Biological Modeling of Neural Networks
Hopfield model of associative memory

Arreguit Jonathan & Bronner Timothée

Professor Wulfram Gerstner
Assistant Mohammadjavad Faraji

Master - Spring 2015

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Contents

1 Introduction

In this project, we implement an online version of the Hopfield network, where storage and recall phases are called randomly according to a probabilistic law. As such, a set of binary patterns is created, and the Hopfield will either apply storage of a random pattern into its memory, or apply recall on a noised pattern of the created pattern set, where the recall will attempt to reconstruct the pattern using its memory through association. Once a noised pattern is recalled, the error between the reconstructed pattern and the original pattern is computed in order to test the Hopfield network's performance. A typical run is illustrated in Figure 1.1:

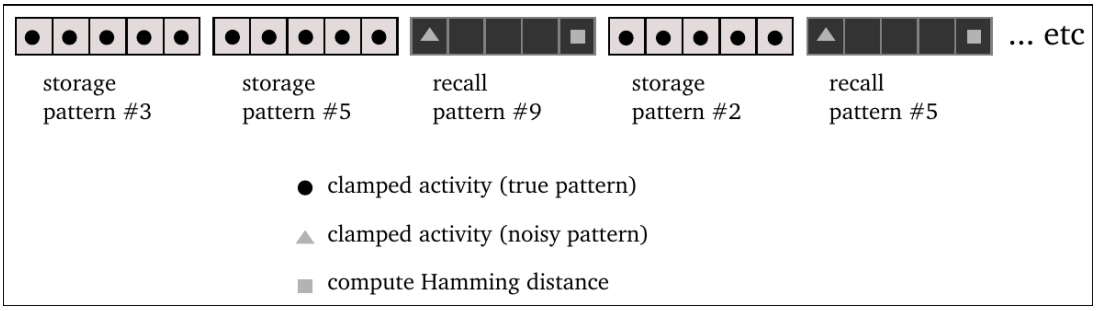


Figure 1.1: Schematic time course of different phases during online learning and recall in the network, as illustrated in the project assignment

For both storage and recall, the synaptic weight w_{ij} are updated according to the following mathematical formula at each step:

$$w_{ij}(t+1) = \lambda w_{ij}(t) + \frac{1}{N} S_i(t) S_j(t) \tag{1.1}$$

where S represents a binary pattern of size N that is being either stored or recalled, and λ is a decay factor which allows the memory of the Hopfield network to forget the memorised patterns with time when $\lambda < 1$.

Next, in the case of a recall, the network state is updated according to the following equation:

$$S_i(t+1) = \text{sign}\left(\sum_{j=1}^{N} w_{ij} S_j(t)\right) \tag{1.2}$$

This equation allows to reconstruct a given pattern, typically a noised version of a pattern from the created pattern set. The goal of the Hopfield network is to reconstruct the noised pattern into a similar pattern to the original, which simulates recognition of a previously learnt pattern. The reconstruction error is computed using the Hamming distance between the reconstructed noised pattern and the original pattern with the following equation:

$$d(\xi, \zeta) = \frac{1}{2}\left(1 - \frac{\xi \cdot \zeta}{N}\right) \tag{1.3}$$

where $\xi \cdot \zeta$ denotes the inner product between the reconstructed pattern ξ and the original pattern ζ.

The project was separated into different exercises, each in which different aspects of this Hopfield network implementation were tested. For each of these exercise, various parameters were chosen accordingly. These parameters include:

- N: Network size or number of fully interconnected units
- Z: Number of phases (storages and recalls)
- c: Number of time steps for each phase
- p_s: Probability of storage
- p_f: Added noise ratio for a pattern to recall, proportion of switched units
- λ: Weight decay factor

As such, the process is run for a total of $Z \cdot c$ time steps. Furthermore, the binary patterns $[-1, +1]$ are of size N and the synaptic weight matrix is of size $N \times N$. The algorithm was implemented in Python using various packages; Numpy [1], Matplotlib [2], Scipy [3], Random, Time, Copy and Image. In particular, the Numpy package was vital in this project to obtain faster computation time, allowing to use a higher resolution. In this report, we present each of the experiments used to test the Hopfield network's performance and discuss the results obtained through illustrated graphs when modifying the parameters.

2 Architecture of the code

The code was heavily based on the Hopfield package distributed in class from week 5 (16 March 2015). The code was separated into different scripts to ease the implementation and understanding.

- *Hopfield.py:* Script that executes the computation for the Hopfield network
- *Project.py:* Script where the parameters are chosen and the exercise processes are called
- *ExerciseN:* Scripts that execute the Hopfield network for each exercise N (where N ranges from 1 to 4).

Figure 2.1 shows how each of these scripts are called. The code for each of these files can be found in appendix A.2. The possibility to call graphical representation of the recall of patterns and to check the progression of the computation were also implemented and can be activated through the use of flags in the *Project.py* file. The project is launched by executing the *Project.py* script.

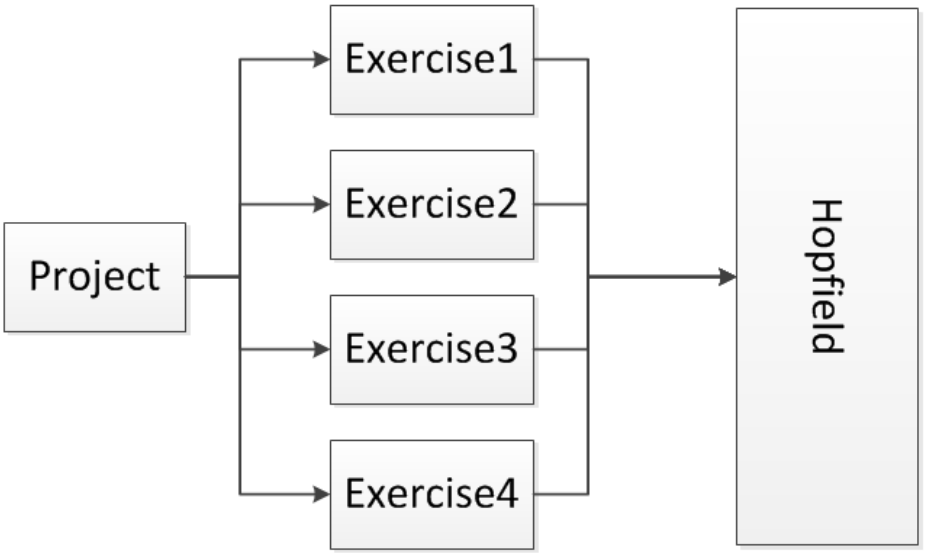


Figure 2.1: Architecture of the code files execution

3 Exercise 1

The first task was to determine the maximum dictionary size p_{max} the network could handle without the recall error exceeding 5%. The Hopfield network was tested for the following parameters.

- $N = 100$ Network size
- $Z = 1000$ Number of phases (storages and recalls)
- $c = 5$ Number of time steps for each phase
- $p_s = 0.8$ Probability of storage
- $p_f = 0.1$ Noise ratio for a pattern to recall
- $\lambda = 1.0$ Weight decay factor
- $p = 1 \dots 30$ Number of patterns in the pattern set (Dictionary size)
- $K = 100$ Number of times the experiment was repeated for cross-validation

More precisely, the Hopfield network is first launched by creating p patterns. At each phase, the algorithm will have $p_s = 80\%$ probability of performing a storage, otherwise it will perform a recall. In case of a storage, a pattern from the pattern set is chosen randomly and stored in the memory using equation 1.1, for which this storage is applied $c = 5$ times for this same pattern. In the case of a recall, a pattern is chosen randomly from the pattern set and $p_f = 10\%$ noise is added to this pattern for the purpose of checking whether the Hopfield network can successfully "recognise" it. To do so, equations 1.1 and 1.2 are called one after the other $c = 5$ times for each phase. After these 5 repeats, i.e. at the end of the recall phase, the error is computed using the Hamming distance, as presented in equation 1.3. The total number of storages and recalls combined is equal to $Z = 1000$ phases, which is launched for each p, the number of patterns in the pattern set, for a total of $K = 100$ times.

The first results obtained are illustrated in Figure 3.1. This figure shows the mean and the standard deviation, relatively to the $K = 100$ experiment repetitions. The 5% error limit represents the maximum desired error, and the original error represents the $p_f = 10\%$ added noise to the original pattern. One can see that the mean error is inferior to 5% up to 11 patterns, and that the noised pattern will even start deteriorating as the recall error $>$ noised error for a $p > 16$ number of patterns. This increased error is caused by the fact that there are more patterns stored in the network's memory, i.e. the synaptic weights. As such, the memory is saturated with the number of patterns to "remember", and will start associating patterns to recall with other patterns than the original one.

It is also interesting to understand how the error evolves over the number of phases. Figure 3.2 shows that the reconstruction error is usually higher at the beginning of the 1000 steps, i.e. for the recalls that are called right after the network was created. This result is to be expected, as it is a consequence of the randomness of the pattern selection for the recall phase. The algorithm could try to recall a pattern which may have not been been stored previously. As the simulation continues, the probability that the recall tests a pattern that has been previously stored increases, therefore the error average decreases and will tend to converge to a final value. It is also important to note that even though the mean error is superior for a higher number of patterns, there can still be some cases where the recall will result in a perfect reconstruction of the noised pattern during recall.

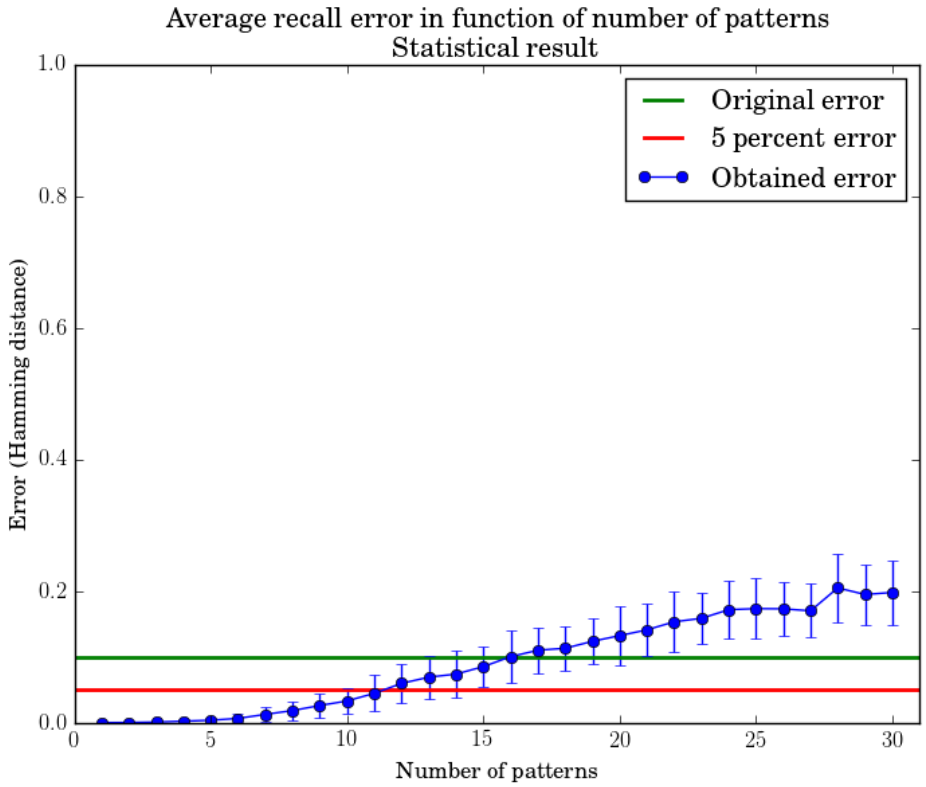


Figure 3.1: Error in function of the number of patterns with STD bars for 100 experiment repetitions

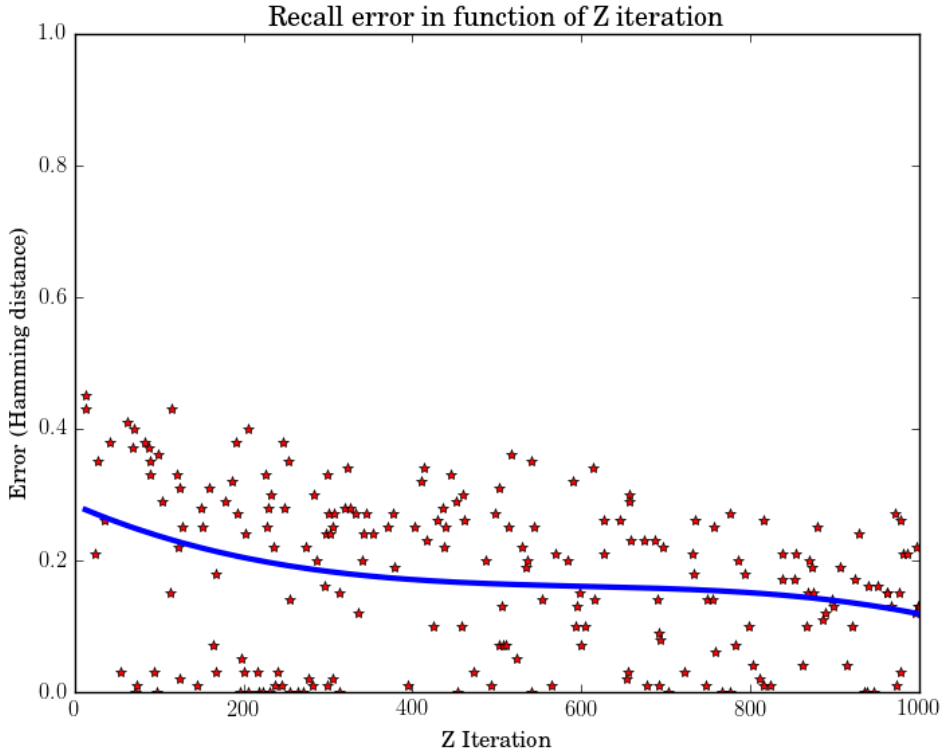


Figure 3.2: Error over number of phases for $p = 30$ patterns

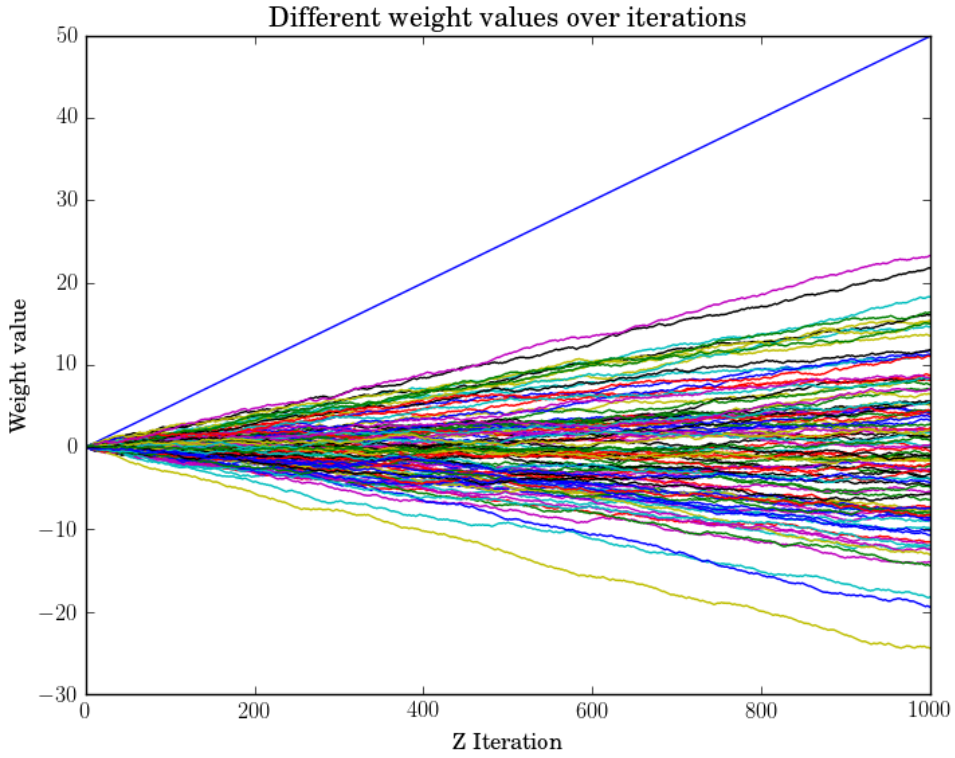


Figure 3.3: Plot of the evolution of certain elements of the matrix in function of the number of phases

Another interesting element to study is the synaptic weight matrix, which simulates the memory of the Hopfield network. In Figure 3.3, different values of the synaptic weight matrix are plotted in function of the number of phases. This figure shows that certain values of the matrix will diverge with each storage and recall. This is caused by the fact that the weight decay factor $\lambda = 1.0$, which means that if the average of an element of same index for each pattern from the set of patterns is biased towards a certain value (± 1), the element of the synaptic weights matrix will diverge in this case. Furthermore, certain elements of the matrix diverge linearly (top straight blue line), these elements are those of the diagonal of the synaptic weights matrix. This is caused by equation 1.1, which will increment the matrix element for $i = j$ (the diagonal) when applying a decay of $\lambda = 1.0$. In appendix A.1, we study and propose a way in order to optimise the values of the diagonal of this matrix in order to obtain better performance from the Hopfield model.

4 Exercise 2

In this exercise, the effect of the dimension N on the maximum dictionary size p_{max} is studied. As such, the following parameters were used:

- $N = 100 \dots 1000$ Network size
- $Z = 1000$ Number of phases (storages and recalls)
- $c = 5$ Number of time steps for each phase
- $p_s = 0.8$ Probability of storage
- $p_f = 0.1$ Noise ratio for a pattern to recall
- $\lambda = 1.0$ Weight decay factor
- $P_{init} = 27$ The initial number of patterns p for which to start the binary search

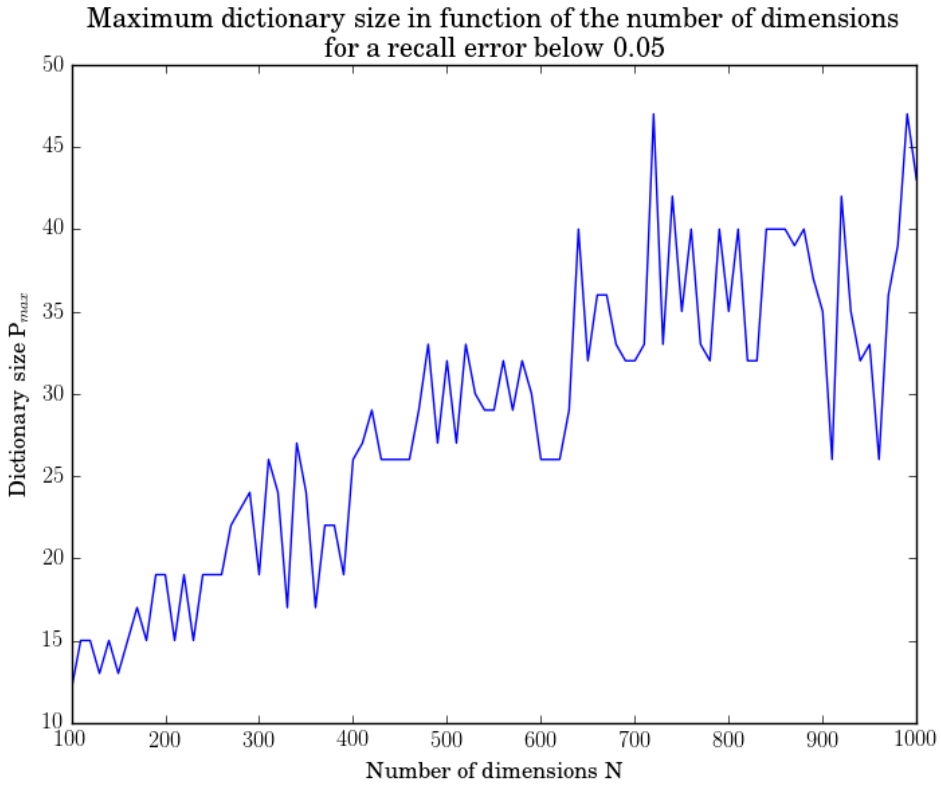


Figure 4.1: Maximum dictionary size for an error below 5% in function of the network size, from 100 to 1000 with steps of 10. The starting point for the binary search is 27 patterns.

For each network size N, ranging from 100 to 1000 with a step of 10, a binary search was implemented in order to find the maximum dictionary size P_{max} for which the mean error over $Z = 1000$ phases is below 5%. To do this, the process is launched for an initial dictionary size $P_{init} = 27$. If the mean error is below 5% after $Z = 1000$ phases, the dictionary size is doubled. Else, if the error is above 5%, the dictionary size is halved (Taking into account that the dictionary size is a natural number). Once the dictionary size is known to be bounded between two values, and is tested for a dictionary size between these two values, the next dictionary size will either be calculated from equation 4.1 if the error is below 5%, or from equation 4.2 if above 5%.

Error smaller than 5%:

$$p = \frac{p_{max} + p}{2} \tag{4.1}$$

Error bigger than 5%:

$$p = \frac{p_{min} + p}{2} \tag{4.2}$$

where p_{min} and p_{max} are the known minimum and maximum boundaries.

The results for this exercise are illustrated in Figure 4.1. As the size of the network increase, it seems that its memory and resilience will increase. However, it is difficult to predict in what way the dictionary size scales with the network size N, as the binary search result is very noise sensitive. It seems none the less that the increase in the maximum dictionary size P_{max} is linearly proportional with the size of the network for these results. It is also important to keep in mind that the computational time for a bigger network size N is significantly higher.

5 Exercise 3

In this exercise, the dictionary size P is fixed to 100, but a sub-dictionary size of $m = 5$ patterns is shifted periodically along the dictionary size, replacing the tested patterns by a new set. This window and period are calculated according to the following:

$$\text{Sub-dictionary} = [round(f(i)) \ mod \ p, \ round(f(i) + m - 1) \ mod \ p] \tag{5.1}$$

with $f(i) = i/20$. In this case, it becomes interesting to implement a weight decay factor $\lambda < 1$ in order to allow the memory to "forget" previously stored patterns along with time. This allows the recall to avoid being misled by previously memorised patterns that would be known to not be tested in the current sub-dictionary. The following parameters were implemented to test the performance:

- $N = 100$ Network size
- $p = 100$ Dictionary size
- $Z = 1000$ Number of phases (storages and recalls)
- $c = 5$ Number of time steps for each phase
- $p_s = 0.8$ Probability of storage
- $p_f = 0.1$ Noise ratio for a pattern to recall
- $\lambda = 0 \ldots 1.0$ Weight decay factor
- $m = 5$ Sub-dictionary size

The value of the weight decay factor has an effect on the rate at which the memory forgets, a value of $\lambda = 1$ means that the memory never forgets and a value of $\lambda = 0$ means that it will forget all at the next time step. Using a decay $\lambda < 1$ will basically allow the simulation to forget previous patterns and only focus on the patterns included in the sub-dictionary. On the other hand, using a $\lambda = 1$ is equivalent to launching the process with a dictionary size of $p = 100$ patterns under the same conditions to exercise 1.

The results obtained for this exercise are illustrated in Figure 5.1. These results show that a decay smaller than 0.55 will lead to the recall not having any effect on the reconstruction, similarly to a total memory loss, as the error stays at 10% which represents the added noise p_f.

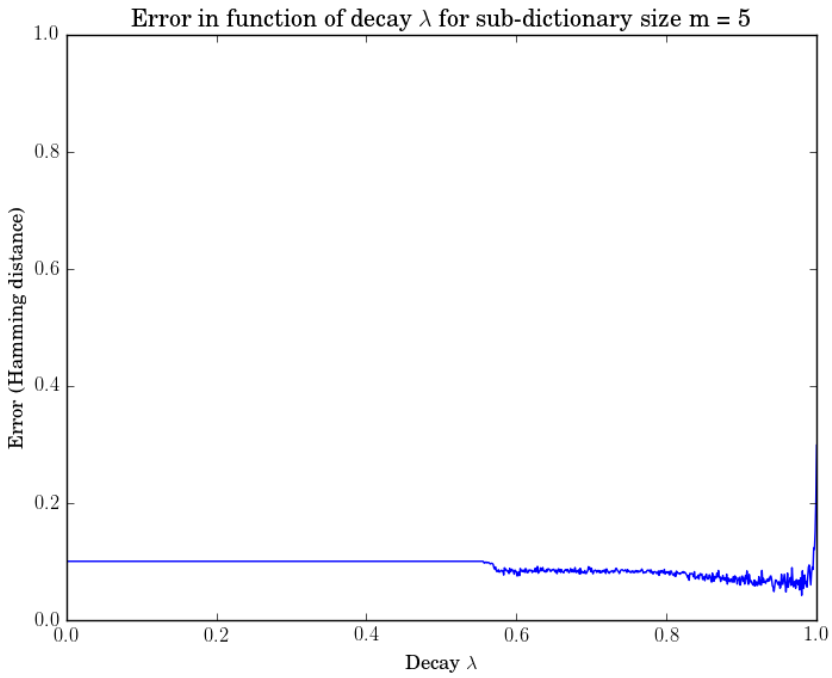


Figure 5.1: Decay from 0 to 1, with a resolution of 0.001

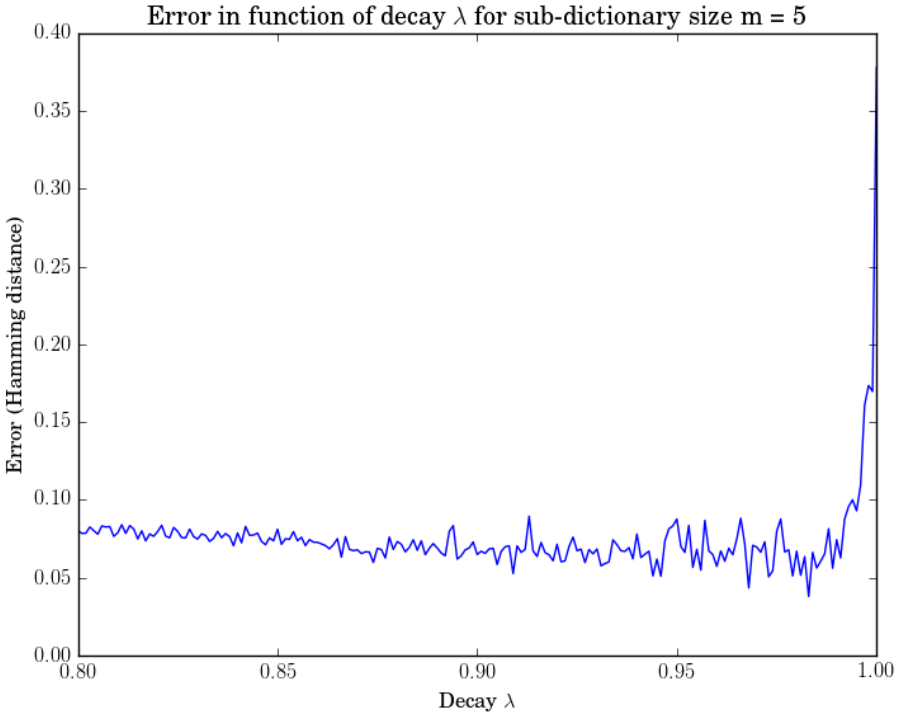


Figure 5.2: Decay from 0.8 to 1.0, with a resolution of 0.001

Figure 5.2 shows in detail the zone where the optimal reconstruction occurs. It can be seen that as the decay is getting closer to 1 the signal gets noisier, which leads to a local minimum of the error for a decay at about 0.98. This end up being almost equivalent to working on a dictionary size of 5 instead of 100. It could nonetheless be advised to chose a smaller decay, such as 0.9, as it avoids being too close to the values for which the error shoots up.

6 Exercise 4

The goal of the final exercise was to investigate the relationship between the optimal decay rate and the size of the subdictionary, or in other words analyse the function: $Error(\lambda, m)$ where λ and m stand respectively for the decay rate and the subdictionary size. The following parameters were used:

- $N = 100$ Network size
- $p = 100$ Dictionary size
- $Z = 1000$ Number of phases (storages and recalls)
- $c = 5$ Number of time steps for each phase
- $p_s = 0.8$ Probability of storage
- $p_f = 0.1$ Noise ratio for a pattern to recall
- $\lambda = 0 \ldots 1.0$ Weight decay factor
- $m = 2...15$ Sub-dictionary size

This 2D function in the shape of a heatmap is illustrated in Figure 6.1. Again, it can be seen that until the weight decay λ reaches a value of approximatively 0.55, the Hopfield process does not modify the noised patterns. On the other hand, there appears to be again a fast increase of the error when the decay approaches $\lambda = 1$.

In order to investigate in more detail the optimal values of the $Error(\lambda, m)$, a zoom has been done between 0.8 and 0.99, illustrated in Figure 6.2. The weight decay value $\lambda = 1$ has been purposefully removed in order to show a greater contrast. It can be seen that the bigger the dictionary size, the smaller the optimal weight decay to obtain the lowest error. This is possibly caused by the fact that there are more different patterns stored in the memory, meaning that it is necessary for the weight decay value λ to be lower to increase "forgetfulness" and focus even more on the sub-dictionary size, even though this increase in forgetfulness could still slightly increase the reconstruction error.

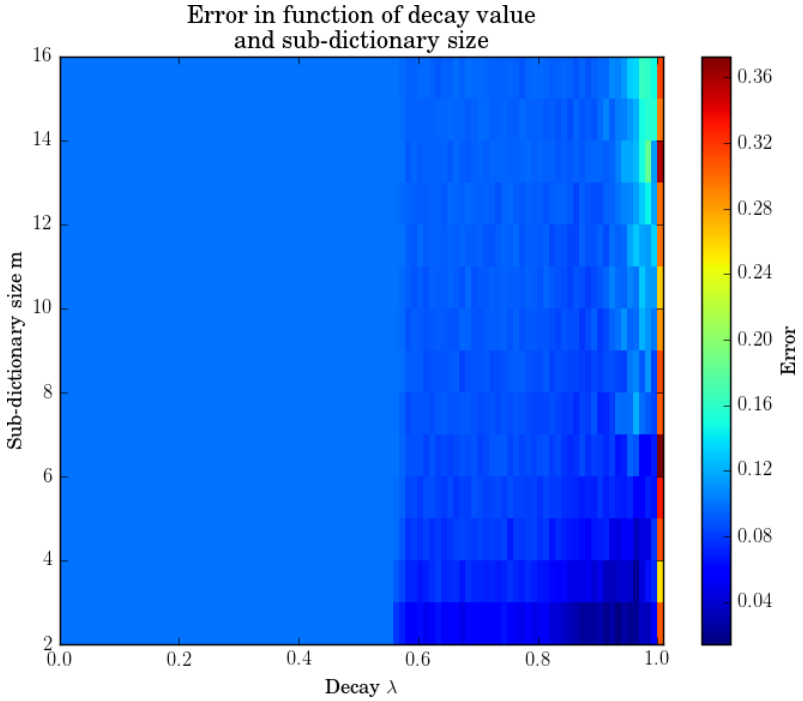


Figure 6.1: Heatmap result for a decay resolution of 0.01

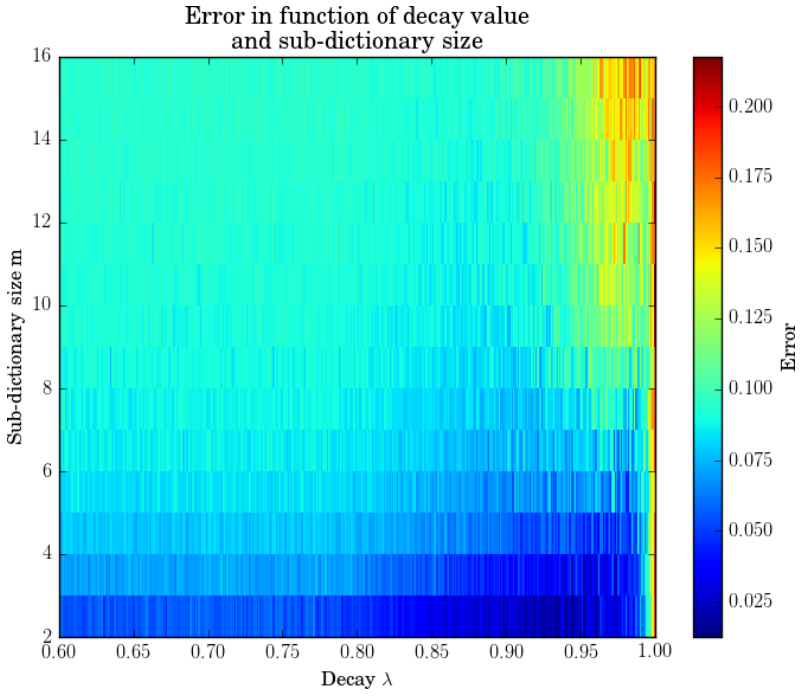


Figure 6.2: Heatmap zoom for a decay resolution of 0.001

7 Conclusion

In conclusion, we successfully implemented a Hopfield network, and tested it under different conditions to measure it's performance when recalling patterns from a set dictionary with added noise. We started off by studying the effect of the number of parameters on the recall error, where we showed that the lower the number of patterns in the dictionary, the better the recall. Having too many patterns could even have a reverse effect, where the recall error could end up being worse than the error from the added noise. Next in exercise 2, we studied the effect of the network size on the maximum number of patterns that can be stored in the dictionary for which the error is smaller than 5%. The results showed that the maximum dictionary size would indeed increase as the network size increased, at the cost of a higher computational cost. Exercise 3 showed the role of the decay factor, where a sub-dictionary window would be shifted along the total dictionary size. We saw that there would be optimal values for which the decay would minimise the reconstruction error. Finally, exercise 4 allowed us to study the reconstruction error when both modifying the weight decay factor and the sub-dictionary size, where we found out how the error reacts in function of both.

On a final note, this project allowed us to study the Hopfield network and test the effect of its different parameters and how they influenced the recall reconstruction error. In particular, a weight decay factor was implemented in the objective of studying the effect of "forgetfulness", where we saw that the ability to forget can actually increase the recognition of new learnt patterns. The goal of Hopfield networks is to propose a model on how biological brains are capable of associating present sensorial information with previous memories. Even though computer results can seem to be quite computationally slow to calculate and the results seem to show that not many patterns can be stored compared to a real brain, it is important to remember that brains work on a massively parallel basis with equivalently far more synaptic weights, although without fully interconnected units. The Hopfield network could still very well be a good mathematical approximation to explain this process for the low-level functionality of the brain.

References

[1] Stéfan van der Walt, S. Chris Colbert, and Gaël Varoquaux. The numpy array: A structure for efficient numerical computation. `http://www.numpy.org/`, Computing in Science & Engineering, 13, 22-30 (2011).

[2] John D. Hunter. Matplotlib: A 2d graphics environment,. `http://matplotlib.org/`, Computing in Science & Engineering, 9, 90-95 (2007).

[3] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python. `http://www.scipy.org/`, 2001–.

A Appendix

A.1 Addition of Hopfield confidence threshold and learning phase

At each step of the recall, the two following equations are called:

The synaptic weights w_{ij} update:

$$w_{ij}(t+1) = \lambda w_{ij}(t) + \frac{1}{N} S_i(t) S_j(t) \tag{A.1}$$

Network state update:

$$S_i(t+1) = \text{sign}\left(\sum_{j=1}^{N} w_{ij} S_j(t) \right) \tag{A.2}$$

Using equation A.1, the weights of the synaptic weights, which can be associated to the memory of the network, are updated according to the pattern $S(t)$ that is presented at a given time. The result of $S_i(t)S_j(t)$ represents the relations that $S_i(t)$ has to $S_j(t)$, with a result $+1$ representing a same value for both, and -1 representing an inverted value. As such, w_{ij} holds the information saying if index i and j of the previous patterns S were usually the same or inverted, with a high absolute value meaning it was more often in the case for one of the two possible results of the product. It is important to note that using equation A.1 holds two truths:

- w_{ij} converges to a maximum possible positive value if $\lambda < 1$ or diverges to infinity if λ=1 for $i = j$, being the diagonal of the matrix, which also holds a same value for
- $w_{ij} = w_{ji}$, the weight matrix is symmetric

Next, when recalling a previous pattern using equation A.2, a new studied pattern is compared to the previous memories stored in the weight matrix. It is interesting to note that this equation will try to reconstruct the pattern in accordance to a similar previously stored pattern. To do so, the new studied keeps its value for a given index i if the the equation result yields a positive number, and is inverted if the equation yields a negative value. Knowing that w_{ij} does not hold any information about the previous patterns, but is only influenced by the decay factor λ and the number of updates that have passed, it could be interesting for this value to be modified "artificially".

A first approach could be to set w_{ij} for all $i = j$, being the diagonal of the weight matrix, to zero. This approach is actually proposed by many papers, but was found to yield worse results than when keeping the original equations A.1 and A.2, where the error would grow more rapidly. Setting the diagonal to high values also yields worsened results, the pattern at different indexes would not be changed by equation A.2 as the w_{ij} for $i = j$ would be too big to obtain a negative result of the sum using the other weights w_{ij} for $i \neq j$.

It was concluded that an optimal value could be found for this diagonal. The first approach was to set this values of the diagonal to a constant value, which yielded improved results indeed. However, it was also noticed that setting the values of the diagonal could yield even better results by changing the values of the diagonal of the weight matrix at each update step. As such, the following equation was implemented in order to tune the diagonal of the weight matrix this approach:

$$w_{diagonal} = \theta \frac{1}{N} \sum_{j \neq i} |w_{ij}| \tag{A.3}$$

Where θ is a confidence factor set between 0 and 1. This factor allows to tune how confident the update step needs to be for a given index. Using this implementation, optimal results were obtained for a confidence θ of 15%, where the memory would indeed obtain error $< 5\%$ for up to 20 patterns for the parameters of exercise 1, illustrated in Figure A.1a.

Another approach would be to set each member of the diagonal w_{ii} to a value in accordance with the other weights w_{ij} of its same row, given by:

$$w_{ii} = \theta \sum_{j \neq i} |w_{ij}| \tag{A.4}$$

This approach has not been tested at the time of this report, but could possibly yield even better results as each diagonal member is tuned according to it's row. It was also not tested if the extra processing steps for a numerical process of the Hopfield Network using this approach would yield as good or improved results than when simply augmenting the dimension of the patterns.

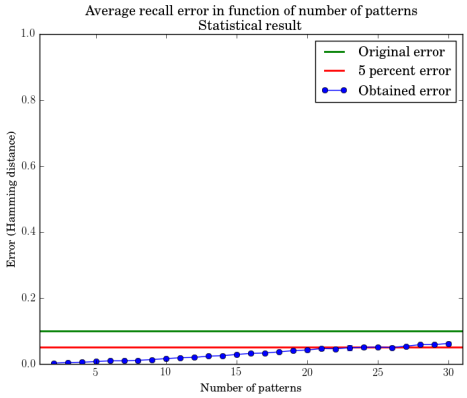
Finally, it was also noticed that many errors would be made at the launch of the Hopfield network, as the weight matrix would have not have yet stored certain parameters. As such, a learning threshold could be added to the diagonal using:

$$w_{ii\,Learning} = \Lambda e^{-\tau t} \tag{A.5}$$

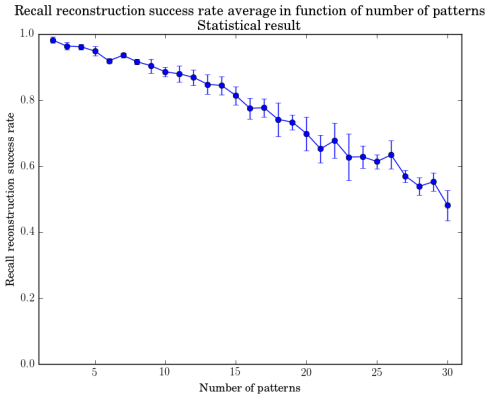
where Λ is a learning threshold and τ allows to tune the decrease over time of this additional confidence requirement. As such, the final implementation of the confidence diagonal can be implemented using:

$$w_{ii} = \theta \sum_{j \neq i} |w_{ij}| + \Lambda e^{-\tau t} \tag{A.6}$$

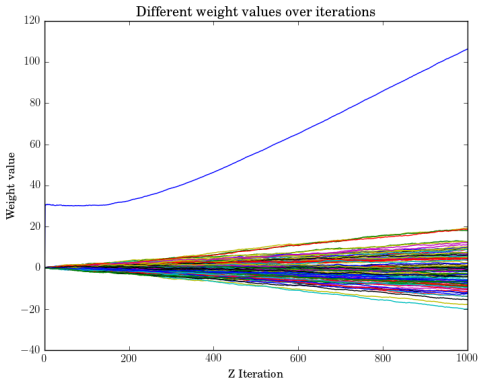
The implementation of this additional work can be seen in lines 97-99 of the *Hopfield.py* script and can be used when decommented. The results for parameters similar to exercise 1 are ilustrated in Figure A.1.



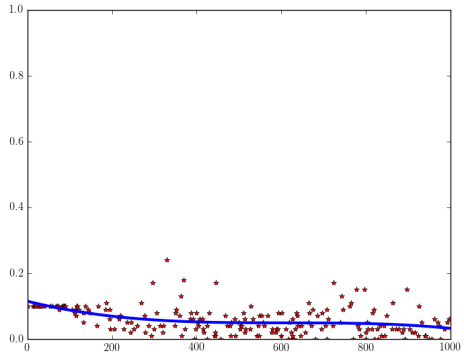
(a) Error



(b) Recall success rate (Error $< 5\%$)



(c) Weights



(d) Hamming distance for $p = 30$ patterns

Figure A.1: Results obtained for $K = 5$, $N = 100$, $Z = 1000$, $\lambda = 1$, $c = 5$, $p_s = 0.8$ and $p_f = 0.1$

A.2 Code

A.2.1 Hopfield.py

```
1  import Image
2  from copy import copy
3  from time import sleep
4  from pylab import *
5  import random as rand
6
7  class hopfield_network:
8      def __init__(self,N,c=5,plot=False):
9
10         """
11         DEFINITION
12         Initialization of the class
13
14         INPUT
15         N: size of the network
16         c: Number of repeats every step (storage and recall)
17         plot: Activate recall plots (True or False)
18         """
19
20         self.N = N
21         self.c = c
22         self.FLAG_plot = plot
23
24     def make_pattern(self,P,ratio):
25
26         """
27         DEFINITION
28         Creates and stores patterns
29
30         INPUT
31         P: number of patterns
32         ratio: probability of a pixel being 1 instead of -1
33         """
34
35         self.pattern = -ones((P,self.N),int)
36         idx = int(ratio*self.N)
37         for i in range(P):
38             for j in range(self.N):
39                 if rand.randint(0,1000) < ratio*1000:
40                     self.pattern[i,j] = 1
41
42
43         self.weight = zeros((self.N,self.N))
44         self.distance_sum = 0
45         self.recognition_success = 0
46         self.recalls = 0
47         self.update_count = 0
48
49     def count_patterns(self):
50
51         """
52         DEFINITION
53         Counts the number of patterns
54
55         OUTPUT
```

```
56          Number of patterns in the class
57          """
58
59          return len(self.pattern)
60
61      def noise_pattern(self,mu,P_f):
62
63          """
64          DEFINITION
65          Adds noise to patterns to recall
66
67          INPUT
68          mu: pattern stored in X, X is then noised by P_f ( Value between 0
                and count_patterns()-1 )
69          P_f: ratio of pixels whose value has been inverted
70          """
71
72          self.x = copy(self.pattern[mu])
73          flip = permutation(arange(self.N))
74          idx = int(self.N*P_f)
75          self.x[flip[0:idx]] *= -1
76
77      def update_weight(self,mu,decay):
78
79          """
80          DEFINITION
81          Update of synaptic weights w_ij
82
83          INPUT
84          mu: For storage, a pattern mu is sent. Else if no mu is sent, the
                weights are updated according to the noised pattern X
85          decay: Weight decay factor ranging from 0 - 1. 1 represents an
                equal memory over all time for all weight updates, and 0
                represents no memory of previous storage.
86          """
87
88          self.weight = decay*self.weight
89          C = 1./self.N
90
91          if mu is not None: # Storage
92              self.weight += C*np.multiply(self.pattern[mu][np.newaxis,:].T,
                    self.pattern[mu])
93          else: # Recall
94              self.weight += C*np.multiply(self.x[np.newaxis,:].T,self.x)
95
96          #Confidence and learning phase
97          #np.fill_diagonal(self.weight, 0)
98          #weight_sum = np.sum(np.abs(self.weight))
99          #np.fill_diagonal(self.weight, 30*(np.exp(-1E-3*self.update_count))
                + 0.15*weight_sum/self.N) # Means the algorithm needs to be at
                least 15% confident that the pixel needs to be inverted
100
101         #Theory
102         #The weights of the memory matrix represent the confidence for an
                Si should be the same (+) or the inverse (-) of Sj depending on
                previous knowledge. As such, a minimum confidence threshold can
                be required for a change to happen by modifying the diagonal,
                which diverges anyway for the given equation in the assignment.
```

```
            This threshold should be calculated using: (Threshold [0-1, 0
            being not confident at all and 1 being absolutely ]) * (The
            total confidence (sum) of the row of the weights matrix for the
            Si term). Finally, the first term is added as confidence needs
            to be higher or mistakes can be made as the memory is just born.
103
104             self.update_count += 1
105
106         def update_state(self):
107
108             """
109             DEFINITION
110             Network state update for each step of the recall with X
111             """
112
113             self.x = np.sign(np.dot(self.weight,self.x))
114
115         def overlap(self,mu):
116
117             """
118             DEFINITION
119             computes the overlap of the test pattern with pattern nb mu
120
121             INPUT
122             mu: the index of the pattern to compare with the test pattern
123             """
124
125             return 1./self.N*np.dot(self.pattern[mu],self.x)
126
127         def grid(self,mu=None):
128
129             """
130             DEFINITION
131             reshape an array of length N to a matrix MxM with M = sqrt((N-1)+1)
132
133             INPUT
134             mu: None -> reshape the test pattern x
135                 an integer i < P -> reshape pattern nb i
136             """
137
138             M = int(math.sqrt(self.N-1)+1) # MxM pixel image for visualization
139
140             if mu is not None:
141                 graphic = zeros(M*M)
142                 graphic[0:self.N] = self.pattern[mu]
143                 x_grid = reshape(graphic,(M,M))
144             else:
145                 graphic = zeros(M*M)
146                 graphic[0:self.N] = self.x
147                 x_grid = reshape(graphic,(M,M))
148             return x_grid
149
150         def hamming_distance(self, mu):
151
152             """
153             DEFINITION
154             Computes the Hamming distance between patterns X and mu
155
```

```
156         INPUT
157         mu: the index of the pattern to compare with the test pattern
158         """
159
160         return 0.5*(1-self.overlap(mu))
161
162     def recall(self,mu,P_f,decay):
163
164         """
165         DEFINITION
166         runs the recall and plots the results (if FLAG_plot = True during
                initialisation)
167
168         INPUT
169         mu: pattern number to use as test pattern
170         P_f: ratio of flipped pixels
171                     ex. for pattern nb 5 with 5% flipped pixels use run(mu
                        =5,P_f=0.05)
172         """
173
174         try:
175             self.pattern[mu]
176         except:
177             raise IndexError, 'pattern index too high'
178
179         # set the initial state of the net
180         self.noise_pattern(mu=mu,P_f=P_f)
181         t = [0]
182         overlap = [self.overlap(mu)]
183
184         if self.FLAG_plot:
185             self.plot_recall(mu=mu,t=t,overlap=overlap)
186
187         for i in range(self.c):
188
189             # run a step
190             self.update_weight(decay=decay,mu=None)
191             self.update_state()
192             t.append(i+1)
193             overlap.append(self.overlap(mu))
194
195             # Visualisation if FLAG_plot = True
196             if self.FLAG_plot:
197                 ## update the plotted data
198                 self.g1.set_data(self.grid())
199                 self.g2.set_data(t,overlap)
200
201                 ## update the figure so that we see the changes
202                 draw()
203                 show(block=False)
204
205             # sleep(0.5) # Can be used to slow down the Visualisation
206
207         self.distance = self.hamming_distance(mu)
208
209         self.distance_sum += self.distance
210
211         if self.distance < 0.05:
```

```
212             # Counts the number of times the pattern was recovered
                    according to a Hamming distance < 0.05. This is only used to
                     analyse the impact of modifying the diagonal of the weights
                     matrix, mentionned in the appendix of the report
213
214             self.recognition_success += 1
215
216         # Count number of patterns recalled to later compute the error
                average
217         self.recalls += 1
218
219     def plot_recall(self,mu,t,overlap):
220
221         """
222         DEFINITION
223         Visualisation of the recall results
224
225         INPUT
226         mu: Original pattern that X is based on
227         t: number of recall update X has gone through
228         overlap: The overlap between pattern mu and X
229         """
230
231         # prepare the figure
232         fig_recall = figure('Recall')
233         clf()
234
235         # plot the current network state
236         subplot(221)
237         self.g1 = imshow(self.grid(),**plot_dic)# we keep a handle to the
                image
238         axis('off')
239         title('Noised pattern to recall')
240
241         # plot the target pattern
242         subplot(222)
243         imshow(self.grid(mu=mu),**plot_dic)
244         axis('off')
245         title('Original pattern %i'%mu)
246
247         # plot the time course of the overlap
248         subplot(212)
249         self.g2, = plot(t,overlap,'k',lw=2) # we keep a handle to the curve
250         axis([0,self.repeats,-1,1])
251         xlabel('Time step')
252         ylabel('Overlap to original')
253
254         # This forces pylab to update and show the figure
255         draw()
256         #show(block=False) # Can be required depending on users python
                version
```

A.2.2 Project.py

```
1  import hopfield
2  reload(hopfield)
3  # import multiprocessing
4  import random as rand
5  import numpy as np
6  import matplotlib.pyplot as plt
7  import pylab as pl
8  import time
9
10 FLAG_progress = False # Show process progress (True or False)
11 FLAG_plot = False
12
13 plt.rc('text', usetex=True)
14 plt.rc('font', family='serif')
15
16 plt.close('all')
17
18 # ---------------------------------------
19 # ---------------------------------------
20 # ---------------------------------------
21
22 start = time.time()
23
24 # Exercise 1
25
26 print '\nEXERCISE 1\n==========\n'
27
28 # Run
29 K = 100 # Number of times the algorithm is run for cross-validations
30
31 # Network size
32 N = 100 # size of the network, i.e. if N=10 it will consists of 10x10
      pixels
33
34 # Random patterns
35 P = range(1,30+1,1) # Total number of patterns that will be stored
36 ratio = 0.5 # probability of a pixel being 1 instead of -1
37
38 # Updates
39 decay = 1.0
40
41 # Steps
42 Z = 1000 # Number of storage/recall iterations
43 c = 5
44
45 # Storage and recalls
46 P_s = 0.8 # Probability ps for storage
47 P_f = 0.1 # ratio of flipped pixels
48
49 execfile('Exercises/Exercise1.py')
50
51 # ---------------------------------------
52 # ---------------------------------------
53 # ---------------------------------------
54
55 # Exercise 2
```

```
56
57 print '\nEXERCISE 2\n==========\n'
58
59 # Network size
60 N = range(100,1000+1,10) # size of the network, i.e. if N=10 it will
       consists of 10x10 pixels
61
62 # Random patterns
63 ratio = 0.5 # probability of a pixel being 1 instead of -1
64
65 # Updates
66 decay = 1.0
67
68 # Steps
69 Z = 1000 # Number of storage/recall iterations
70 c = 5
71
72 # Storage and recalls
73 P_s = 0.8 # Probability ps for storage
74 P_f = 0.1 # ratio of flipped pixels
75
76 # Exercise parameters
77 error_max = 0.05
78 P_init = 27
79
80 execfile('Exercises/Exercise2.py')
81
82 #------------------------------------------
83 #------------------------------------------
84 #------------------------------------------
85
86 # Exercise 3
87
88 print '\nEXERCISE 3\n==========\n'
89
90 # Network size
91 N = 100 # size of the network, i.e. if N=10 it will consists of 10x10
       pixels
92
93 # Random patterns
94 p = 100 # Total number of patterns that will be stored
95 ratio = 0.5 # probability of a pixel being 1 instead of -1
96
97 # Updates
98 Decay = range(0,1000+1,1) # Decay multiplied by resolution
99 resolution = 0.001 # Multiplied to the values of Decay
100
101 # Steps
102 Z = 1000 # Number of storage/recall iterations
103 c = 5
104
105 # Storage and recalls
106 P_s = 0.8 # Probability ps for storage
107 P_f = 0.1 # ratio of flipped pixels
108
109 # Exercise parameters
110 T_window = 20
111 m = 5
```

```
112
113 execfile('Exercises/Exercise3.py')
114
115 #-----------------------------------------
116 #-----------------------------------------
117 #-----------------------------------------
118
119 # Exercise 4
120
121 print '\nEXERCISE 4\n==========\n'
122
123 # Network size
124 N = 100 # size of the network, i.e. if N=10 it will consists of 10x10
        pixels
125
126 # Random patterns
127 p = 100 # Total number of patterns that will be stored
128 ratio = 0.5 # probability of a pixel being 1 instead of -1
129
130 # Updatesimport hopfield
131 Decay = range(0,1000+1,1) # Decay multiplied by resolution
132 resolution = 0.001 # Multiplied to the values of Decay
133
134 # Steps
135 Z = 1000 # Number of storage/recall iterations
136 c = 5
137
138 # Storage and recalls
139 P_s = 0.8 # Probability ps for storage
140 P_f = 0.1 # ratio of flipped pixels
141
142 # Exercise parameters
143 T_window = 20
144 M = range(2,15+1,1)
145
146 execfile('Exercises/Exercise4.py')
147
148 #-----------------------------------------
149 #-----------------------------------------
150 #-----------------------------------------
151
152 end = time.time()
153 print 'Elapsed time:', end-start, 'seconds'
```

A.2.3 Exercise1.py

```
# Hopfield project
# Exercise 1
# Arreguit Jonathan & Bronner Timothee

# Find the maximum dictionary size p_max the network can handle, without the error exceeding 0.05

from scipy.optimize import curve_fit, minimize, basinhopping

# Results storage
Results_P_errors = np.array([[[None,None] for i in range(len(P))] for j in range(K)])
P_errors = [[None for i in range(len(P))] for j in range(K)] # For plot
P_success = [[None for i in range(len(P))] for j in range(K)]
Mean_P_error = [None for i in range(len(P))]
Std_P_error = [None for i in range(len(P))]
Results = np.zeros((K,len(P)))

# Creation fo a NxN pixel Hopfield network
hn = hopfield.hopfield_network(N=N, c=c, plot=FLAG_plot)

for k in range(K): # Number of iterations
    P_n = 0 #  Counter for number of patterns tested
    for p in P:
        # Create P patterns
        hn.make_pattern(P=p, ratio=ratio)
        print 'ITERATION: ', k+1,'/',K
        print 'NUMBER OF PATTERNS', p
        weight_mem = [] # Storage of the weight matrix over Z*c updates. This is only used to analyse the impact of modifying the diagonal of the weights matrix, mentionned in the appendix of the report
        distance_mem = [] # Storage of the Hamming distance at each recall. This is only used to analyse the impact of modifying the diagonal of the weights matrix, mentionned in the appendix of the report

        for z in range(Z):
            if FLAG_progress and z%(Z/10) == 0:
                print 100.0*z/Z, '%'
                loading = 0
            weight_mem.append(hn.weight)
            if rand.randint(0,1000)/1000.0 < P_s: # Storage
                mu = rand.randint(0,hn.count_patterns()-1)
                for r in range(c):
                    hn.update_weight(mu=mu, decay=decay)
            else: # Recall
                hn.recall(mu=rand.randint(0,hn.count_patterns()-1), P_f=P_f, decay=decay)
                distance_mem.append([z,hn.distance])

        print 'RECALLS: ', hn.recalls
        print 'ERROR AVERAGE; ', 100.0*hn.distance_sum/hn.recalls, '%'
        print 'SUCESS RATE: ', 100.0*hn.recognition_success/hn.recalls, '%'

        # Results storage
```

```
48          Results_P_errors[k][P_n][0] = p
49          Results_P_errors[k][P_n][1] = 1.0*hn.distance_sum/hn.recalls
50          Results[k,P_n] = 1.0*hn.distance_sum/hn.recalls
51
52          # Plot
53          P_errors[k][P_n] = 1.0*hn.distance_sum/hn.recalls
54          P_success[k][P_n] = 1.0*hn.recognition_success/hn.recalls
55          # hn.plot_error_rate(P=P, error_rate=P_errors,K=K)
56
57          P_n += 1
58          print # Next line
59
60  # Statistics
61  Mean_P_error = np.mean(P_errors, axis=0)
62  Std_P_error = np.std(P_errors, axis=0)
63  Mean_P_success = np.mean(P_success, axis=0)
64  Std_P_success = np.std(P_success, axis=0)
65  # ax0.errorbar(p, Mean_P_error, yerr=Std_P_error, fmt='-o')
66
67  # Plot of the Hamming distance for each iteration over the number of
        patterns
68  plt.figure('Exercise 1 a Distance') # fig_error_rate =
69  plt.title('Recall error in function of number of patterns\nobtained from %i
        iterations'%K)
70  plot_error_rate = [None]*(K)
71  for k in range(K):
72      plt.plot(Results_P_errors[k,:,0],Results_P_errors[k,:,1])
73  plt.axis([min(P),max(P),0,1])
74  plt.xlabel('Number of patterns')
75  plt.ylabel('Error (Hamming distance)')
76  plt.show(block=False)
77
78  # Plot of the mean of the Hamming distance for K iterations over the number
        of patterns
79  plt.figure('Exercise 1b Error')
80  plt.title('Average recall error in function of number of patterns\
        nStatistical result')
81  plt.plot([min(P)-1,max(P)+1],[0.1,0.1],'g',linewidth=2)
82  plt.plot([min(P)-1,max(P)+1],[0.05,0.05],'r',linewidth=2)
83  plt.errorbar(P, Mean_P_error, yerr=Std_P_error, fmt='-o')
84  plt.axis([min(P)-1,max(P)+1,0,1])
85  plt.xlabel('Number of patterns')
86  plt.ylabel('Error (Hamming distance)')
87  plt.legend(['Original error','5 percent error','Obtained error'])
88  plt.show(block=False)
89
90  # Plot of the reconstruction success rate when the recall gives a final
        Hamming distance < 0.05 for K iterations
91  # This is only used to analyse the impact of modifying the diagonal of the
        weights matrix, mentionned in the appendix of the report
92  plt.figure('Exercise 1c Reconstruction')
93  plt.title('Recall reconstruction rate in function of number of patterns\
        nobtained from %i iterations'%K)
94  plot_error_rate = [None]*(K)
95  for k in range(K):
96      plt.plot(Results_P_errors[k,:,0],P_success[k][:])
97  plt.axis([min(P),max(P),0,1])
98  plt.xlabel('Number of patterns')
```

```
 99 plt.ylabel('Recall reconstruction success rate')
100 plt.show(block=False)
101
102 # Plot of the mean of the reconstruction success rate when the recall gives
        a final Hamming distance < 0.05
103 # This is only used to analyse the impact of modifying the diagonal of the
        weights matrix, mentionned in the appendix of the report
104 plt.figure('Exercise 1d Reconstruction')
105 plt.title('Recall reconstruction success rate average in function of number
        of patterns\nStatistical result')
106 plt.errorbar(P, Mean_P_success, yerr=Std_P_success, fmt='-o')
107 plt.axis([min(P)-1,max(P)+1,0,1])
108 plt.xlabel('Number of patterns')
109 plt.ylabel('Recall reconstruction success rate')
110 plt.show(block=False)
111
112 # Save results
113 np.savetxt('Results/Ex1Results.txt',Results)
114
115 # Plot of certain weight matrix indices over Z*c updates
116 # This is only used to analyse the impact of modifying the diagonal of the
        weights matrix, mentionned in the appendix of the report
117 plt.figure('Exercise 1 Weight')
118 plt.title('Different weight values over iterations')
119 weight_mem_plot = [[None] for i in range(hn.N)]
120 for j in range(0,hn.N,1):
121     for i in range(len(weight_mem)):
122         weight_mem_plot[j].append(weight_mem[i][j][0])
123     plt.plot(weight_mem_plot[j])
124 plt.xlabel('Z Iteration')
125 plt.ylabel('Weight value')
126 plt.show(block=False)
127
128
129 # Plot of the hamming distance over Z*c updates
130 # This is only used to analyse the impact of modifying the diagonal of the
        weights matrix, mentionned in the appendix of the report
131 plt.figure('Z - Hamming distance')
132 plt.title('Recall error in function of Z iteration')
133 distance_mem_plot = np.asarray(distance_mem)
134 plt.plot(distance_mem_plot[:,0],distance_mem_plot[:,1],'r*')
135 plt.xlabel('Z Iteration')
136 plt.ylabel('Error (Hamming distance)')
137 plt.axis([0,Z,0,1])
138
139 def func(x, a, b, c, d):
140     return a*x**3+b*x**2+c*x+d
141 popt, pcov = curve_fit(func, distance_mem_plot[:,0], distance_mem_plot
        [:,1])
142 f = func(distance_mem_plot[:,0], *popt)
143 plt.plot(distance_mem_plot[:,0],f, linewidth=3)
144
145 plt.show(block=False)
```

A.2.4 Exercise2.py

```
# Hopfield project
# Exercise 2
# Arreguit Jonathan & Bronner Timothee

# Find how the maximum number of patterns p_max scale with N for an error smaller than 0.05

# Results storage
Results_N_Pmax = np.array([[None for i in range(2)] for j in range(len(N))]) # [N,P_max]
N_n = 0 # Counter for number of sizes tested

for n in N:

    print 'Number of fully interconnected units:', n

    # Initialisation of parameters for size n
    error = 0
    previous_min = 0
    previous_max = 0
    found = False
    p = P_init

    # Creation of network
    hn = hopfield.hopfield_network(N=n, c=c, plot=FLAG_plot)

    # Search for maximum dictionary size for n dimensions
    while not found:
        hn.make_pattern(P=p, ratio=ratio)
        print 'Testing for', p, 'patterns with Pmin =',previous_min, 'and Pmax =',previous_max
        for z in range(Z):
            if FLAG_progress and z%(Z/10) == 0:
                print '\r', 100.0*z/Z, '%\r'
                loading = 0
            if rand.randint(0,1000)/1000.0 < P_s: # Storage
                mu = rand.randint(0,hn.count_patterns()-1)
                for r in range(c):
                    hn.update_weight(mu=mu, decay=decay)
            else: # Recall
                hn.recall(mu=rand.randint(0,hn.count_patterns()-1), P_f = P_f, decay=decay)

        # Average Hamming distance over the total number of recalls
        distance_mean = 1.0*hn.distance_sum/hn.recalls

        if distance_mean < error_max: # Bigger dictionary size possible
            if previous_min < p and previous_max < p:
                previous_min = p
                previous_max = p
                p = p*2
            elif previous_max > p:
                previous_min = p
                p = (previous_max+p)/2
                if p == previous_max-1:
                    found = True # Found maximum dictionary size
```

```
53
54          elif distance_mean > error_max: # Smaller dictionary size required
55              print 'Maximum error crossed for', p, 'patterns'
56              previous_max = p
57              p = (previous_min+p)/2
58              if p == previous_min:
59                  found = True # Found maximum dictionary size
60
61      # Results storage
62      Results_N_Pmax[N_n][0] = n
63      Results_N_Pmax[N_n][1] = p
64      print 'RESULT:\nFor patterns with', n, 'dimensions:\nMaximum
            dictionaray size is', p, 'patterns\n'
65      N_n += 1
66
67  # Plot of the maximum dictionary size in function the dimension N of
        patterns
68  plt.figure('Exercise 2')
69  plt.title('Maximum dictionary size in function of the number of dimensions\
        nfor a recall error below 0.05')
70  plt.plot(N,Results_N_Pmax[:,1])
71  plt.xlabel('Number of dimensions N')
72  plt.ylabel('Dictionary size P_{max}')
73
74  plt.show(block=False)
```

A.2.5 Exercise3.py

```
# Hopfield project
# Exercise 3
# Arreguit Jonathan & Bronner Timothee

# Find how the error changes with the weight decay factor

# Results storage
Results_decay_errorrate = [[None,None] for i in range(len(Decay))]
decay_n = 0 # Counter for number of decays tested

print 'NUMBER OF PATTERNS', p, '\n'
for decay in Decay:
    decay *= resolution
    print 'DECAY', decay

    hn = hopfield.hopfield_network(N=N, c=c, plot=FLAG_plot)
    hn.make_pattern(P=p, ratio=ratio)

    for z in range(Z):
        xi_start=round(z/T_window)%p
        window = [(i+xi_start)%p for i in range(m)]

        if FLAG_progress and z%(Z/10) == 0:
            print 100.0*z/Z, '%'
            loading = 0
        if rand.randint(0,1000)/1000.0 < P_s: # Strorage
            mu = window[rand.randint(0,m-1)]
            for r in range(c):
                hn.update_weight(mu=mu, decay=decay)
        else: # Recall
            hn.recall(mu=window[rand.randint(0,m-1)], P_f=P_f, decay=decay)

    print 'RECALLS:', hn.recalls
    print 'ERRORS:', hn.distance_sum
    print 'ERROR RATE;', 100.0*hn.distance_sum/hn.recalls, '%'

    # Storage
    Results_decay_errorrate[decay_n][0] = decay
    Results_decay_errorrate[decay_n][1] = 1.0*hn.distance_sum/hn.recalls

    decay_n += 1

    print # Next line

# Plot of the Error rate in function of decay value for a given m
plot_decay = []
plot_error = []
for i in range (len(Results_decay_errorrate)):
    plot_decay.append(Results_decay_errorrate[i][0])
    plot_error.append(Results_decay_errorrate[i][1])

plt.figure('Exercise 3')
plt.title('Error in function of decay $\lambda$ for sub-dictionary size m = 5')
plt.plot(plot_decay,plot_error,'-')
plt.axis(ymin=0,ymax=1.0)
```

```
56  plt.xlabel('Decay $\lambda$')
57  plt.ylabel('Error (Hamming distance)')
58  plt.show(block=False)
```

A.2.6 Exercise4.py

```
# Hopfield project
# Exercise 3
# Arreguit Jonathan & Bronner Timothee

# Find how the optimal weight decay factor changes as the sub-dictionary size m increases

# Results storage
Results_decay_errorrate = np.zeros((len(M),len(Decay)))
decay_n = 0 # Counter for number of decays tested
m_n = 0 # Counter for m

Decay_jump = Decay[1] - Decay[0]
M_jump = M[1] - M[0]

print 'NUMBER OF PATTERNS', p, '\n'
for m in M:
    for decay in Decay:
        decay *= resolution
        print 'DECAY:', decay
        print 'M SIZE:', m

        hn = hopfield.hopfield_network(N=N, c=c, plot=FLAG_plot)
        hn.make_pattern(P=p, ratio=ratio)

        for z in range(Z):
            xi_start=round(z/T_window)%p
            window = [(i+xi_start)%p for i in range(m)]

            if FLAG_progress and z%(Z/10) == 0:
                print 100.0*z/Z, '%'
                loading = 0
            if rand.randint(0,1000)/1000.0 < P_s: # Storage
                mu = window[rand.randint(0,m-1)]
                for r in range(c):
                    hn.update_weight(mu=mu, decay=decay)
            else: # Recall
                hn.recall(mu=window[rand.randint(0,m-1)], P_f=P_f, decay=decay)

        print 'RECALLS:', hn.recalls
        print 'ERROR RATE;', 100.0*hn.distance_sum/hn.recalls, '%'

        # Storage of results
        Results_decay_errorrate[m_n][decay_n] = 1.0*hn.distance_sum/hn.recalls

        decay_n += 1

        print # Next line

    decay_n = 0
    m_n += 1

# Heatmap of the error in function of the decay and the sub-dictionary size m
```

```
53 Decay.append(Decay[-1]+Decay_jump)
54 M.append(M[-1]+M_jump)
55 Decay_plot = np.zeros((len(M),len(Decay)))
56 M_plot = np.zeros((len(M),len(Decay)))
57
58 for i in range(len(M)):
59     Decay_plot[i,:] = Decay
60 Decay_plot *= resolution
61 for i in range(len(Decay)):
62     M_plot[:,i] = M
63
64 Error_rate_D_M = Results_decay_errorrate
65 figex4, ax = plt.subplots()
66 figex4.canvas.set_window_title('Exercise 4')
67 heatmap = ax.pcolor(Decay_plot,M_plot,Error_rate_D_M)
68
69 cbar = plt.colorbar(heatmap)
70 cbar.set_label('Error')
71 plt.title('Error in function of decay value\nand sub-dictionary size')
72 plt.axis([Decay[0]*resolution,Decay[-1]*resolution,M[0],M[-1]])
73 plt.xlabel('Decay $\lambda$')
74 plt.ylabel('Sub-dictionary size m')
75 plt.show(block=False)
```