



ÉCOLE POLYTECHNIQUE  
FÉDÉRALE DE LAUSANNE

BIOLOGICAL MODELING OF NEURAL NETWORKS

---

# Hopfield Model

STORAGE OF SEQUENCES OF PATTERNS IN ASYMETRIC  
HOPFIELD NETWORKS WITH DELAYED SYNAPSES

---

Miryam CHAABOUNI  
Joseph LEMAITRE

# 1 Exercise 1 : Standard Hopfield Network

## 1.1 Exercise 1.1 : Implementation

We created a class `hopfieldNetwork` which has the following attributes :

- `N` : number of neurons
- `pattern` : array of  $P \times N$  states, where `P` is the number of patterns
- `weight` : array of  $N \times N$  that represents the matrix of interaction between neurons
- `x` : state of the network at each time step, e.g.,  $x = \{1, -1, -1, \dots\}$

and the following functions :

**`__init__`** creates an instance of the class `hofieldNetwork` with the number of neurons `N`

**`makePattern`** creates a numpy array of  $N \times P$  with a given ratio of neurons with value 1 and the other at -1, using the function `numpy.random.choice`.

**`makeWeight`** calculates the interaction weights using `numpy.fromfunction` and a lambda according to the formula :  $w_{ij} = \frac{1}{N} \sum_{m=1}^P \xi_i^\mu \xi_j^\mu$

**`dynamic`** updates the state of neuron `i` according to the formula :  $S_i = \text{sign}(\sum_{j=1}^N w_{ij} S_j)$  using the weight matrix and the current state of the network

After creating an instance of the class `hopfieldNework` and creating the desired number of patterns, we run the simulation using the `run` function which does the following steps :

1. initialize the network by copying one of the patterns  $\xi^\mu$  then flipping the state of randomly chosen neurons
2. For all the neurons in the network, in a random order : we update its state using the `dynamic` function
3. repeat until convergence

For the convergence criterion, we store the value of the network  $x$  at each step, and after each iteration we compare the old value of  $x$  to the new one by a simple subtraction. If the difference is negligible, we stop. In addition, we set a time limit  $t_{max} = 100$  steps, to exit even if the model doesn't converge. At each time step, we store the value of the overlap and the normalized pixel distance which will be useful later.

## 1.2 Exercise 1.2 : Pattern retrieval

At this step we add two functions :

**overlap** calculates the overlap between the pattern  $\xi^\mu$  and the current state of the network according to the formula :  $m^\mu = \frac{1}{N} \sum_{i=1}^N \xi_i^\mu S_i(t)$

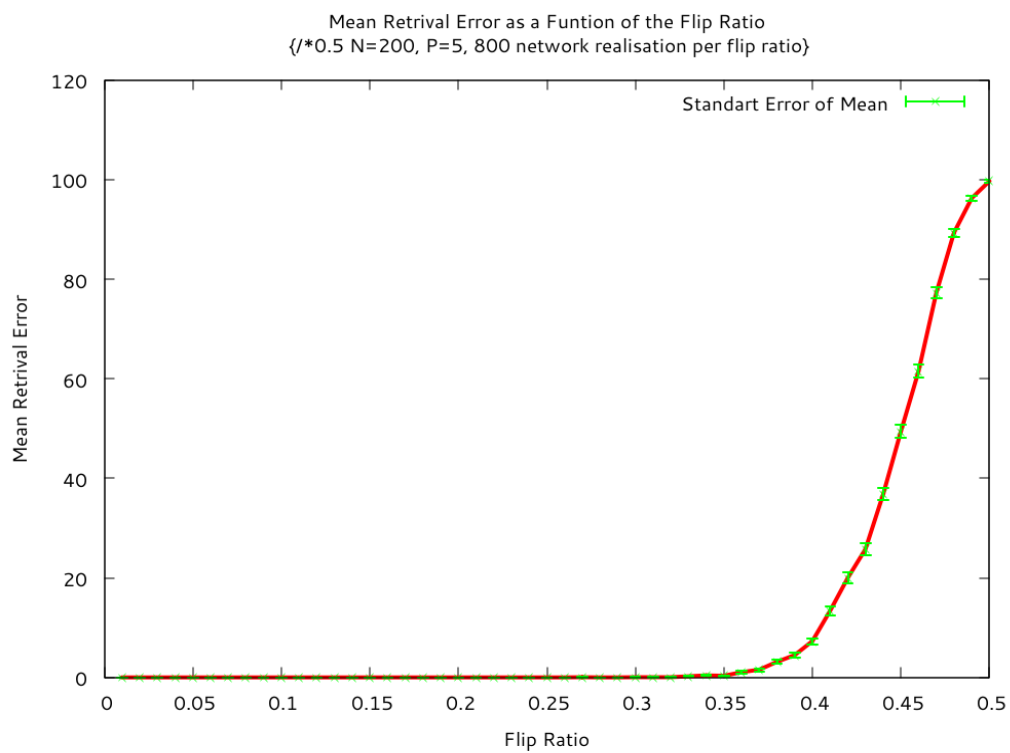
**pixeldistance** calculates the percentage of neurons in the network that differ from the pattern  $\xi^\mu$  using the formula :  $(1 - m^\mu) \times 100$ , where  $m^\mu$  is the overlap returned by the aforementioned function.

To get the retrieval error as a function of the ratio  $c$  of the flipped bits at the initialization of the network, we define the function `patternRetrieval`. We fix  $N = 200$  and  $P = 5$ . We take 50 values of  $c$  in the interval  $[0.01, 0.51]$ . For each value of  $c$ , we run the simulation 50 times, and get the pixel distance at the end of each simulation. We show the average of the retrieval error in figure 1.2. The error bar represents the standard error of the mean, and is given by SciPy function `stats.sem`. This error is small here because for each configuration  $N, P, c$ , we averaged over 50 realisations. We could do this because the code was fast.

As we could expect it, the error is negligible ( $< 1\%$ ) if we flip a reasonable ratio of the neurons at the initialization step. According to the figure, the threshold value is around  $c = 0.35$ . As the number of flipped neurons increases, the error grows exponentially. For  $c = 0.5$ , meaning that we start with a network where half of the neurons are different from the retrieved pattern, the error reaches 100%.

We conclude that the Hopfield model works well only if the initial state of the Network is close to the target pattern. That's in accordance with the associative memory theory : since it's content addressed data, we need to start near the data we want to retrieve.

Figure 1: Mean retrieval error as a function of the flip ration  $c \in [0.01, 0.51]$ .



### 1.3 Exercise 1.3 : Capacity estimation

The number of patterns that can be stored and retrieved correctly by a neurons network depends on the number of neurons. In Hofield model, the storage capacity of a network is defined by :

$$C_{stor} = \frac{P^{max}}{N} \quad (1)$$

where  $P^{max}$  is the maximum number of patterns that the network can retrieve correctly.

To study this capacity, we define the function maxLoad. We create a network of  $N$  neurons and a growing number of patterns  $P$ . We fix  $c = 0.1$ . For each  $P$  we try to retrieve all the patterns. When the mean error for all those try reaches 2% we stop and save the number of patterns created as  $P_{max}$ . We do this 10 times per  $N$  to have statistical average of  $P_{max}$  and  $\alpha_{max} = P_{max}/N$ . To speed up the simulation, we start from  $P = 0.1N$  because we are sure that a network with 500 neurons will be able to retrieve correctly at least 50 (and less pattern). If this assumption was false, we would have known by analysing the ouput of the program. The results are shown in table 1.3. The

$N$	100	250	500
$P_{max}$	$\approx 14$	$\approx 38$	$\approx 77$
$\alpha_{max}$	$0.1480 \pm 0.007423$	$0.1532 \pm 0.003322$	$0.1546 \pm 0.001550$

Table 1: Capacity storage of a network of  $N$  neurons

error on the value of  $\alpha_{max}$  is calculated by ...

As we could expect, we observe that the capacity of the network increases with the number of neurons. Moreover, we see that the maximal load that we obtain corresponds extremely well to the theoretical value.

Note that we only retrieve one pattern at a time. If we want to retrieve a sequence of patterns, the literature says that  $C_{stor}$  should not exceed 0.138[1], otherwise, the error will propagate at each pattern of the sequence. From our table, we see that this threshold is exceeded for  $N=250$  and  $N=500$ . Therefore, we should to modify the model if we want to retrieve a sequence of patterns.

## 2 Exercise 2

### 2.1 Exercise 2.1 : Implementation

A lot of the function used for exercise 1 are reused for the 2. Here we want to implement a particular type of Hopfield Network that have a compoment added : the Assymetrical weights. We used Oriented-Object Inheritance to derive our class `hopfieldNetworkAsymmetric` from the `hopfieldNetwork`. Our attributes are the same than in the first part, with the addition of :

- **Assymweight** : array of  $N \times N$  that represents the projections of the weight from a pattern on the one of the next.
- **SPrev** : array of  $N \times t_{max}$  that contain the evaluation of  $S$  for all  $t \in [0, now]$ . It is used to calculate the  $\bar{S}$  that will stabilize our transitions.

We add the following methods :

**makeAssymmetricWeight** calculates the projections of weights on each other using the other weight according to the formula :  $w_{ij}^L = \frac{\lambda}{N} \sum_{\mu=1}^P \xi_i^{\mu+1} \xi_j^{\mu}$

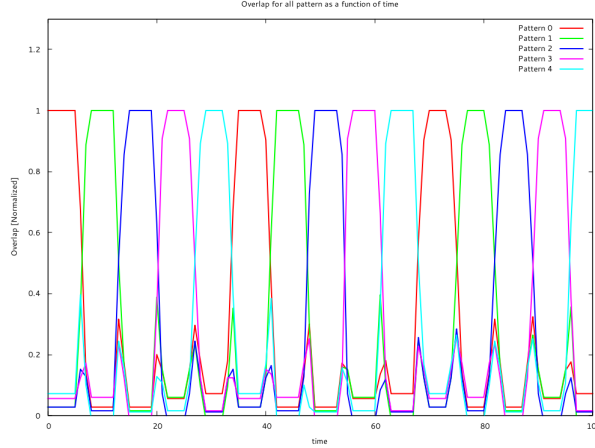
**filterfunction** The implementation of our filter function. We choosed the heaviside function :  $G(t) = \frac{1}{\tau} \Theta(-t + \tau)$

**dynamic** It is a two step function that performs the evolution of the network for one step. Since we do synchronious update, all those variables are vectors.

- First we calculate the new  $\bar{S}$  from the memory of all previous states, as  $\bar{S} = \sum_{t'=0}^t G(t') S(t - t')$
- Then updates the state of all the neurons in one step, according to the formula :  $S_i = \text{sign}(\sum_{j=1}^N w_{ij} S_j + w_{ij}^L \bar{S})$  using the two weights matrix with the current and filtered state of the network.

The process is the same as in exercice 1. If we plot the overlap for all the patterns at every time step, we see that our network goes from one stored pattern to another (Figure 2.1)

Figure 2: Illustration of the sequential behaviour



## 2.2 Exercise 2.2 : $\lambda$ range estimation

We used the heaviside filter function. We had a bit of freedom to define *sequential behaviour*. We decreased lambda to get the minimum value for which all pattern are visited. We found :

$$\lambda_{min} \approx 0.9$$

For values below this one, the network get stuck in a particular pattern. Due to the random generation of some variables, the value of  $\lambda_{min}$  is approximative.

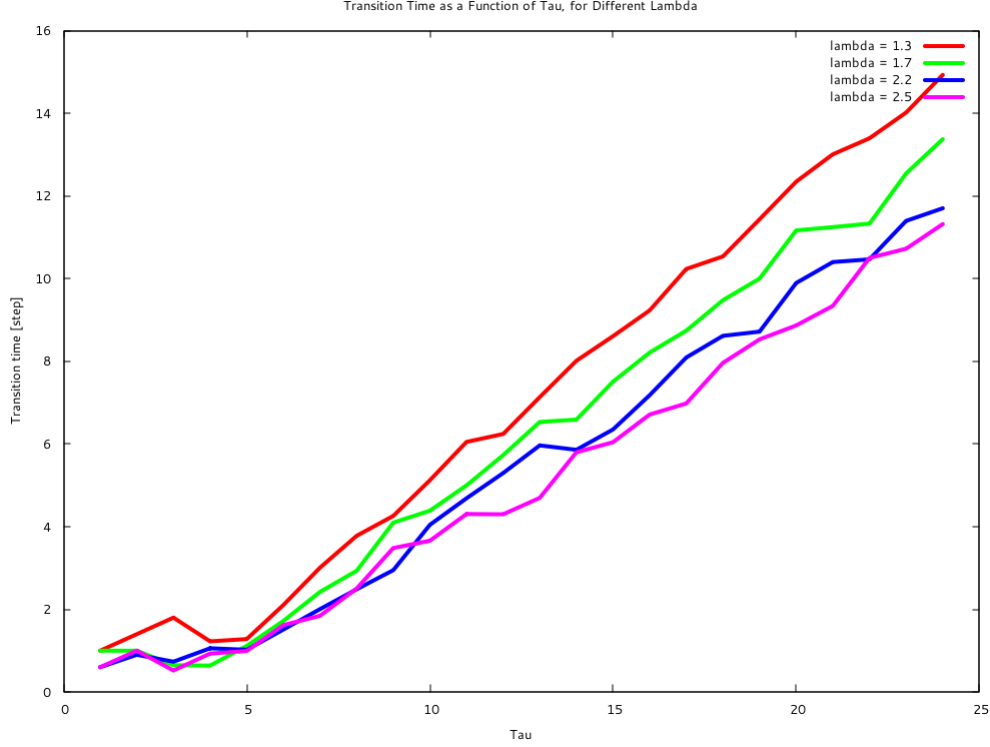
For the calculus of  $\lambda_{max}$ , we saw that for high lambda the network jump over some frame, not getting an overlap of one. Therefore we defined the sequential behaviour as all state are visited with a overlap of 1, one after each other. We found :

$$\lambda_{max} = 3.6$$

## 2.3 Exercice 2.3 : Estimation of the transition time

The transition time (that we defined as the time the network stay in a given frame with a overlap of 1 before trying to catch another) depend on  $\tau$ , the filter function parameter and on  $\lambda$ .

Figure 3: Mean retrieval error as a function of the flip ratio  $c \in [0.01, 0.51]$ .



We run our network 5 times for each configuration ( $\lambda \in [1.3, 1.7, 2.2, 2.5]$  and  $\tau \in [1, 25]$ ), saving the transition time on a file to plot it with gnuplot.

The results are displayed in Figure 2.3. If we discard the small values of  $\tau < 5$ , we see a linear behaviour, the transition time increasing with  $\tau$ . The slope is fixed by other parameters, and  $\lambda$  gives a shift of downward to the curve curve.

## 2.4 Exercice 2.4 : Sequence Storage Capacity

By varying  $P$  with  $N$  fixed, we saw that increasing the number of pattern stored lead the network to never acheive an overlap of 1, or only for a few pattern (with  $N = 500$  and  $P = 40$  for example, only one pattern, the 4, was retrieved correctly). If we print the overlap we see that for a high  $P$ , the mean overlap is considerably lower (only 22 out of 400 iteration sees their



overlap over 0.7 for  $N = 500$  and  $P = 50$ , this number was 375 with the same setup, for  $P = 10$ ). So transition from fully recovered patterns are our control behaviour to test the number of pattern we can store.

We fixed all the parameter of our network, except the number of pattern  $P$  and the size of the network  $N$ . We then moved  $P$  until one of the pattern stored was missed during an iteration (this is our criterion but several can be chosen, like a threshold with the transition time between two pattern that get very long with  $P$ ). The results are displayed on Table 2.4. Again due to the randomness of the process we show here statistical average, but on too few sample to be sure.

$N$	250	500	1000
$P_{max}$	15	33	52
$\alpha_{max}$	0.06	0.066	0.052

Table 2: Sequence storage capacity of a network of  $N$  neurons

We can compare those results to the one for a non-sequential network. Our maximum load is significantly lower, divided by a factor  $\approx 2.66$ .

## References

- [1] W. Gerstner, W. Kistler, R. Naud, L. Paninski, *Neuronal Dynamics : from single neurons to networks and models of cognition*, Cambridge University Press, 2014.