Artificial Neural Networks and Deep Architectures, DD2437

Short report on lab assignment 3

Hopfield networks

Alex Norlin, Athanasios Charisoudis and Giovanni Castellano

February 23, 2022

1 Main objectives and scope of the assignment

The main objective of this lab was to implement and study Hopfield networks. In particular, we focused on the recall operation and its main aspects, networks' distortion resistance, their capacity, and the energy landscape of them.

2 Methods

We decided to use the programming language Python to implement and test the two algorithms. Thanks to the library Numpy we could easily perform mathematical operations efficiently. Moreover, we used matplotlib to plot results.

3 Results and discussion

In this section, the results of our main experiments are given. Initially, we store in our 8-D Hopfield network 3 patterns and test its recall capabilities. Subsequently, we do the same for high-dimensionality (1024-D) inputs. Then we observe the recall trajectory in the energy landscape, test the models resilience to distorted inputs and measure its capacity on dense as well as sparse patterns.

3.1 Convergence and attractors

In this first question, we were asked to store in the 8-node Hopfield net, 3 patterns. Then, starting from noisy versions of themselves, we checked if our net converged to the correct outputs (i.e. if it successfully denoised the recall's inputs). Sequential (asynchronous) recall was used. In what follows, "CanRecall" checks if the pattern is actually an attractor, whereas "DidRecall" checks if recall's output was same as the input.

- [**-1 -1 1 -1 1 -1 -1 1**]: `CanRecall: True`, `DidRecall: True` (2 iters)
- [**-1 -1 -1 -1 -1 1 -1 -1**]: `CanRecall: True`, `DidRecall: False` (2 iters)
- [**-1 1 1 -1 -1 1 -1 1**]: `CanRecall: True`, `DidRecall: True` (2 iters)

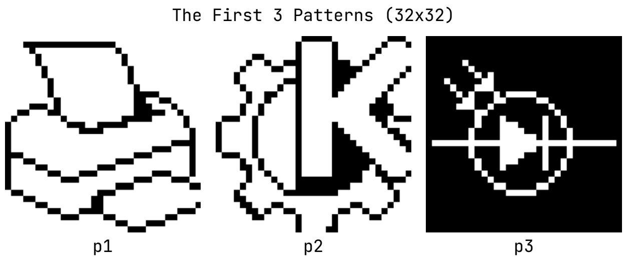

Figure 1: The first 3 input patterns.

As can be seen, p2 could not be recalled, even though it is an attractor. The entire list of net's attractors are:

- **[-1 -1 -1 -1 -1 1 -1 -1]**
- **[-1 -1 1 -1 -1 1 -1 1]**
- **[-1 -1 1 -1 1 -1 -1 1]**
- **[-1 -1 1 -1 1 1 -1 1]**
- **[-1 1 -1 -1 -1 1 -1 -1]**
- **[-1 1 1 -1 -1 1 -1 1]**
- **[-1 1 1 -1 1 -1 -1 1]**
- **[1 -1 -1 1 1 -1 1 -1]**
- **[1 1 -1 1 -1 1 1 -1]**
- **[1 1 -1 1 1 -1 1 -1]**
- **[1 1 -1 1 1 1 1 -1]**
- **[1 1 1 1 -1 1 1 1]**
- **[1 1 1 1 1 -1 1 1]**

If we add further noise to recall's inputs, then as expected the behavior is worse. In particular, if we flip 5/8 bits, we have:

- [1 1 1 1 -1 1 -1 1] (original: **[-1 -1 1 -1 1 -1 -1 1]**): `CanRecall`: `True`, `DidRecall`: `False` (2 iters)
- [1 1 1 1 1 1 -1 -1] (original: **[-1 -1 -1 -1 -1 1 -1 -1]**: `CanRecall`: `True`, `DidRecall`: `False` (3 iters)
- [-1 1 -1 -1 1 -1 1 -1] (original: **[-1 1 1 -1 -1 1 -1 1]**: `CanRecall`: `True`, `DidRecall`: `False` (2 iters)

3.2 Multidimensional Inputs

In this section, we test Hopfield nets performance on nine (9) 1024-D (32x32) input patterns, the first 3 of which are given in figure 1. To check if the inputs are "stable", we initially test if starting from each one of them, we can recall it. Indeed, as we found, **all 3 input patters**, p1-3, **are stable**.

Next, we check the net's performance on noisy inputs. In particular, p10-11, are used which are degraded version of inputs p1 and p2-3, respectively. As can be

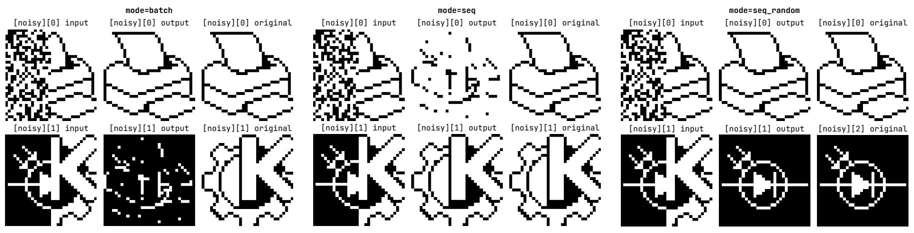

Figure 2: Recall performance on noisy inputs, for batch (left triplet), sequential (center) and random-sequential (right). First row: p10 (noisy) vs p1 (clean). Second row: p11 (noisy) vs p2/p3 (clean). In every triplet: input (left column), output (center), original (right).

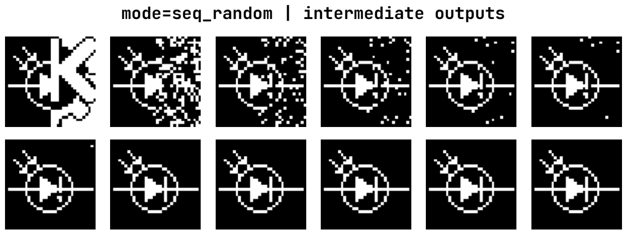

Figure 3: Intermediate outputs when running in random-sequential mode, given p11 as input. The outputs are taken every 10 iterations.

seen in figure 2 for all recall modes (batch/synchronous, asynchronous/sequential and random-asynchronous/random-sequential), the net exhibits some robustness to degraded patterns. More specifically, p1 can be recovered from p10 in batch and random-sequential recall modes; it diverged to a spurious attractor in sequential recall mode. p11 results in p2 in sequential mode and in p3 in random-sequential one; in batch mode it results to a spurious attractor which is the flipped version of the previous spurious attractor. For the random-sequential mode, a random node was selected at each iteration, with the probabilities being the inverse of the times a node was selected. The patterns towards convergence when the input is p11 and target output is p3 are given in figure 3.

3.3 Energy

Starting with examining the energy at the different attractors we find that when storing the first three images the energy for those attractors in the network where -491312, -4661396 and -499115. Now this alone doesn't tell us much, hence we compared it to the the distorted version of image 1 and 2. The result of this was an energy level of -141988 and -59221 respectively. This seems right as we see when plotting the energy trajectory, see figure 4 (a), as the energy should be lower for the stored pattern than the distorted ones in most cases.

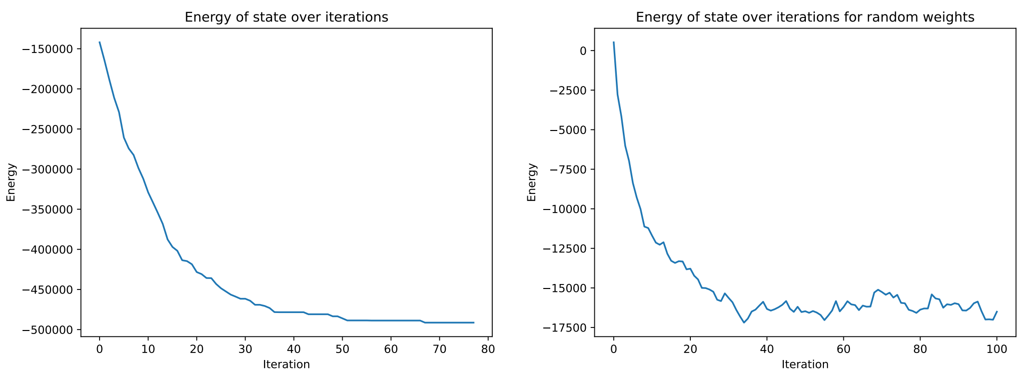

Figure 4: Energy level over a number of iterations for (a) the normal case (b) randomized weights in the network. As one can see, in case (b) the network doesn't converge as the energy increases on some iterations.

Following that we tested what would happen if one used randomized weights in the network instead of calculating them the usual way. The result can be seen in figure 4 (b). It is evident that it doesn't converge which is not surprising, the resulting state looks like noise. If you make this randomised weight matrix symmetric you do converge to a result. The result still do look like noise however.

3.4 Distortion Resistance

In figure 5, we test how much of a noisy input is a Hopfield net able to resist. In particular, we test with 2 and 3 1024-D inputs, and as can be seen, with around 30% of bits flipped the network converges to the desired pattern with high probability. However, if we keep increasing the amount of noise, the performance of the network decreases. We notice that in both cases the network cannot reconstruct images with more than 50% of noise. It is interesting to notice that the performance decreases very quickly, moreover, in general the network can withstand more noise if there are less stored images.

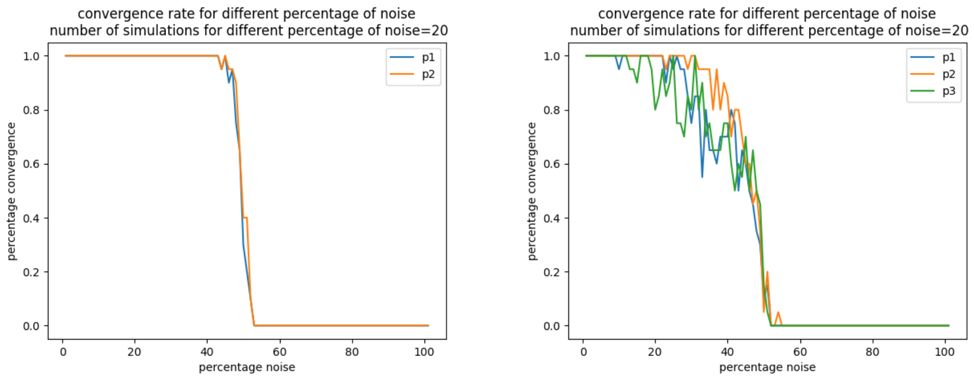

Figure 5: How much noise can be removed from the images. (a) storing 2 images (b) storing 3 images.

We noticed also, that the network needs a very small number of iterations to converge. During our simulations with this data, the network never needed more than 3 iterations; although, it is clear that if the noise is too much the network will converge to the wrong pattern. We know for instance, that if a pattern x is stable, then also $-x$ will be stable.

3.5 Capacity

Using the same dataset, we cannot add a fourth image to the network, in fact, when we tried to do it, the network was not able to reconstruct any of the stored images, not even with 0% of noise. We tried to perform the same experiment using some images generated randomly. We noticed that with such kind of data, the network is able to store up to 70 images. The problem with our first dataset is that the images are very similar, generating random patterns in a high dimensional space instead, the results are very likely to be orthogonal, therefore the capacity of the network increases.

Afterwards, we studied the behaviour of a network with 100 neurons. In particular, we were interested on how the number of stable patterns changes as we increase the stored patterns. In figure 6(a) we can observe two curves; the blue one refers to the stability of patterns starting from the original patterns. The orange one instead, refers to stability checked starting from a noisy version of the patterns (10% noise). Surprisingly, the curves are very different. However, we should only take into consideration the orange curve. In fact, as the number of stored patterns increases the network always tends to output the given input. This does not mean that the network is behaving as an associative memory, and therefore the blue curve is not significant. Zooming the plot (figure 6(b)) we can see that highest value for the orange curve is about 10, that is the capacity of the network. We can notice in figure 7(a) that if we set the diagonal of the weight matrix equals to zero, the blue curves becomes very similar to the orange one. This happens because is the diagonal of the matrix that force the network to output the given input. Moreover, in figure 7(b) we can observe that the capacity decreases if we add some bias to the patterns.

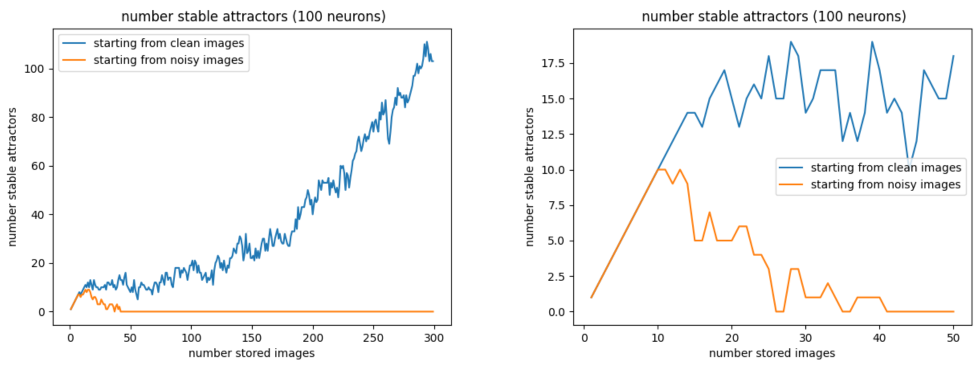

Figure 6: Number of stable images in the network as we increase the number of stored images. (a) original (b) zoomed

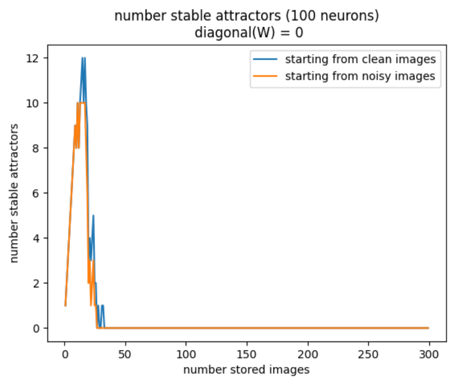

Figure 7: Number of stable images in the network as we increase the number of stored images setting diagonal(W) = 0 (a) without bias (b) with bias

3.6 Sparse Patterns

Using mostly the same methods as previously with setting the diagonal elements to 0 we tested how well the network could remember sparse binary patterns with a given activity with different biases. We did normalize the weights in this network. In figure 8 (a) is the result for different biases using 10% activity and in figure 8 (b) is the result for different biases using 1% activity. Generally it seems that using a small bias (and even smaller for even sparser patterns) is better. We are however, at a loss for why the number of stored patterns increase after the initial decrease in some cases.

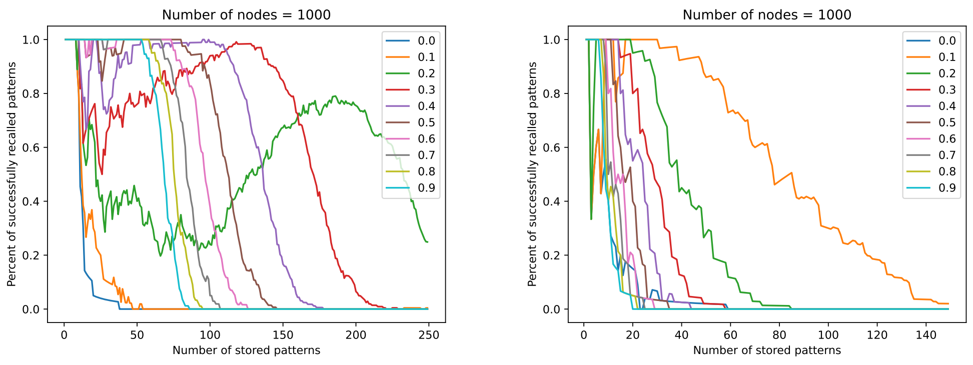

Figure 8: Percentage of stored patterns that the network could successfully recall with the colored lines representing networks with different bias. (a) Is for sparse patterns with 10% activity while (b) Is for sparse patterns with 1% activity.

4 Final remarks

Not much to note here. The sparse patterns part seemed weird since the result was very confusing. Otherwise a good and fun lab.