Artificial Neural Networks and Deep Architectures, DD2437

Short report on lab assignment 3

Hopfield networks

Etienne Jodry, Frano Rajic, Ivan Stresec

September 27, 2020

1 Main objectives and scope of the assignment

Our major goals in the assignment were:

- implementation and testing of Hopfield networks
- demonstrate how auto-associatives networks can be used to do pattern completion and noise reduction
- analysing attractor dynamics of Hopfield networks
- investigating the storage capacity and explore features that help increase it in associative memories

We construct an auto-associative memory of the Hopfield type, and explore its capabilities, capacity and limitations. We study the networks dynamics and analyse its origins.

2 Methods

We have used Python 3.8 with some of its built-in libraries along with Matplotlib and NumPy libraries for all of our experiments. We also often utilized Jupyter Notebooks and Google Sheets for result visualization and analysis.

3 Results and discussion

We modeled a configurable Hopfield network model that was used for solving all of the following tasks. The model uses self connections by default. In all of the following tasks, self connections are used unless otherwise stated.

3.1 Convergence and attractors

To start using Hopfield networks, we used a network with 8 neurons and using the Hebbian learning rule stored three 8-dimensional patterns in it. We used batch learning for this task. All three patterns were, as expected, stable states, i.e. attractors.

When testing the network using patterns with a 1-bit mistake, and two 2-bit mistakes. The network converged to a stored pattern for the 1-bit mistake and one of the 2-bit mistake patterns, but didn't converge to a stored pattern for the second pattern with the 2-bit mistake.

Testing all possibilities, we found that the network has 14 attractors. When using very distorted stored patterns the convergence often leads to another attractor, i.e. fails to complete the distorted pattern.

3.2 Sequential Update

For this subassignment we loaded patterns from `pict.dat` and created a 1024-dimensional network to handle them.

Our network successfully learned patterns/pictures p1, p2, and p3. We could observe all of them were stable states, an update starting from one of those pictures would never produce any change in neuron activity.

Giving the network pattern p10, which is a degraded pattern p1, would converge to p1, but giving the network pattern p11, a mix of p2 or p3, would converge to a stable state, but that state was neither p2 or p3. If using sequential updating (adding a stochastic effect to updating) the network would sometimes converges to p3.

Giving the network a random state and using the sequential updating rule leads to convergence to one of the attractors (stable states). In Figure 1 we can see the process of convergence to a stable state.

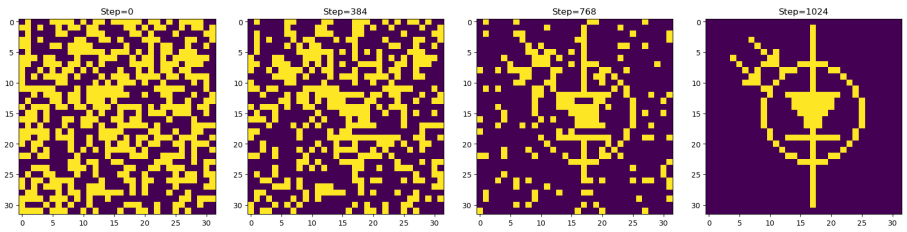

Figure 1: Convergence process using the sequential updating rule

3.3 Energy

Our 3 first attractors are the learnt patterns, then we sampled 1000 randomly generated patterns and executed the update rule until convergence for each of them, to find new attractors.

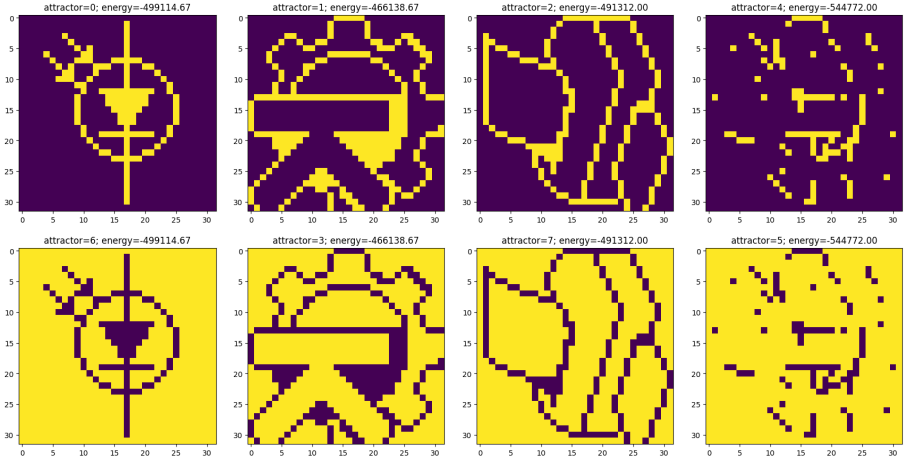

Figure 2: Attractors and their energies, for network that learnt pictures 1, 2, 3. Learn patterns are the first three images in first row. Other images show spurious states.

We found 8 attractors in total, it is interesting to note that each attractor has it's "negative" counterpart of same energy. The learnt patterns and their counterparts have roughly an energy of $-5 * 10^5$ each. The global minimum is the newly found attractor that we did not learn (last column on fig 2) with an energy of -544772. This spurious state somewhat reminds of the mixture of pictures in columns 1 and 3.

We computed energies for the following distorted patterns:

- $E(p10) = -141988.0$
- $E(p11) = -59221.33$

Those have a much higher energy compared to the attractors, which is normal because attractors lie in local minima.

In Figure 3 we see the curve of convergence from a randomly generated pattern to an attractor using the sequential update rule, the energy of the state is steadily decreasing, going downhill direction. It converges in 1 update epoch, after which another update epoch is made to ensure the pattern's stability.

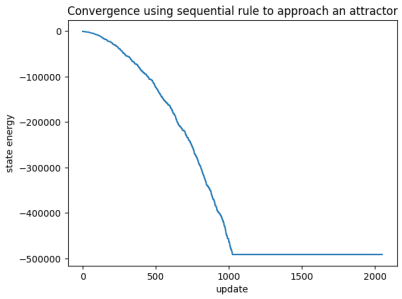

Figure 3: Convergence from a randomly generated pattern to an attractor

3.3.1 Random weight matrices

Asymmetric case We studied the evolvement of the energy of the state of the model when initializing the matrix at random. Figure 4 compares batch and sequential (1 epoch is 1024 updates) mode.

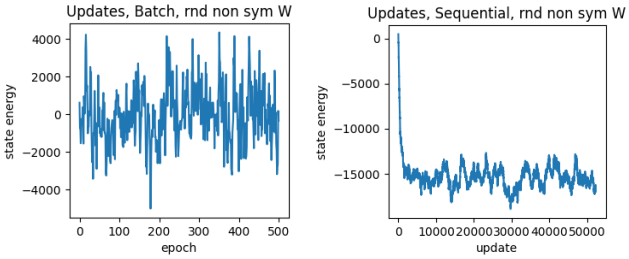

Figure 4: Evolution of the energy of the state when initializing the weights as a random asymmetric matrix. Both models stopped by reaching a maximal number of iterations.

In batch mode this is terrible, the energy bounces back and forth around 0, no convergence nor stabilization happens. In sequential a minimization occurs, but the system does not seem to be able to find a stable state.

Symmetric case By turning the weights to a symmetric matrix (i.e. $w = 0.5 * (w + w^\intercal)$), we can observe different results in Figure 5. Both models are leading to a stabilization when performing updates.

However in batch mode, it leads to an oscillation between two states and its energy remains very close to 0 which is very high when compared to the stable state found by sequential update rule, which is around -30000 in this case.

The Observed difference between a converging symmetric and non-converging asymmetric weights matrix can be explain by the fact that the Hopfield network was designed to minimize energy values as defined by a symmetric matrix and the fact that the designed update rule is designed to accomplish that minimiza-

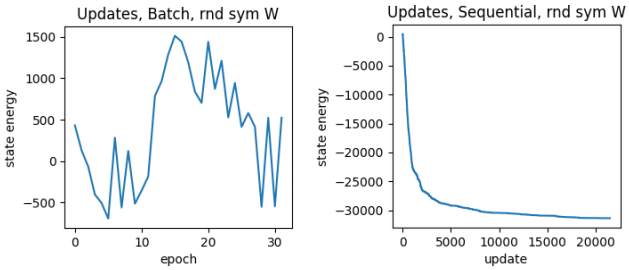

Figure 5: Evolution of the energy of the state when initializing the weights as a random symmetric matrix. Both models stopped by stabilization.

tion. Using an asymmetric weights matrix breaks the first assumption needed to prove the convergence of a Hopfield network to a stable state.

3.4 Distortion Resistance

We added noise by flipping a certain percentage of the 3 learnt images and ran the network on this data to try to retrieve original pattern. Results (averaged over a few runs to make the curve smoother) are seen in Figure 6.

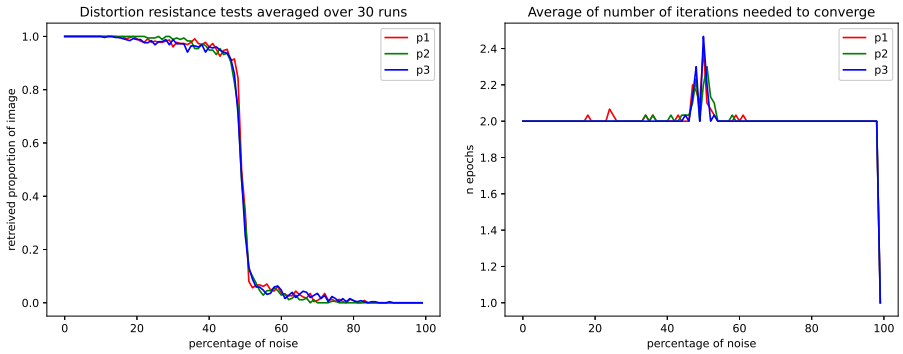

Figure 6: Performance of the network in respect to noise

We did not observe any difference in the behaviour of the network according to the attractor that was being retrieved. Around 50% of noise, there is a very steep downhill in the curve. After that point it is nearly impossible for the network to find the original pattern. However, before this point is reached, the network shows good capabilities for this task.

Except in the "grey zone" of around 50% noise, the little model stabilizes in approximately 1 epoch, and even in the "grey zone" it just slightly goes up in average number of epochs. This shows that extra iterations did not help much in this case, the network would mostly stabilize in just one iteration. This observation can be explained by the fact that only three patterns are stored and

not much of the networks capacity is yet used. As shown in Figure 2 we found another attractor (and its negative counterpart). After this noise threshold, the model tends to converge to another attractor than the pre-noise one.

The last scenario is at a noise level of 100% (the pattern is fully flipped) and since the negative is also a stored pattern, it is also a stable local minima.

3.5 Capacity

In this task, we observed how adding more memories affected the stability of the network, using retrieval rate of non-distorted and moderately distorted patterns as a measure of the networks capacity.

Firstly we tried adding more memories to the initial three pictures (p1, p2 and p3) and observed that adding just one additional memory would prevent the network from converging to learnt patterns (Fig. 7), given either non-distorted or distorted images. Only first three patterns can be stored safely with perfect pattern retrieval, whereas for four patterns, the performance would abruptly drop to 0% retrieval.

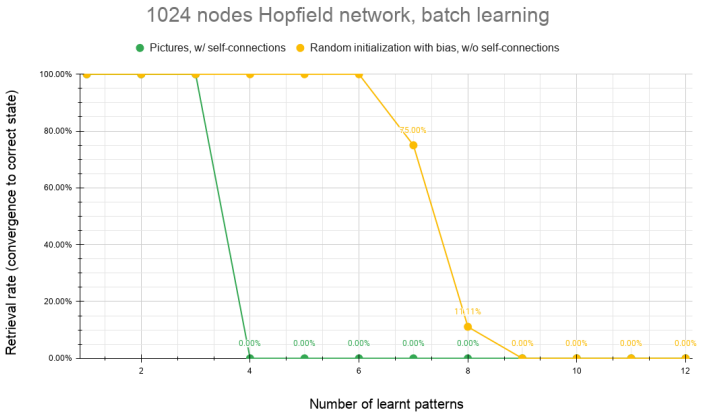

Figure 7: 1024 nodes Hopfield network retrieval rate performance comparison for learning a handful of pictures and biased random vectors

On the other hand, when using random vectors generated using $\mathcal{N}(0, 1)$ (and then computing $sign()$) instead, we noticed enormous difference in performance metrics when adding more patterns to learn. The network could store up to 53 patterns with perfect retrieval for patterns with distortions of maximal 20% and up to 73 patterns with retrieval higher than 90% percent for patterns with up to the same amount of distortions, as shown in Figure 8. Learning more patterns gradually decreases the performance, until after 200 learnt patterns when the retrieval rate is very low and approaches 0%. In Figure 8 one can also observe that the retrieval rate performance on non-distorted patterns is

constantly higher that the performance on patterns with noise added. This difference in behaviour means lower robustness towards noise and higher noticeable degradation of basins of attraction.

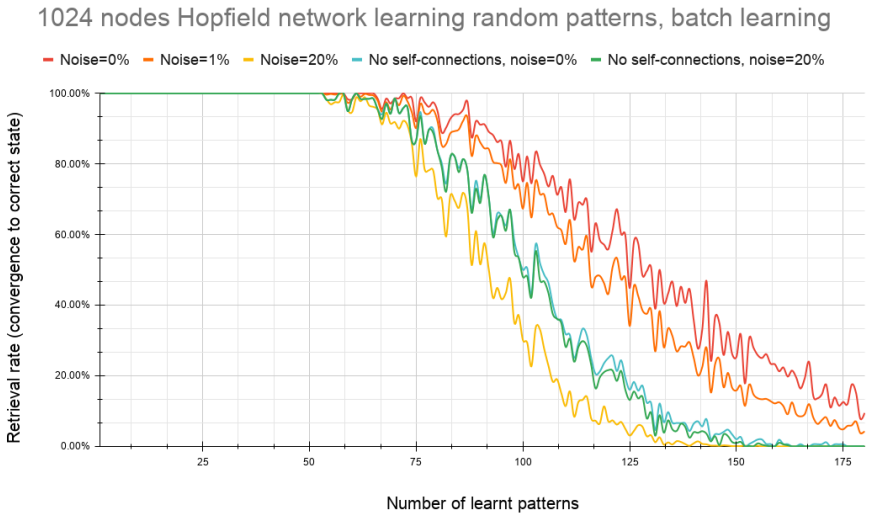

Figure 8: 1024 nodes Hopfield network retrieval rate performance relation with different number of learnt patterns for learning random vectors with and without noise and self-connections. The cyan and green line overlap.

The huge difference in comparison with learning pictures can be explained by noting that the pictures are much more correlated than the randomly generated vectors.

In further investigation, we removed the self-connections w_{ii} and observed that the difference between the curves from pure and noisy patterns, with up to 20% of distortions, had no noticeable difference (Fig. 8, cyan and green line overlap). Even though removing the self-connections seems to lower the memory performance (Fig. 8, cyan and green line are below red and orange line), it is generally preferred not to have self-connections because they promote formation of spurious patterns and have a negative effect on noise removal capabilities. Having no self-connections results in higher noise robustness and thus more useful as well as more uniform basins of attraction, as can be seen in Figure 8 where the cyan and green line are above the yellow line.

Additionally, we generated biased patterns with using $\mathcal{N}(0.5,\, 1)$ instead of $\mathcal{N}(0,\, 1)$, so the patterns had more 1's than -1's. The observed retrieval rate was drastically worse compared to the former case where no bias was used. The results were very similar to observed measures for pictures in that both had abrupt retrieval rate performance drop for less than 10 learnt patterns. The similarity can be seen in Figure 7.

3.6 Sparse Patterns

For our final subassignment we had to see how the network performed when learning sparse patterns, as well as observe the effect of adding constant bias. For testing we used a 100-neuron network and, ipso facto, 100-dimensional patterns. The learning and update rule were modified as specified in the subassignment in order to achieve a {0, 1} binary output (as opposed to the {-1, 1} used before). In what follows, we describe only the observations we made, but we were not able to draw conclusions or base the results on intuitive basis.

With self-connections, learning patterns with an activity of 10% for different bias (θ) values resulted in various results. The network performed best using $\theta = 0.3$ and could store up to 25 sparse patterns. This is a large number in relation to the dimensionality of the network.

For a lower bias of around $\theta = 0.05$ if the network is given too many patterns it saturates and becomes stable in all states (the diagonal becomes the strongest part of the weight matrix), allowing it to learn any amount of patterns, but rendering it useless. Using a larger bias like the aforementioned $\theta = 0.3$ doesn't allow this to happen as the bias drags everything into 0 (Figure 9).

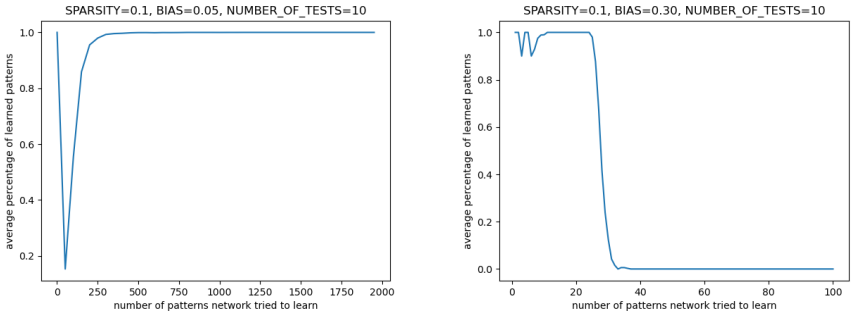

Figure 9: Percentage of learned patterns vs. number of patterns network tried to learn for $\theta = 0.05$ and $\theta = 0.3$

Without self-connections the network could reliably learn up to 10 10% patterns, 6 5% patterns, and about 100 1% patterns. If we add any bias to the 1% pattern network it drags everything into 0 making the network useless.

4 Final remarks

This assignment was an interesting insight into the workings of Hebbian networks. It was slightly easier than the first two labs, but since the tasks were quite sequential in relation one to another it was hard to split the work between team members efficiently.