Artificial Neural Networks and Deep Architectures, DD2437

Short Report on Lab Assignment 3

Hopfield Networks

Ahmet Avci, Ryan Davis and Sifan Jiang

February 19, 2019

1 Main Objectives and Scope of the Assignment

Our major goals in the assignment were

- Test various parameters which affect Hopfield network's performance. Track energy of networks and how it relates to convergence.
- Analyze how adding a bias term affects capacity with sparse patterns.

2 Methods

I this lab, we used MATLAB to work with the first two parts to test the properties of the convergence and attractors and the sequential update. We used Python for the rest parts to investigate the properties of energy, distortion resistance, capacity, and sparse patterns.

3 Results and Discussion

3.1 Convergence and Attractors

- The network was able to store the patterns x_1, x_2, and x_3 because the same patterns are received when applying updating to these three patterns. For the distorted versions, x_1d and x_3d would converge to their origin versions, while the second state of x_2d is different to its origin pattern. However, x_2d converges to x_3.
- Trying all possible patterns, 14 attractors are found in this network which are listed in tab 1.

Table 1: All attractors in the network.

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
-1	-1	-1	-1	-1	1	-1	-1

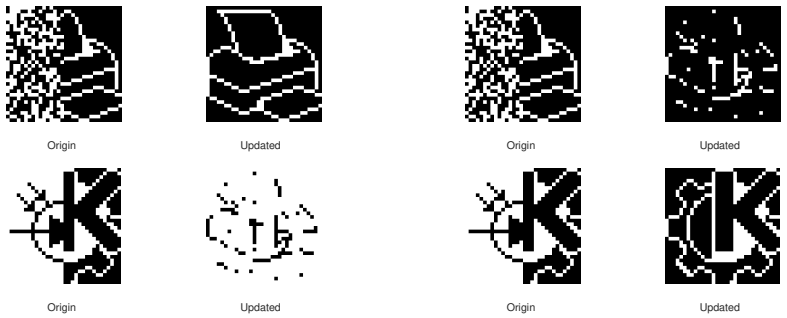
- If the starting pattern is every dissimilar to the stored ones, for example more than half is wrong, the output would not vonverge to any of the stored patterns. However, it would converge to one of the other attractors.

3.2 Sequential update

- The three patterns are stable since after training the network with these three patterns and applying the updating to these three patterns, the same patterns would be returned.
- If we use synchronous update, the network would be able to complete a degraded pattern, but unable to a mixture pattern. If we use asynchronous update, the network would not be able to complete a degraded pattern, but able to comple a mixture pattern. The result is illustrated in fig 1.
- When using sequential update and only one unit is updated in each iteration, and the unit is randomly selected, it takes large iterations to converge and the new state would converge to the most similar attractor. The result is illustrated in fig 2.

3.3 Energy

We find that the energy is ~ 1400 at each of the three trained attractors x_1, x_2, and x_3. The energy is much higher at the distorted patterns x_{10} and x_{11} which have -412 and -170 energy respectively. This makes sense as we expect energy



(a) Synchronous update. (b) Asynchronous update.

Figure 1: Picture completion.

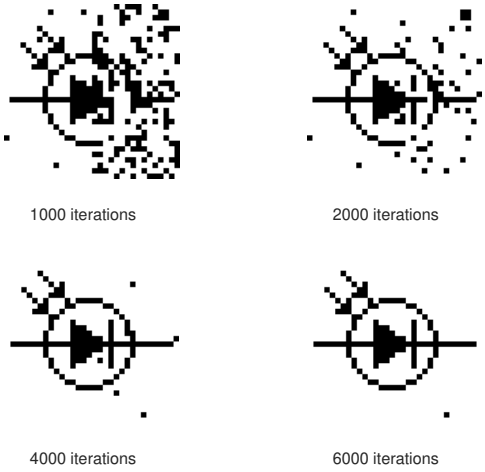


Figure 2: Randomly selected units.

to lower as the network's node activations travel towards convergence at a low energy state.

Tracking the energy changes from iteration to iteration when using the sequential update rule shows that the energy decreases each iteration until convergence. For example, using the pict.dat data trained on x_1, x_2, and x_3 and then tracking the energy change of x_{11} shows that the energy starts at -170.5, decreases to -1307.1 after one iteration, decreases to -1593.0 after a second, and stays at -1593.0 as it has converged to the trained attractor of x_3. The energies are shown in tab 2.

When using a weight matrix with normally distributed random values instead of one trained with Hebbian learning, we see that the network's energy randomly changes values and fails to converge. Which is shown in fig 3.

Table 2: Energy of the different attractors.

1	-1436.390625
2	-1362.640625
3	-1459.25
4	-717.48046875
5	-522.890625
6	-680.296875
7	-682.73046875
8	-168.546875
9	-264.51171875
10	-412.98046875
11	-170.5

When making the weight matrix symmetric, we see that x_{11} starts at an energy of -0.085. After a single iteration, the network has an energy of -1.87. Beyond this, the network is unable to make any movement towards an attractor and stays at -1.87.

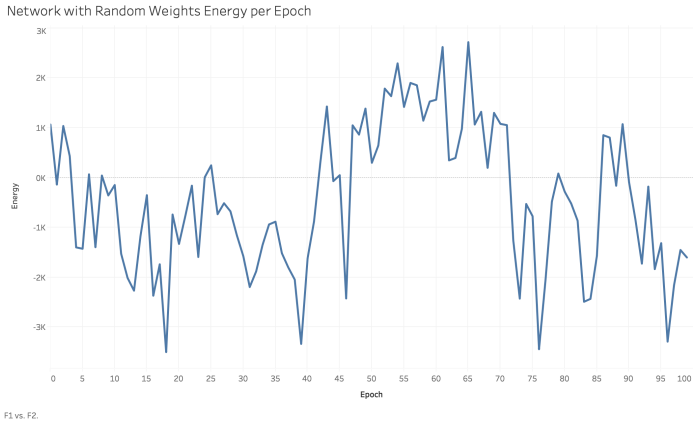


Figure 3: Network with random weights energy per epoch.

3.4 Distortion Resistance

For each percentage of noise and each trained attractor, we generate 10 noisy attractors each with a percentage of bits flipped from the original attractor.

Plotting the results of using the network to attempt to de-noise the generated noisy attractors in fig 4 shows some interesting results. We see that if the amount of flipped bits is kept below about 20% the network does a very good job at de-noising the inputs and converging back to the original attractor.

Each of the attractors behave similarly with respect to the amount of noise added with poor performance with noise percentages from 30 to 50 and then

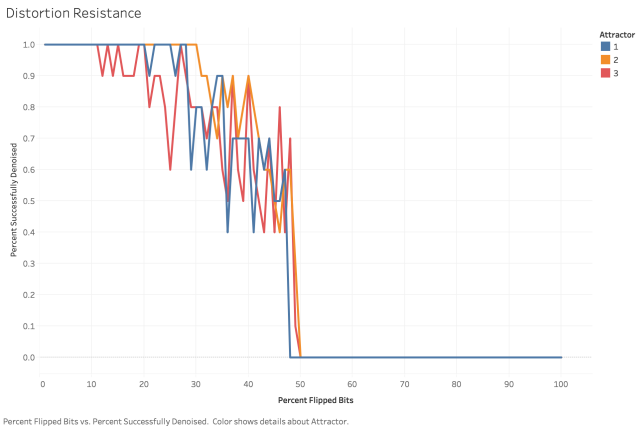


Figure 4: Distortion resistance.

zero accurate convergences after 50 percent noise.

With higher levels of noise we see that the network sometimes converges to an untrained attractor or to the incorrect attractor. For example, a noisy x_1 might converge to x_2. We find that additional iterations do not help at all as the network converges after one iteration (using synchronous updates).

3.5 Capacity

Testing the storage capacity of the network by adding more memories beyond $x1$, $x2$, and $x3$ shows that even just a fourth memory drastically reduces the de-noising capability of the network. In fact, the network can not even recall $x1$, $x2$, and $x3$ with no noise when the fourth memory is trained as well. Training for 5, 6, and 7 memories does not change this result. The result is shown in fig 5.

When using randomly generated memories instead of those stored in Pict.data we see that the storage capacity of the network is greatly increased. We plot these results in fig 6. We don't see the extreme drop off at 4 memories and instead are able to increase the memory count to beyond 13 with out drop off. We also see that for random patterns the network performs better at de-noising memories. For 3 memories the network has perfect de-noising all the way up to 44 percent flipped bits.

The difference in performance from Pict.dat memories and randomly generated memories can be explained by realizing that randomly generated memories are more likely to be more orthogonal to each other than the pictures. The 0.138 storage capacity (which would correspond to ~ 141 memories for a 1024 dimension input) assumes orthogonality between memories.

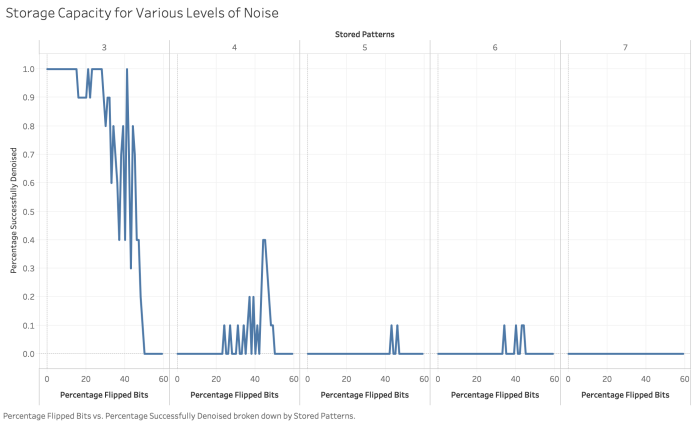


Figure 5: Storage capacity for various levels of noise.

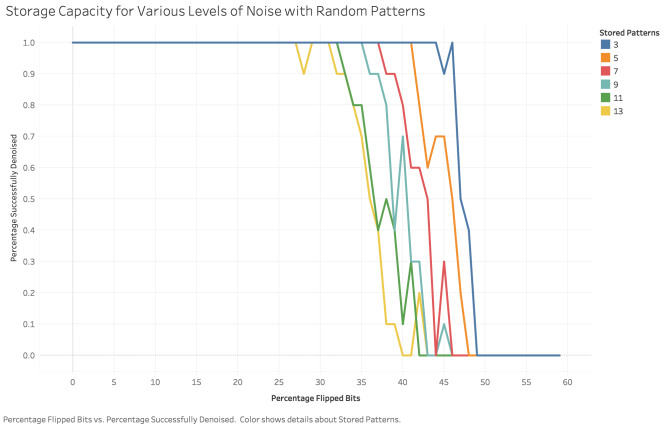


Figure 6: Storage capacity for various levels of noise with random patterns.

Now we generate 300 random patterns of 100 dimensionally and test how many of them remain stable for different numbers of trained memories. In fig 7 we see that all trained memories remain stable until the number of trained memories exceeds 11. After 16 trained memories, the number of stable memories starts decreasing. Beyond 35 trained memories, no memories remain stable. This shows the effects of memory capacity beyond just de-noising, memories themselves begin to diverge as too many are stored.

If instead of testing the number of stable memories, we test the number of slightly noisy (2 bits flipped) memories which converge to the correct memory, we see an extremely similar result with slightly fewer memories being successfully de-noised than non-noisy memories remaining stable. This result holds across just about all memory counts and is not surprising. While 11 trained memories can all remain stable, only about 7 trained memories can all correctly converge when two bits are flipped.

When generating the random patterns and biasing the bits in the patterns to

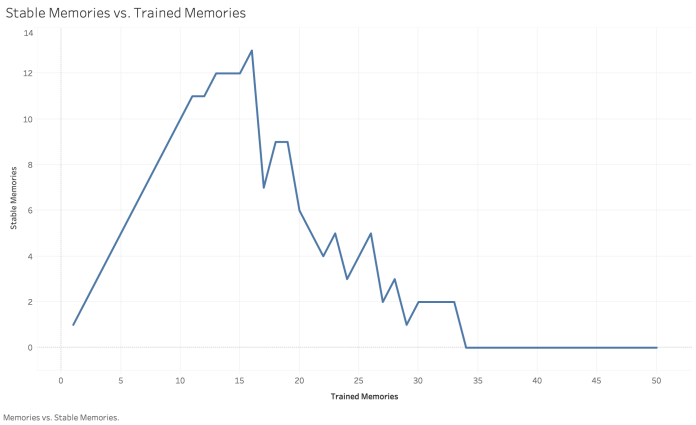


Figure 7: Stable memories vs. trained memories.

be 1 instead of -1 (with 2:1 odds) we see that the number of stable patterns decreases even further. Past 5 trained memories the number of stable memories never increases and past 8 trained memories the number of stable memories decreases. This is explained by the same reason as the difference between the pictures and random memories above; biasing the bits towards 1 decreases the orthogonality of the vectors and lowers the memory capacity far beneath the ideal 0.138.

3.6 Sparse Patterns

Now we test the number of memories which remain stable as the number of stored memories increase when using sparse memories and a network with a bias term. In fig 8 we see that for memories with 0.1 average activity, 0.025 and 0.05 are good values for the bias term. 0 is too low and results in a storage capacity of roughly four memories. Biases above 0.05 are too high and also result in poor storage capacities.

When decreasing the average activity to 0.01 we immediately see the surprising result that while not all memories remain stable, the total number of stable memories continues to increase as we increase the number of memories stored for 0 to 100 memories. This is true for all tested levels of bias (the colors not shown are hidden under the red 0.05 bias line). We also see that the optimal tested bias changes from 0.05 to 0.025. This can once again be explained via the concept of orthogonality. For 0.01 average activation, it will be common for a single one of the 100 bits to be activated. If there are 100 vectors each with a single activated bit and that bit is different than the others, then the 100 vectors will each be orthogonal to each other. In the 0.1 average activity case, there will be less orthogonality among vectors.

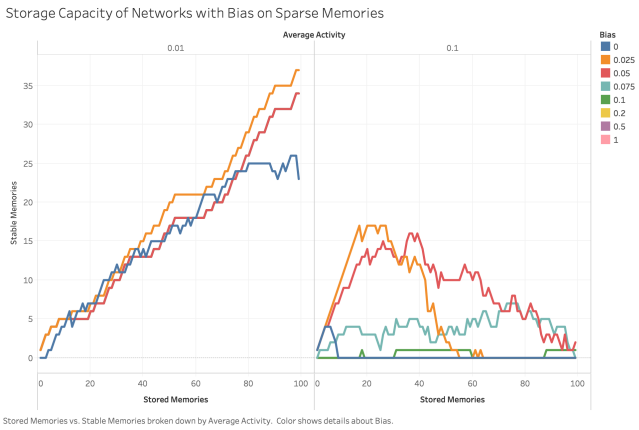


Figure 8: Storage capacity of networks with bias on sparse memories.