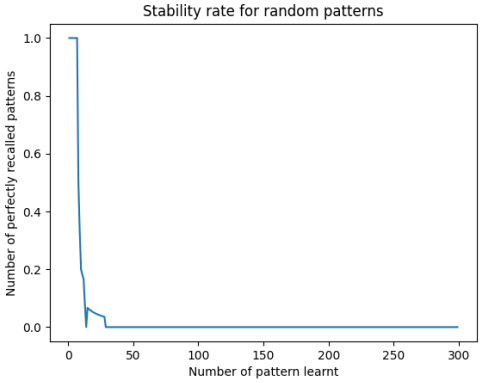


Figure 8: Evolution of the number of perfectly recalled patterns for random patterns with a bias.

3.6 Sparse Patterns

In this section we adjusted the algorithm for training and updating the Hopfield network so that it can be used for binary patterns. The patterns consist of zeros that are also called *ground state* and ones called as *active state*. The training algorithm uses average activity of the input patterns to shift the weights in order to balance positive and negative inputs. We tested the modified network on patterns with different number of active units - 10%, 5% and 1% of active units. The update algorithm uses bias to offset the weighted sum. In order to study its effects we used different values for bias between 0 and 3 and calculated maximum number of patterns that the network can store. The results are demonstrated in the figure below.

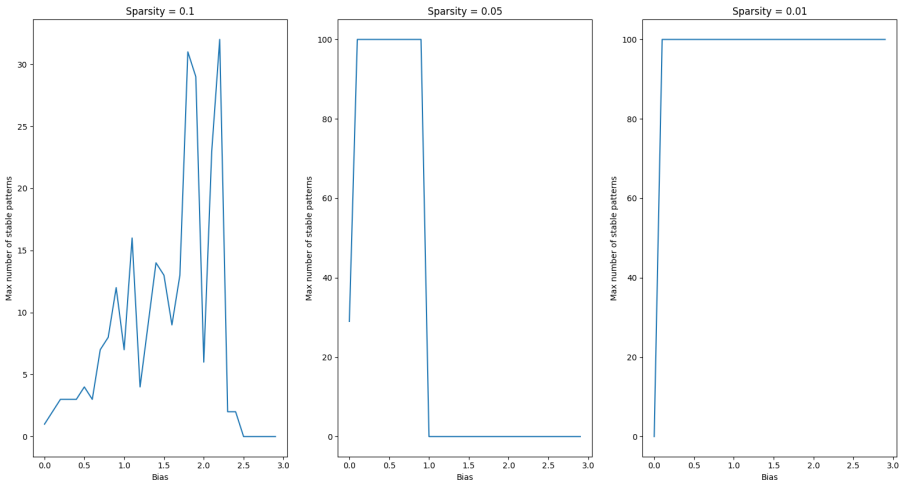


Figure 9

It can be seen that the optimal value for bias is somewhere around 2 for 10% of active units, between 0 and 1 for 5% of active units. We had to stop testing above the 100 patterns because the algorithm would take too much time so we couldn't determine whats the optimal value for 1% of active units. With lower number of active units network can safely store greater number of patterns.

4 Final remarks

During this laboratory, we saw the concrete implementation of the Hopfield network and a lot of different use case and limits of this algorithm. A few points remain unclear to us, it seems for instance that the elimination of the self connections in the network should improve the performances. We find the opposite in our experimentation. It was also very surprising to us that the

network could memorize so little with the images even though we made guesses as to why it is the case.