# An investigation of memory implementation in discrete recurrent neural networks

Aude Forcione-Lambert
University of Montreal, Faculty of Arts and Sciences
supervised by Guillaume Lajoie

August 22, 2018

**Abstract**

We first replicate the findings on the 3-bit flip-flop recurrent neural network from Sussilo and Barak [10], originally in continuous time, with a network using discrete time-step. Using slow points analysis and principal components analysis, we show that the strategy used by our network to solve the task is essentially the same as in the original paper. We then modify the task to add a delay of several time-steps and study the way the network adapts to fill the new requirements. Using again principal components analysis and the networks eigenvalues, we show that the networks naturally converge to a solution similar to a dampened Fourier transform in order to achieve the required step function. This opens up avenues of inquiries about how the brain keeps track of time and how complex time-dependent tasks can be achieved in the field of artificial intelligence.

## 1 Introduction

Neural networks are increasingly used in machine learning to solve complex tasks. Despite their now established usefulness, the dynamics developed internally during training are only just starting to be studied[6]. In parallel, artificial neural networks have been used for years by computational neuroscientists in an effort to test hypothesis by replication and to obtain systems that are easier to measure than biological neurons [13]. In both cases, the theory of dynamical systems is used to understand the non-linear high-dimensional systems studied, and a mathematicians perspective is an asset.

Some of the most powerful neural networks in artificial intelligence as well as in the brain are recurrent neural networks (RNN). These networks have the particularity of including feedback connections. Not only does this permits them to remember and process information on a long timescale [5, 7, 11] (an ability essential in many tasks, for example in text comprehension), but it also introduces nonlinear dynamics and even potentially chaotic behaviors [2, 9] that makes it possible to solve much more complex problems that are themselves nonlinear. In this project we study how a one-layered RNN implements two simple short-term memory tasks, using tools from the mathematical field of dynamical systems.

## 2 Methods

### 2.1 Mathematical theory

#### 2.1.1 Dynamical systems

A dynamical system is any time-dependent function that describes movement on a manifold. A

neural network is a dynamical system where the phase space is such that each coordinate represents the state of a single neuron, and the time-dependent function map is determined by the strength of the connections between neurons (connections analogous to axons in the brain).

While linear dynamical systems are always solvable analytically, nonlinear networks are generally not. The theory of dynamical systems develops tools to better understand how complex dynamical systems behave when solving them isn't possible. In this work we use primarily slow point analysis and principal component analysis to characterize the neural networks studied.

#### 2.1.2 Slow point analysis

When working with complex dynamical systems, often the first step to understanding how it works is to identify its fixed points. But, while attractors are easily identified by modeling the system and letting trajectories converge naturally, saddle points are another story. When working with neural networks receiving inputs, saddle points with few unstable dimensions are especially useful to find as they will determine how the trajectory in phase-space reacts to inputs.

This is where slow point analysis is useful. We know that saddle points should be "slower moving" than the region around them. Since neural networks are by design first order differential or difference equations, speed is already available. Sussillo and Barak developed slow-point analysis to find near-linear points. Specifically they used the "kinetic energy" of the system:

$$q = \frac{1}{2}\left|\dot{S}\right|^2$$

where $S$ represents the phase space variables of a dynamical system.

In this project we used a discrete-time network, so the equation had to be modified slightly:

$$q = \frac{1}{2}\left|S(t+1) - S(t)\right|^2$$

The minimums of the function indicate the fixed points of the system and the type (attractor, saddle point) can be determined by the number of positive eigenvalues of the Jacobian matrix at that point.

#### 2.1.3 Principal component analysis

When trying to understand high-dimensional dynamical systems, It would be useful to find the most significant dimensions to visualize the data. This is precisely the task for which principal component analysis (or PCA) was developed [8]. The goal of PCA is to find a sorted orthogonal basis such that the first basis vectors are the most significant to understand the data.

More precisely, let's suppose we have a collection of points $X$ in a $n$-dimensional space (such that $X$ is a matrix of dimension number of points $\times\, n$). We want to find the unitary vector such that the variance of $X$ is maximal along its axis. This will be the first principal component. We then find another unitary vector that is orthogonal to the first one such that the variance is again maximal in that direction. This is the second principal component. We repeat that process until we've found a new n-dimensional basis or weve found enough components to suit our purpose.

To do this, we first compute the covariance matrix:

$$C_X = \frac{1}{n}X^T X$$

We then diagonalize the matrix, sorting the eigenvalues in descending order. Because covariance matrix are by definition symmetric, they are always diagonalizable. We will get an orthonormal matrix $P$ and a diagonal matrix $C_Y$ such that

$$C_Y = PC_X P^T$$

Where $P$ is the principal components matrix.

### 2.2 Recurrent neural networks

#### 2.2.1 Task definition

The first task to be implemented was the *3-bit flip-flop*[10]. The network received inputs from three independent channels consisting of spikes

of 1 or -1 at random intervals (Poisson distribution with $\lambda = 50$). The network then had to match and maintain the inputs in three different outputs, without cross-talk. Later we added delay to the expected output as an extension of the study. We go into more details about the delayed task in a later section of the report. A snippet of the input-expected output signal (with and without delay) is shown on figure 1.

#### 2.2.2 Network structure

The network was a one-layered discrete and densely connected recurrent neural network with a 100 neurons. The network is composed of three real-valued matrix, the square matrix $W$ representing the recurrent connection between the neurons, the $3 \times 100$ $W_{in}$ matrix representing the strength of connections between the input channels and the neurons of the network and the $100 \times 3$ $W_{out}$ matrix representing the strength of the connection between the neurons of the network and the output neurons. At each neuron, the signal goes through an "activation function", in this case the hyperbolic tangent.

Let $X$ be the 3 dimensions input function over time, $S$ be the 100 dimensions network state over time and $Y$ be the 3 dimensions output function over time. The update rule is then defined as

$$S(t) = \tanh\left(X(t)W_{in} + S(t-1)W\right)$$

$$Y(t) = \tanh\left(S(t)W_{out}\right)$$

A schematic of the neural network is shown on figure 2**A**.

The hyperbolic tangent is often used as the activation function in studies about neural networks as it is a good representation of biological spiking neurons. It models the way the number and strength of excitatory and inhibitory signals entering the neuron affects its spiking frequency[3].

A discrete as opposed to a continuous-time network was chosen for a few reasons. First, we wanted to see if the findings of Sussillo and Barak could be transposed to a discrete-time network, as the researchers themselves where wondering in their paper. This was further motivated by the fact that in the field of artificial intelligence, discrete-time networks are widely used[6] as they are better adapted to the computer architecture. Any link between the study of neural networks for neurobiology and for artificial intelligence would be beneficial to both fields. Last but not least is the fact that discrete-time networks are much easier and faster to implement than their continuous cousins and a wide array of resources to train them are freely available.

#### 2.2.3 Training

The $W_{in}$, $W$ and $W_{out}$ weights matrices were originally set randomly (using a linear probability function) between $-\frac{1}{\sqrt{3}}$ and $\frac{1}{\sqrt{3}}$ for $W_{in}$ and between $-\frac{1}{\sqrt{N}}$ and $\frac{1}{\sqrt{N}}$ for $W$ and $W_{out}$ where $N$ is the number of neurons. This strategy is often used in the literature to get an initial output with a probability distribution approaching a normal law with average 0 and variance 1. Only the $W$ and $W_{out}$ matrices were trained.

The first step of training the network was to define the training sample. Since the network is recurrent, the output is potentially the result of an infinite number of steps. The solution is to "unfold" the network, that is consider only a certain number of steps back in time during training. In figure 2**B**, we see the unfolded version of the network.

It is important to choose carefully the number of unfolded steps used during training. Too many steps and we risk having the training algorithm correct too aggressively the weights as information about the error becomes less relevant the longer you go back in time. Too few steps and the network won't have time to exhibit its full behavior, rendering the training algorithm incapable of effectively correcting the error.

Through trial and error we finally settled on feeding inputs 10 time-steps before comparing the effective output to the desired one. We also fed back the last state of the networks at the end of a training step as the initial state during the next training step, which assured that late in the training process, when only small corrections are

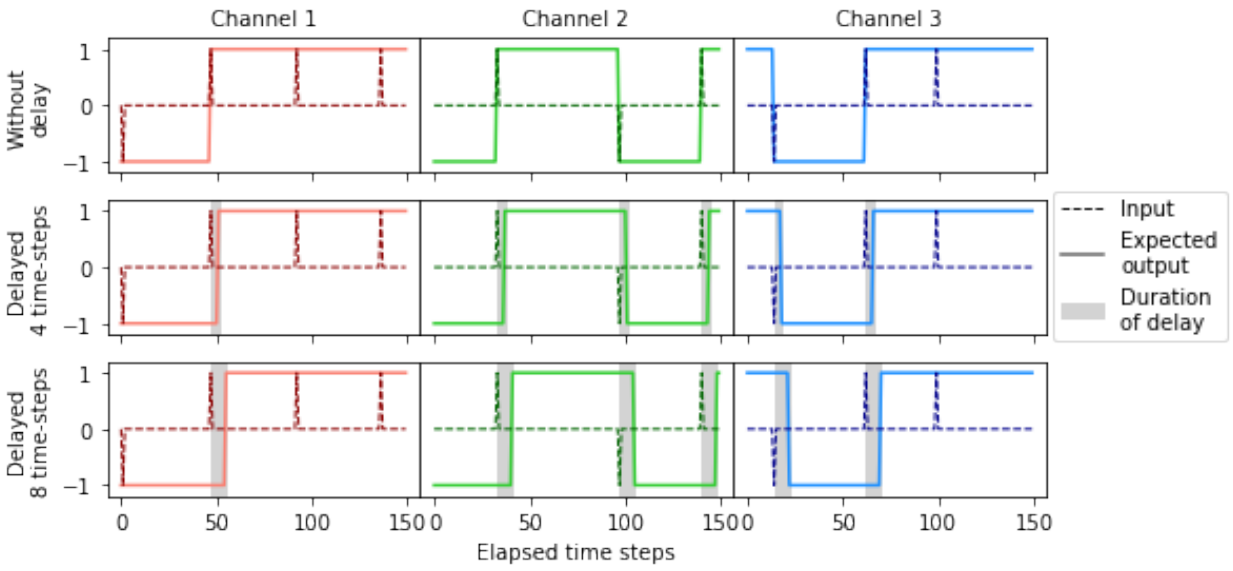


Figure 1: Task definition. The neural network is expected to match and maintain the last input for each channel while preventing cross-talk. For the delayed task definition, the neural network must in addition delay its response (and therefore remember the input) during a certain number of time steps.

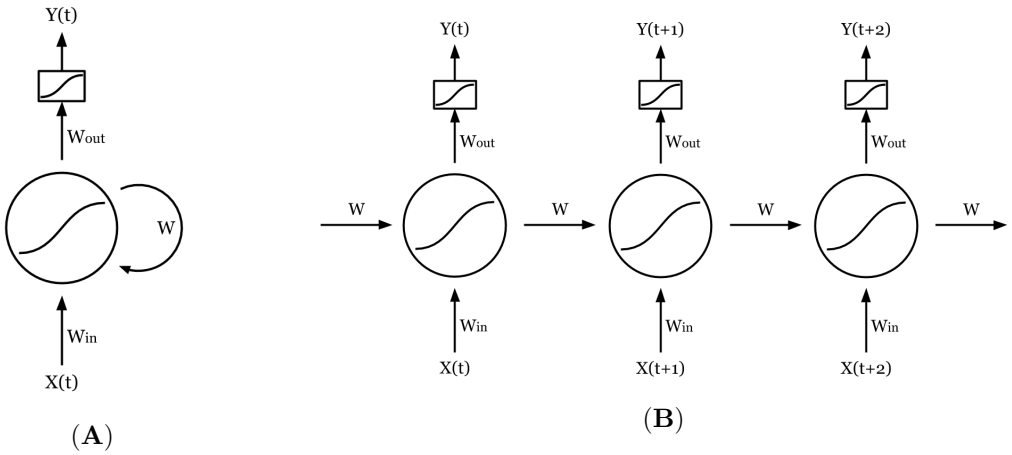


Figure 2: **(A)** Diagram of the recurrent neural network. Each time step, the neurons receive the previous states of all neurons through a weight matrix representing the strength of the connecting axons. Every neuron in the network uses the hyperbolic tangent as their internal function. **(B)** The unfolded diagram. A truncated unfolded version of the recurrent neural network is used during training.

required, the networks start close to their ideal state.

Once we've defined the parameters of the training task, we can start actually training the network. Training a neural network requires two principal algorithms: a definition of error and an optimization method. The choice of these algorithms among the many available will depend on the architecture of the network, the nature of the task and often a few trial and errors.

The definition of error essentially turns the training into a minimization problem. Every set of weights corresponds to a point in a high-dimensional scalar potential function.

For the networks used during this project, we used the mean squared error which is widely used in the field for networks outputting degrees of confidence in binary solutions. If you use $n$ different samples for each training step, that for each sample you expect an output $\hat{Y}$ and get the output $Y$, the mean squared error is given by

$$\frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

You need to evaluate many samples at a time since the training samples differ from the testing sample. Therefore the computed error is only an estimation of the real scalar potential error function. The more elements in a batch, the greater the confidence in the direction of optimization, but the fewer batches are available from a certain set of data. As always with neural networks it's a question of finding balance through trial and error. In this case we used batches of 300 samples (each sample being 10 time-steps long as discussed above).

For the training algorithm we used the Adam optimizer[4]. We will not expand on the Adam optimizer here as it is another field of study entirely, but we will posit that it is superior to the traditional gradient descent in this context as gradient descent is too blunt and will continuously overshoot the target towards the end. In contrast, the Adam optimizer keeps track of the speed and acceleration of training, scaling down the error correction as the weights approach their optimal configuration.

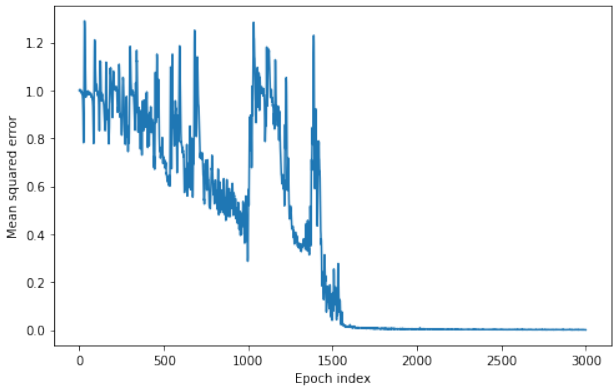


Figure 3: Sample loss over training epoch. This graphic was generated during training of the RNN with a random seed of 7 and a delay of 4 time-steps.

We included a typical error over training step or "epoch" function in figure 3.

When compiling the final data, we trained 100 different networks, with delays from 0 to 9 time-steps, with 10 different random seeds. Networks with the same random seed but different delays have the same $W_{in}$ matrix, and started with the same $W$ and $W_{out}$ matrices during training. Since all networks were trained with the same data set, this insured that differences between networks were due to implementing different tasks and not random chance. Using different random seeds helped give more credibility to the results by insuring that any observed behavior wasn't an isolated case.

In the results, we only use the networks with a random seed equal to 7 as the resulting graphics were the most aesthetically pleasing, but any discussed feature was also observable in the other networks.

### 2.3 Computing tools

It is not necessary to read this section to understand the project but it could be useful for students or researchers looking to replicate or further the project using the same tools. In this subsection, you will find the definitions of the main softwares and libraries used while developing the code, as well as useful introductory tutorials.

Note that the code used to generate all graphics in this paper is available on our github[14], as well as all final code developed during the project.

#### 2.3.1 Git

During the project, all code was saved and shared through git. Git is a version-control free software. It lets multiple users collaborate on projects, including sharing documents and merging modifications efficiently. It also keeps backups of all past versions of the code, allowing to see and go back in the historic. The website github is the most widely used platform for publishing projects developed with git. It also has some great resources for learning to work with git, including hands-on interactive tutorials[16].

#### 2.3.2 Jupyter

We used jupyter notebooks to test new code and create the graphics used during the project. Jupyter is a software providing an interactive environment for python and many other programming languages. The lab uses a JupyterHub server permitting anyone with an account on the DMS's cloud server to connect through https and work remotely. Note that the cloud server is only accessible from the University of Montreal's network. If you are not directly connected to the University, you will first need to connect to it via the VPN[18]. Once connected to the network, open your browser and go to "`https://cloud.dms.umontreal.ca:8000`". Use your DMS's identification and password when prompted.

To learn to use the Jupyter notebook, the "Help" section in the tools bar of the notebooks is very well designed. I recommend doing the "User Interface Tour" and reading the "Keyboard Shortcuts" page.

#### 2.3.3 TensorFlow

TensorFlow is a python, Java, C and Go library optimized for machine learning[19]. It is developed and maintained by Google and is currently the most popular, stable and efficient library in the field. The project was developed using the low-level API for more flexibility. You can find a tutorial for using the API on the TensorFlow project website[15].

For our project we used TensorFlow version 1.5.0 for training and testing the networks.

#### 2.3.4 Scikit-learn

Scikit-learn is a suite of libraries for machine learning in python[20], although we only used its PCA object in the project. The PCA object provides helpful functions and variables to interpret data through principal components analysis, and lets us do so without cluttering the code with too many elementary equations. A link to the object's documentation is provided in the bibliography[17].

#### 2.3.5 PyTables and Pandas

A large amount of data was generated during the project and had to be stored efficiently. All data was stored using the HDF5 file format[22], which was specifically created for saving long lists of data in a portable format. To read and manipulate the HDF5 files, we used both pandas and PyTables[21].

We find that PyTables is more efficient and gives more options when saving data. We used it in python files doing the heavy-lifting part of data generation. The PyTables team have a short introductory tutorial on their website[0].

Pandas is a bit less flexible, but it has useful display functions for Jupyter notebooks. We used it when experimenting with the data. The Pandas team have a list of internal and external tutorials on their website[0]. We recommend starting with "10 minutes to pandas".

## 3 Results and Discussion

### 3.1 Replication

After successfully training the first neural networks (using the original task definition without delay), the first task was to try to replicate Sussillo and Barak's observations. We tested the

trained network during roughly a hundred thousand steps saving the neurons states and the network output at every step. We then analyzed the resulting signal in the 100-dimensions phase space.

For the slow point analysis, if we let $S$ be the current state, the update function ignoring input will be defined as:

$$F(S) = \tanh(SW)$$

and the modified "kinetic energy" function as before:

$$q(S) = \frac{1}{2}\left|F(S) - S\right|^2$$

We used the default scipy minimize function, starting from 500 different random positions, to find all the slow points. We compute the gradients at each point:

$$\frac{\partial q_i(S)}{\partial x_j} = W_{ij}\,\mathrm{sech}^2\left(\sum_{k=0}^{N} W_{ik}S_k\right) - \delta_{ij}$$

where $\delta_{ij} = 1$ if $i = j$ and 0 otherwise and $N$ is again the number of neurons. Note that if you try reading the code, the $\frac{1}{\cosh}$ function is used instead as numpy does not implements the sech function. Finding the Jacobian matrix at the position of the slow point and counting its number of positive eigenvalues gives the number of unstable dimensions of the slow point.

The PCA projection indeed showed the characteristic "cube" discovered in the original research, as visible in figure 4**A**. Since the network has three channels with two possible states each, It effectively has eight possible states in total. Here the corners of the cube represent these eight states, while the three directions parallel to edges of the cube map to the three output tensions.

The only difference with the original result in continuous time is the "loops" protruding from the corners. We will discuss these loops later when analysing the network with delay.

The slow point analysis was again plotted in the 3 main PCs as shown in figure 5**A**. As a general rule, we find attractors at the corners of the cube described by the path of the states, saddle points with one unstable dimension on the edges, saddle points with two unstable dimensions on the faces and a saddle point with 3 unstable dimensions at the center. This is also consistent with the findings of Sussillo and Barak.

### 3.2 Furthering the study: introducing a delay component to the task

After putting the network architecture in place and satisfyingly showing that the RNN used the same strategies in discrete-time as in continuous-time to solve the 3-bit flip-flop task, we decided to study memory in a new context. With the original network, the task was to maintain a certain state until a new input was given. As we saw, this can be achieved with a relatively simple set of fixed points. But artificial intelligence and neural networks in the brain often have to accomplish tasks that require keeping track of time on the short term. To simulate this kind of task, we modified the target function to be delayed a few time-steps (see figure1). Now, once the network receives a new input, it needs to refrain from changing states immediately, maintain its current state until the prescribed delay has past, then switch to the new state, well after the input was presented.

With a 100 neurons, the networks could perform the task up to a delay of about 9 time-steps before failing to fit one of the outputs correctly. We again analyzed these networks using PCA and slow point analysis. We included the results for the networks with a random seed of 7 (the same seed as the featured network without delay) and delays of 4 and 8 time-steps in figures 4**B**, 4**C**, 5**B** and 5**C**.

The PCA shows little differences when augmenting delay. The most striking difference is the disappearance of the loops at the corners. We see small remnants of them in the figure for the network with a delay of 4 time-steps, but they become completely invisible in the figure for the network delayed by 8 time-steps. Our theory is that these loops correspond to inputs that must be ignored as the network is already outputting the correct value in that particular

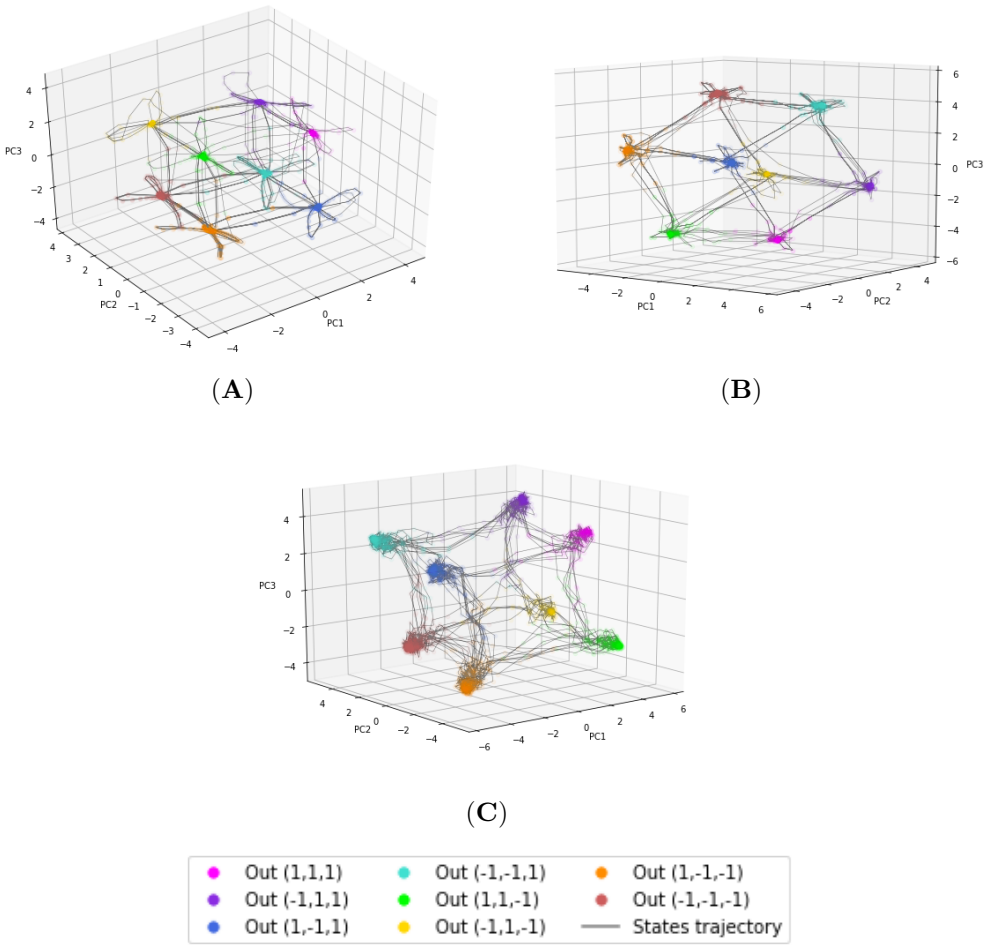


Figure 4: **(A)** Dynamics during 3000 time-steps of the test for the network without delay. The states of the neurons are plotted according to the first three principal components. Each point represents the network's state at a different time-step and is color-coded according to the effective output value at that time. The trajectory is plotted in gray. In the absence of input, the network will maintain its position in phase-space. Injecting an input in the network induces a switch of the network towards the new output. **(B)** Dynamics of the delayed network. The network must generate the same output as before but delayed by 4 time-steps. **(C)** Dynamics of the network delayed by 8 time-steps.

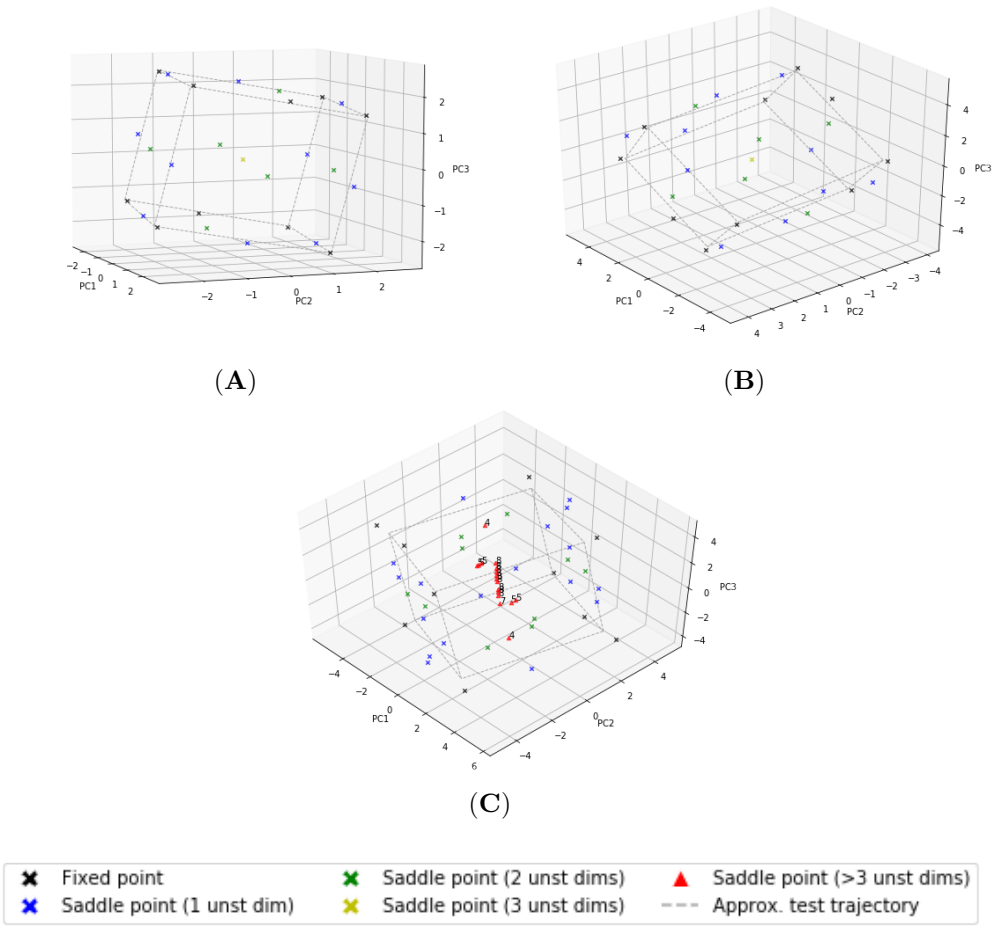


Figure 5: **(A)** Slow point analysis of RNN without delay. For clarity, a cube is used to mark the approximate trajectory of neurons states during the test. The slow points are color coded with respect to their number of unstable dimensions. This is consistent with the findings of Sussillo and Barak. **(B)** Slow point analysis of RNN delayed by 4 time-steps. There are no discernible difference with the slow point analysis of the network without delay. The delay is rather implemented by changing the direction of the input vectors in phase space. **(C)** Slow point analysis of RNN delayed by 8 time-steps. There are now slow points with more than three unstable dimensions. Those are marked by red triangles and the number of unstable dimensions is written above.

channel. This gives a hint about how the network processes inputs.

Let's call the space spanned by the three first PCs the "output space". Evidently this space is the one (or is very close to) the space spanned by the three vectors in $W_{out}$. Similarly, let's call the space spanned by the three vectors in $W_{in}$ the "input space". In a network without delay, the input and output spaces can correspond as inputs are expected to be immediately reflected to the outputs. In the network with a 4 time-steps delay, the input should on the contrary be ignored at first by the output neurons. This means that the input and output spaces should be (almost) completely orthogonal. This explains why the "loops" are almost invisible. In the network delayed by 8 time-steps, the loops disappear completely, indicating that there are even more hidden dynamics separating the input and output spaces.

The slow-point analysis is virtually identical for the networks without and with 4 time-steps of delay. The network with 8 time-steps of delay is much more cluttered and a line attractor appears to the center, but I suspect that this is a result of the networks' capacity being stretched to the limit. As we will see, such a long delay monopolizes a lot of dimensions; the resulting loss of exactitude might simply be a case of "close enough" for the requirements.

With these observations in mind, we would like to investigate further the different hidden dynamics brought by the added delay. The standard PCA and slow-point analysis failed to sufficiently inform, so we turn to triggered averages.

We searched every instance during testing when the network had to switch from state (-1,-1,-1) to (-1,-1,1) (this choice of input switch was arbitrary but fixed) and kept a few steps from before the switch up to a few steps after it. We then averaged the signals for each instance found, aligned at the switch time, and obtained a single short signal representing the mean activity surrounding the switch. The PCA projection of this new signal revealed interesting features as shown in figure 6.

The periodic nature of the observed signals motivated us to plot the eigenvalues of the $W$ matrix in the complex plane (figure 7). The complex values of the eigenvalues give information on the rotating speed in different dimensions.

The triggered PCA projections reveal what the hidden dynamics look like. In figure 6**I**, we see a decomposition akin to a Fourier analysis of a step function. The network activities in the second and third principal components have sinusoid shapes with approximately the same frequency and a relative phase shift of about a fourth of their wavelength. We see the same pattern in the fourth and fifth principal components but with a frequency about twice as fast. Again the sixth and seventh components oscillate rapidly at about the same frequency. However, contrary to Fourier transform components, the oscillations are dampened over time. There is also a first principal component that is not sinusoidal. What we see is effectively a dampened Fourier analysis. The first principal component starts drifting as soon as the input is received, effectively tracing a sort of sinusoid with a very slow frequency. At the same time, in different perpendicular dimensions, a series of oscillators receive energy from the input and start tracing sinusoids. As time goes on, the first PC continues drifting until it reaches a new stable state while the other principal components slowly lose energy and their movement flattens out. The final effect when adding these signals with the output matrix is a clean step function.

The same type of dynamics but slightly less complex is visible in figure 6**B**, for the network with a delay of 4 time-steps. In comparison, the network without delay shown in figure 6**A** only has a spike at the moment of the input, probably to inhibit outputs in the other channels.

This interpretation is also supported by the eigenvalues analysis. The network without delay has only three eigenvalues that distinguish themselves from the random cloud around zero. They are on the real axis (in other networks two of these had imaginary parts but were still very close to the real line), and slightly higher than one. Considering the hyperbolic tangent activation function applied at every time-step, this indicates three dimensions in which the state components are maintained at about 1.

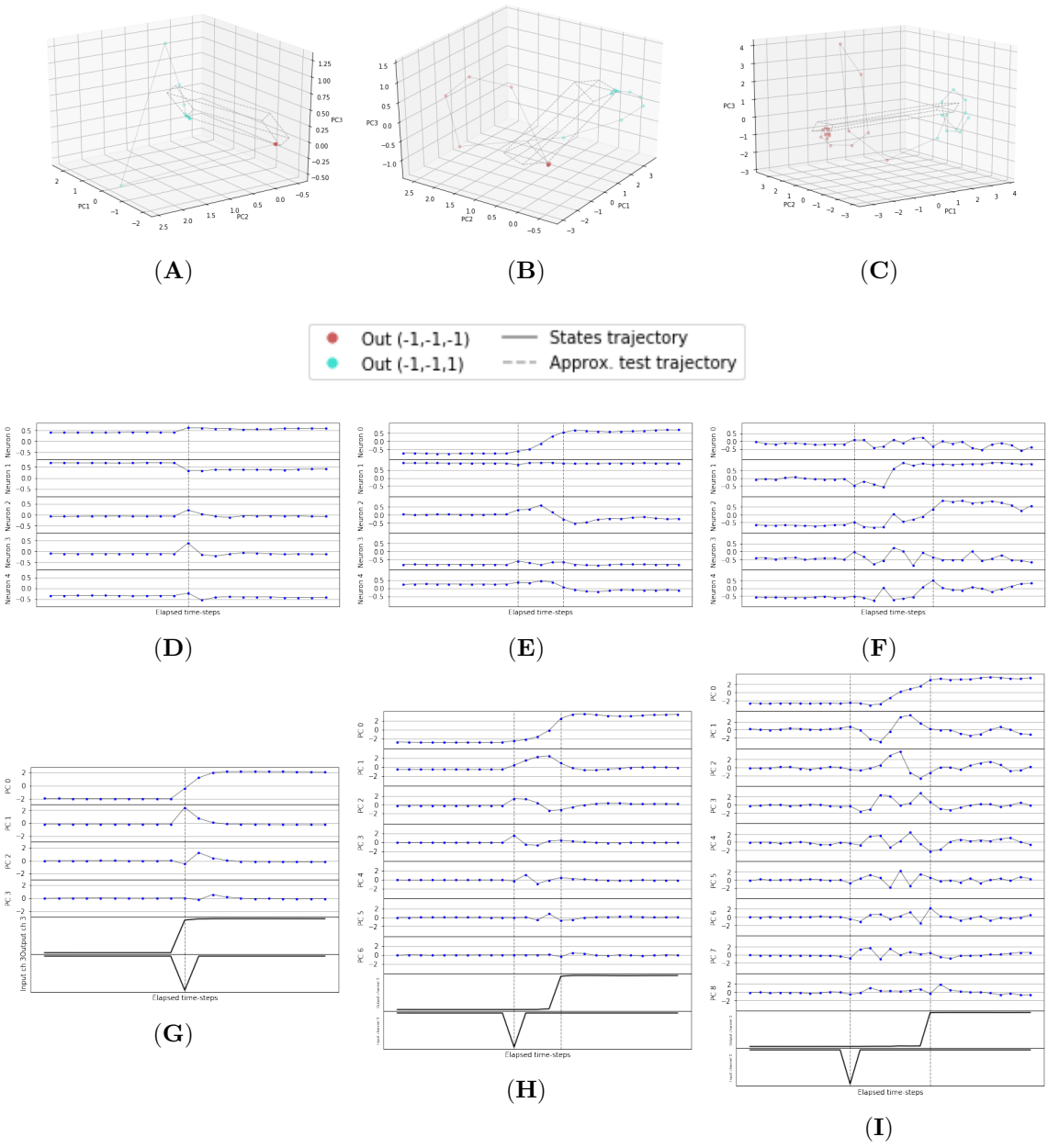


Figure 6: **(A) to (C)** The switch from output (-1,-1,-1) to output (-1,-1,1) for the RNNs without delay, delayed by 4 and delayed by 8 time-steps was studied using a triggered average. The resulting signal was then plotted according to the first principal components. **(D) to (F)** Sample neurons' tensions over time. **(G) to (I)** The first few principal components for each network. We see that for the network without delay, almost all relevant information is contained in three dimensions and therefore is accurately represented in (A). However, for the delayed network, there are interesting dynamics in the dimensions that were discarded in (B) and (C). The output was calculated from the triggered average with each network output function.

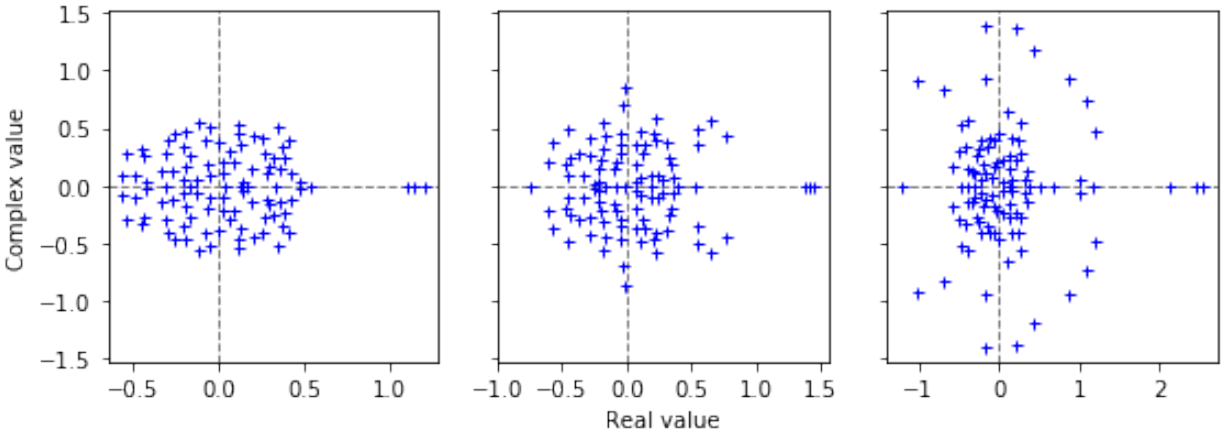


Figure 7: Analysis of the eigenvalues for the three studied RNNs. The eigenvalues of the network without delay is displayed to the left, the ones for 4 time-steps at the center and the ones for 8 time-steps to the right.

The story becomes more interesting with the networks delayed by 4 and 8 time-steps. The three main eigenvalues are still present, indicating stable values. Their increase in magnitude may be to suppress the other dynamics in these three dimensions. However, the rotating dynamics are clearly indicated by eigenvalues with smaller magnitude but larger varying angles. The different angles represent different rotating speeds, and as seen previously in the triggered averages, the network delayed by 8 time-steps has more periodic dimensions than the one delayed by 4 time-steps.

This additional information increases our confidence in our hypothesis.

## 4 Conclusion

It has been suggested in recent studies that periodic signals might be essential to performing memory-dependent tasks [12], but to our knowledge such a well-coordinated dynamic has never been observed.

However, the network used in this study is unrealistic as a biological simulation for a few reasons. First, as already discussed, it evolves in discrete time. In the brain, the neurons of course work continuously. This is significant as discrete systems can exhibit behaviors that are impossible to reproduce in a continuous system with the same number of dimensions. For example, see Strogatz [1] for one dimensional maps versus dynamics on the one-dimensional line. Despite this, we do not think that this is a large obstacle to application of the results in a biological context. Discrete dynamics can be replicated in a higher dimension continuous system, and as shown in the first part of the results, the present network seemed to behave very much like its continuous homologue in the original task.

What might be a larger problem is the way the recurrent connections are defined in our study. In the brain, the neurons are sparsely connected. That means that a neuron has only a small probability to be connected to any other given neuron. In comparison the network used in this study is densely connected, which means that every neuron is connected to every other. We do not know how the implemented solution to the delayed task might be influenced by switching to a sparsely connected architecture. It might be interesting to study this in a later research.

Another obstacle that is often referenced in studies about neural networks in the field of neurobiology is the unrealistic training algorithm. In the brain, there is no structure defining what the right output is and correcting network weights in consequence. Lately more realistic learning rules have been developed. It might be interesting to

see to what solution the network would converge with these new learning rules and if the dampened Fourier analysis can still be developed using those algorithms. Even simpler would be to check if the other training functions widely used in the field of artificial intelligence result in the same final dynamics or if the periodic behavior is a characteristic of the Adam algorithm.

Even with all this said, we feel that the results of this research could be useful in artificial intelligence. For example, we would like to see how designing a network with a periodic bias instead of a traditional fixed bias might help reduce the number of trained weights necessary to perform a given memory-dependent task. In commercial applications where networks can have millions of trainable weights, this could save massive amounts of computer power.

# 5 Acknowledgements

Thanks to Michele Nasoni for setting up the Jupyter server and rapidly answering all our computer related questions. And of course a big thank you to Guillaume Lajoie for supervising my summer internship, teaching me about the fascinating world of recurrent neural networks and suggesting lines of research when I was stuck.

# 6 Bibliography

## Reference manual

[1] S.H. Strogatz. *Nonlinear Dynamics And Chaos*. Studies in nonlinearity. Sarat Book House, 2007. ISBN: 9788187169857. URL: https://books.google.ca/books?id=PHmED2xxrE8C.

## Articles

[2] Rainer Engelken and Fred Wolf. "Dimensionality and entropy of spontaneous and evoked neural rate dynamics". Manuscript submitted for publication.

[3] Gregor M. Hoerzer, Robert Legenstein, and Wolfgang Maass. "Emergence of Complex Computational Structures From Chaotic Neural Networks Through Reward-Modulated Hebbian Learning". In: *CEREBRAL CORTEX* 24.3 (Mar. 2014), 677–690. ISSN: 1047-3211. DOI: {10.1093/cercor/bhs348}.

[4] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

[5] Rodrigo Laje and Dean V. Buonomano. "Robust timing and motor patterns by taming chaos in recurrent neural networks". In: *NATURE NEUROSCIENCE* 16.7 (July 2013), 925–U196. ISSN: 1097-6256. DOI: {10.1038/nn.3405}.

[6] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. "Deep learning". In: *NATURE* 521.7553 (May 2015), 436–444. ISSN: 0028-0836. DOI: {10.1038/nature14539}.

[7] Kanaka Rajan, Christopher D. Harvey, and David W. Tank. "Recurrent Network Models of Sequence Generation and Memory". In: *NEURON* 90.1 (Apr. 2016), 128–142. ISSN: 0896-6273. DOI: {10.1016/j.neuron.2016.02.009}.

[8] Jonathon Shlens. "A Tutorial on Principal Component Analysis". In: *CoRR* abs/1404.1100 (2014). arXiv: 1404.1100. URL: http://arxiv.org/abs/1404.1100.

[9] David Sussillo and L. F. Abbott. "Generating Coherent Patterns of Activity from Chaotic Neural Networks". In: *NEURON* 63.4 (Aug. 2009), 544–557. ISSN: 0896-6273. DOI: {10.1016/j.neuron.2009.07.018}.

[10] David Sussillo and Omri Barak. "Opening the Black Box: Low-Dimensional Dynamics in High-Dimensional Recurrent Neural Networks". In: *NEURAL COMPUTATION* 25.3 (Mar. 2013), 626–649. ISSN: 0899-7667. DOI: {10.1162/NECO_a_00409}.

[11] David Sussillo et al. "A neural network that finds a naturalistic solution for the production of muscle activity". In: *NATURE NEUROSCIENCE* 18.7 (July 2015), pp. 1025+. ISSN: 1097-6256. DOI: {10.1038/nn.4042}.

[12] Philippe Vincent-Lamarre, Guillaume Lajoie, and Jean-Philippe Thivierge. "Driving reservoir models with oscillations: a solution to the extreme structural sensitivity of chaotic networks". In: *JOURNAL OF COMPUTATIONAL NEUROSCIENCE* 41.3 (Dec. 2016), 305–322. ISSN: 0929-5313. DOI: {10.1007/s10827-016-0619-3}.

[13] Rafael Yuste. "From the neuron doctrine to neural networks". In: *NATURE REVIEWS NEUROSCIENCE* 16.8 (Aug. 2015), 487–497. ISSN: 1471-003X. DOI: {10.1038/nrn3962}.

## Web references

[14] Aude Forcione-Lambert. *An Investigation of Memory Implementation in Discrete Recurrent Neural Network*. https://github.com/aude-forcione-lambert/an-investigation-of-memory-implementation-in-discrete-recurrent-neural-network. (Accessed on 08/17/2018).

[15] *Low Level APIs — Introduction*. https://www.tensorflow.org/guide/low_level_intro. (Accessed on 08/17/2018).

[16] *Resources to learn Git*. https://try.github.io/. (Accessed on 08/17/2018).

[17] *sklearn.decomposition.PCA - scikit-learn 0.19.2 documentation*. http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html. (Accessed on 08/19/2018).

[18] *UdeM - TI - Réseau - VPN*. http://www.ti.umontreal.ca/reseau/vpn.html. (Accessed on 08/17/2018).

## Softwares

[19] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[20] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[21] PyTables Developers Team. *PyTables: Hierarchical Datasets in Python*. http://www.pytables.org/. 2002-2018.

[22] The HDF Group. *Hierarchical Data Format, version 5*. http://www.hdfgroup.org/HDF5/. 1997-2018.

# A 3-bit_flop_entrainement.py

Code for training one network in an interactive environment.

```
# # Importation des librairies et donn es d'entra nement

# Importations

# In[1]:


import tables
import numpy as np
import tensorflow as tf
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from mpl_toolkits.mplot3d import Axes3D
from cycler import cycler
from sklearn.decomposition import PCA


# In[2]:

get_ipython().run_line_magic('matplotlib', 'notebook')


# D fintion des variables d'entra nement

# In[3]:


nb_neurones = 100
backprop_len = 10
nb_batches = 300
nb_epochs = 15000


# Chargement des donn es pour entra nement et test

# In[4]:


input_file = tables.open_file('/opt/DATA/train_test_arrays.h5', mode='r')

table = input_file.root.test_array
x_test_array = table.read()['x']
y_test_array = table.read()['y']

train_array_table = input_file.root.train_array

if (backprop_len*nb_batches*nb_epochs)>train_array_table.nrows:
    print('Warning: Number of data requested larger than available data points. Train array
        will loop on available data.')

indexes = [[(i*nb_epochs*backprop_len+j*backprop_len)%(train_array_table.nrows-backprop_len)
     for i in range(nb_batches)] for j in range(nb_epochs)]


# # Entra nement du r seau

# In[5]:


random_seed = 0
time_delay = 0


```

```
# Architecture du r seau

# In[6]:


np.random.seed(random_seed)
win = tf.constant(np.random.uniform(-np.sqrt(1./3), np.sqrt(1./3), (3, nb_neurones)), dtype=
    tf.float64)
w = tf.Variable(np.random.uniform(-np.sqrt(1./nb_neurones), np.sqrt(1./nb_neurones), (
    nb_neurones, nb_neurones)), dtype=tf.float64)
wout = tf.Variable(np.random.uniform(-np.sqrt(1./nb_neurones), np.sqrt(1./nb_neurones), (
    nb_neurones, 3)), dtype=tf.float64)


# Architecture pour entra nement

# In[7]:


x_train = tf.placeholder(tf.float64, [backprop_len, nb_batches, 3])
y_train = tf.placeholder(tf.float64, [nb_batches, 3])
state_train_init = tf.placeholder(tf.float64, [nb_batches, nb_neurones])

x_series = tf.unstack(x_train, axis=0)
current_state_train = state_train_init
for current_input in x_series:
    current_input = tf.reshape(current_input, [nb_batches, 3])
    current_state_train = tf.reshape(current_state_train, [nb_batches, nb_neurones])
    next_state_train=tf.tanh(tf.matmul(current_state_train,w)+tf.matmul(current_input,win))
    current_state_train = next_state_train

pred_y_train = tf.tanh(tf.matmul(next_state_train,wout))


# In[8]:


loss = tf.reduce_sum(tf.square(tf.subtract(y_train,pred_y_train)))/nb_batches/3
train = tf.train.AdamOptimizer(learning_rate=0.005).minimize(loss)


# Architecture pour test

# In[9]:


x_test = tf.placeholder(tf.float64, [3,])
state_test_init = tf.placeholder(tf.float64, [1, nb_neurones])

current_input = tf.reshape(x_test, [1, 3])
current_state = tf.reshape(state_test_init, [1, nb_neurones])

next_state = tf.tanh(tf.matmul(current_state,w)+tf.matmul(current_input,win))
pred_y_test = tf.tanh(tf.matmul(next_state,wout))


# Initialisation du r seau

# In[10]:


sess = tf.Session()
sess.run(tf.global_variables_initializer())


# Entra nement du r seau

# In[11]:
```

```
128
129
130 rnn_state = np.random.rand(nb_batches, nb_neurones)
131 nb_small_loss = 0
132 loss_array = []
133
134 for epoch_index in range(nb_epochs):
135
136     _train = np.array([train_array_table.read(start=i,stop=i+backprop_len) for i in indexes[
        epoch_index]]).swapaxes(0,1)
137     _x_train = _train['x']
138     _y_train = _train['y'][backprop_len-time_delay-1,:]
139
140     _state, _loss, _train, _w, _wout = sess.run(
141         [current_state_train,loss,train,w,wout],
142         feed_dict = {
143             x_train : _x_train,
144             y_train : _y_train,
145             state_train_init : rnn_state
146         })
147
148     rnn_state = _state
149
150     if _loss < 0.01:
151         nb_small_loss += 1
152     else:
153         nb_small_loss = 0
154
155     if nb_small_loss >= 200:
156         break
157
158     loss_array.append(_loss)
159
160     if (epoch_index)%100 == 0:
161         print('Epoch '+str(epoch_index))
162
163 loss_array = np.array(loss_array)
164
165
166 # In[12]:
167
168
169 plt.figure()
170 plt.subplot(1,1,1)
171 plt.xlabel("Nombre d'epochs")
172 plt.ylabel("Erreur")
173 plt.plot(loss_array)
174
175
176 # Test du r seau
177
178 # In[13]:
179
180
181 rnn_state = np.random.rand(1,nb_neurones)
182 output_array_test = []
183 states_array_test = []
184
185 for i in range(len(x_test_array)):
186     _output, rnn_state, _w, _win, _wout = sess.run(
187             [pred_y_test, next_state, w, win, wout],
188             feed_dict = {
189                 x_test : x_test_array[i,:],
190                 state_test_init : rnn_state
191             })
192     output_array_test.append(_output[0])
193     states_array_test.append(rnn_state[0])
194
```

```
output_array_test = np.array(output_array_test)
states_array_test = np.array(states_array_test)


# Affichage des r sultats du test

# In[14]:


nb = 5000

fig, (ax1,ax2,ax3) = plt.subplots(3,1,figsize = (10,5))

ax1.plot(np.arange(nb),x_test_array[:nb,0],'r',linewidth=0.5)
ax1.plot(np.arange(time_delay,nb),y_test_array[:nb-time_delay,0],'b--',linewidth=1)
ax1.plot(np.arange(nb),output_array_test[:nb,0],'k',linewidth=1)
ax1.xaxis.grid()
ax1.set_ylabel("Channel 1")

ax2.plot(np.arange(nb),x_test_array[:nb,1],'r',linewidth=0.5)
ax2.plot(np.arange(time_delay,nb),y_test_array[:nb-time_delay,1],'b--',linewidth=1)
ax2.plot(np.arange(nb),output_array_test[:nb,1],'k',linewidth=1)
ax2.xaxis.grid()
ax2.set_ylabel("Channel 2")

ax3.plot(np.arange(nb),x_test_array[:nb,2],'r',linewidth=0.5)
ax3.plot(np.arange(time_delay,nb),y_test_array[:nb-time_delay,2],'b--',linewidth=1)
ax3.plot(np.arange(nb),output_array_test[:nb,2],'k',linewidth=1)
ax3.xaxis.grid()
ax3.set_ylabel("Channel 3")

plt.xlabel("temps (cycles)")
```

# B 3-bit_flop_analyse.py

Code for analyzing one network in an interactive environment.

```
# Importations

# In[1]:


import numpy as np
import scipy
from scipy import signal
from sklearn.decomposition import PCA
import pandas as pd

import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import matplotlib.animation as animation
from matplotlib.lines import Line2D
from mpl_toolkits.mplot3d import Axes3D
from cycler import cycler

from __future__ import print_function
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

import importlib


# In[2]:

```

```
29
30 np.random.seed(0)
31 colors = ('indianred','darkorange','gold','lime','turquoise','royalblue','blueviolet','
       magenta')
32 get_ipython().run_line_magic('matplotlib', 'notebook')
33
34
35 # In[3]:
36
37
38 from fcts_aff_anal import *
39
40
41 # Importations des donn es
42
43 # In[4]:
44
45
46 data_file = pd.HDFStore('/opt/DATA/train_test_arrays.h5').root.test_array.read()
47 x_test_array = data_file['x']
48 y_test_array = data_file['y']
49 test_array = pd.DataFrame({'x': x_test_array.tolist(), 'y': y_test_array.tolist()})
50
51
52 # In[5]:
53
54
55 random_seed = 7
56 time_delay = 8
57 nb_neurones = 100
58
59 data_file = pd.HDFStore('/opt/DATA/RNN.h5')
60 group = data_file.get_node('RNN_seed_'+str(random_seed)+'_delay_'+str(time_delay))
61
62 table = group.loss.read()
63 loss = table['loss']
64 table = group.training_steps.read()
65 w_train_array = table['w']
66 wout_train_array = table['wout']
67
68 table = group.testing_results.read()
69 states = table['states']
70 output = table['output']
71 test = pd.DataFrame({'states':states.tolist(), 'output':output.tolist()})
72
73 table = group.final_weights.read()
74 win = table['win'][0]
75 w = table['w'][0]
76 wout = table['wout'][0]
77
78 data_file.close()
79
80
81 # In[6]:
82
83
84 data_file = pd.HDFStore('/opt/DATA/analyse.h5')
85 data = data_file.root.data.read()
86 data_file.close()
87
88 min_q = data[np.where(np.all((data['random_seed']==random_seed,data['time_delay']==
       time_delay),axis=0))[0]]['min_q'][0]
89 q_type = data[np.where(np.all((data['random_seed']==random_seed,data['time_delay']==
       time_delay),axis=0))[0]]['q_type'][0]
90 q_type = q_type[np.where(np.any(min_q!=0,axis=1))[0]]
91 min_q = min_q[np.where(np.any(min_q!=0,axis=1))[0]]
92
93
```

```
# Analyse

# In[7]:


pca = PCA(n_components=3)
pca.fit(states)
pc_1, pc_2, pc_3 = pca.transform(states).T

out_type = np.array([np.argmax(tension_to_index(out)) for out in test['output']])
test['out_type'] = out_type


# In[8]:


center_array = []
stand_dev_array = []
for i in range(8):
    indexes = np.where(np.equal(out_type,i*np.ones(len(out_type))))[0]
    center, stand_dev = centerStandDev(states[indexes])
    center_array.append(center)
    stand_dev_array.append(stand_dev)
center_array = np.array(center_array)
stand_dev_array = np.array(stand_dev_array)


# In[9]:


vec = []
vec.append(center_array[np.argmax(tension_to_index((1,1,1)))]-center_array[np.argmax(
    tension_to_index((-1,1,1)))])
vec.append(center_array[np.argmax(tension_to_index((1,1,1)))]-center_array[np.argmax(
    tension_to_index((1,-1,1)))])
vec.append(center_array[np.argmax(tension_to_index((1,1,1)))]-center_array[np.argmax(
    tension_to_index((1,1,-1)))])
vec = np.array(vec)


# In[10]:


pca_all = PCA()
pca_all.fit(states)
cov_eigvals = pca_all.explained_variance_

np.sum(cov_eigvals)**2/np.sum(cov_eigvals**2)


# Dessin de l'analyse

# In[11]:


plotTrainResults(loss)


# In[12]:


min_q_pca = pca.transform(min_q).T
interact(plotPrincipalComponents, dots=False, lines=True, spheres=False, slowpts=True, vecs=
    True, nb=fixed(3000), pc_1=fixed(pc_1), pc_2=fixed(pc_2), pc_3=fixed(pc_3), out_type=
    fixed(out_type), min_q=fixed(min_q_pca), q_type=fixed(q_type), vecs_array=fixed(pca.
    transform(vec)), center_array=fixed(pca.transform(center_array)), stand_dev_array=fixed(
    pca.transform(stand_dev_array)))

```

```
155
156 # In[13]:
157
158
159 nb_before = time_delay+10
160 nb_after = 11
161 pattern = np.concatenate((np.ones(nb_before)*7,np.ones(nb_after)*3)).astype('int')
162 triggered_indexes = matchPattern(out_type,pattern)
163 triggered_indexes = np.array([np.arange(i,i+nb_before+nb_after) for i in triggered_indexes])
164 triggered_states = np.mean(states[triggered_indexes],axis=0)
165
166 triggered_pca = PCA(3)
167 triggered_pca.fit(triggered_states)
168 triggered_pc_1, triggered_pc_2, triggered_pc_3 = triggered_pca.transform(triggered_states).T
169
170
171 # In[14]:
172
173
174 plotPrincipalComponents(True,True,False,True,True,nb_before+nb_after, triggered_pc_1,
        triggered_pc_2, triggered_pc_3, pattern, triggered_pca.transform(min_q).T, q_type,
        triggered_pca.transform(vec))
175
176
177 # In[15]:
178
179
180 triggered_pc_1_2, triggered_pc_2_2, triggered_pc_3_2 = pca.transform(triggered_states).T
181 plotPrincipalComponents(True,True,False,True,True,nb_before+nb_after, triggered_pc_1_2,
        triggered_pc_2_2, triggered_pc_3_2, pattern, pca.transform(min_q).T, q_type, pca.
        transform(vec))
182
183
184 # In[16]:
185
186
187 triggered_pca_all = PCA()
188 triggered_pca_all.fit(triggered_states)
189
190 output_array = np.tanh(np.matmul(triggered_states,wout))
191 input_array = np.zeros(output_array.shape)
192 input_array[nb_before-time_delay] = (index_to_tension(pattern[-1])-index_to_tension(pattern
        [0]))/2
193 plotNeuronsActivity(triggered_pca_all.transform(triggered_states).T,input_array,output_array
        )
194
195
196 # Comparaison avec transformation de fourier
197
198 # In[17]:
199
200
201 T = 55
202 t = np.arange(-9,46)
203 y = np.concatenate((np.ones(5),-1*np.ones(50)))
204
205 n = np.arange(-50,50)
206 cn = np.array([np.sum(y*np.exp(-2j*np.pi*n_i/T*t))/T for n_i in n])
207
208 fig, (ax1,ax2,ax3) = plt.subplots(1,3)
209 ax1.plot(t,y)
210 ax2.plot(n,np.abs(cn))
211 ax3.plot(n,np.angle(cn))
212
213
214 # In[18]:
215
216
```

```
princ_comps = np.argsort(np.abs(cn))[-20:]
fd = np.array([cn[i]*np.exp(2j*np.pi*n[i]/T*t) for i in princ_comps]).real

fig, (ax1,ax2) = plt.subplots(1,2)
ax1.plot(t,fd.T)
ax2.plot(t,np.sum(fd, axis=0))


# In[19]:


fd = fd.T

pca_fourier = PCA()
pca_fourier.fit(fd[:15])
plotNeuronsActivity(pca_fourier.transform(fd[:15]).T,np.array([y,y,y]).T[:15],np.array([y,y,
    y]).T[:15])
```

# C 3-bit_flop_bulk_analyse.py

Code for comparing the features of different networks in an interactive environment.

```
# In[1]:


import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
from mpl_toolkits.mplot3d import Axes3D


# In[2]:


np.random.seed(0)
colors = ('indianred','darkorange','gold','lime','turquoise','royalblue','blueviolet','
    magenta')
get_ipython().run_line_magic('matplotlib', 'notebook')


# In[3]:


data_file = pd.HDFStore('/opt/DATA/train_test_arrays.h5').root.test_array.read()
x_test_array = data_file['x']
y_test_array = data_file['y']
test_array = pd.DataFrame({'x': x_test_array.tolist(), 'y': y_test_array.tolist()})


# In[4]:


data_file = pd.HDFStore('/opt/DATA/analyse.h5')
data = data_file.root.data.read()
data_file.close()


# In[5]:


dimension = data['dimension']
eigvals_imag = data['eigvals_imag']
eigvals_real = data['eigvals_real']
eigvals = eigvals_real+1j*eigvals_imag
```

```
min_q = data['min_q']
q_type = data['q_type']
random_seed = data['random_seed']
time_delay = data['time_delay']
converges = data['converges']
train_output = data['train_output']
max_cov_val = data['max_cov_val']
max_cov_delay = data['max_cov_delay']
anal_data = pd.DataFrame({'random_seed': random_seed.tolist(), 'time_delay': time_delay.tolist(),
                          'dimension': dimension.tolist(), 'min_q': min_q.tolist(), 'q_type': q_type.tolist(),
                          'eigvals': eigvals.tolist(), 'converges': converges.tolist(),
                          'train_output': train_output.tolist(), 'max_cov_val': max_cov_val.tolist(),
                          'max_cov_delay': max_cov_delay.tolist()})


# In[6]:


plt.figure()
plt.plot(anal_data[anal_data.converges].groupby(['time_delay']).mean()['dimension'],'.-')
plt.xlabel('D lais')
plt.ylabel('Dimension moyenne')


# In[7]:


mat = anal_data[anal_data.random_seed == 1]['train_output'].as_matrix()
mat = np.stack(mat)


# In[8]:


fig = plt.figure(figsize=(10,10))
ax = fig.add_subplot(111, projection='3d')
for j in range(10):
    for i in range(mat.shape[1]):
        if not np.all(mat[j,i,:,1]==0):
            ax.plot(np.ones(mat.shape[2])*i,np.arange(mat.shape[2]),mat[j,i,:,1]+(3*j),color=plt.cm.hsv(i/30))
ax.set_xlabel('tape d\'entrainement')
ax.set_ylabel('Temps')
ax.set_zlabel('D lais')


# In[9]:


corr_img = np.zeros((100*4,12,3))
for i in range(10):
    corr_img[0::4,i,0] = np.abs(anal_data[anal_data.random_seed==1][anal_data.time_delay==i]['max_cov_val'].as_matrix()[0][0])
    corr_img[1::4,i,1] = np.abs(anal_data[anal_data.random_seed==1][anal_data.time_delay==i]['max_cov_val'].as_matrix()[0][1])
    corr_img[2::4,i,2] = np.abs(anal_data[anal_data.random_seed==1][anal_data.time_delay==i]['max_cov_val'].as_matrix()[0][2])
    corr_img[3::4,i,:] = 1

data_file = pd.HDFStore('/opt/DATA/RNN.h5')
group = data_file.get_node('RNN_seed_1_delay_0')
table = group.final_weights.read()
win = table['win'][0]
data_file.close()

```

```
corr_img[:,10,:] = 1

corr_img[0::4,11,0] = win[0,:]/(np.max(abs(win)))
corr_img[1::4,11,1] = win[1,:]/(np.max(abs(win)))
corr_img[2::4,11,2] = win[2,:]/(np.max(abs(win)))
corr_img[3::4,11,:] = 1


# In[10]:


fig, ax = plt.subplots(figsize=(20,2))
plt.imshow(np.swapaxes(abs(corr_img),0,1),aspect='auto')
plt.title('Corr lation maximum')
plt.xlabel('# de neurone')
plt.ylabel('D lais de temps')
custom_legend = [
        Line2D([0], [0], color = 'r', label = 'Out 1'),
        Line2D([0], [0], color = 'g', label = 'Out 2'),
        Line2D([0], [0], color = 'b', label = 'Out 3'),
]
ax.legend(handles = custom_legend)
```

# D gen_io.py

Code for generating inputs and expected outputs for training and testing.

```
import tables
import numpy as np
np.random.seed(0)

class IODataPoint(tables.IsDescription):
    x = tables.Int64Col(3)
    y = tables.Int64Col(3)

def generateData(n,current_y,table):
    ptr = table.row
    input_indexes = np.random.poisson(50, (int(2.1*10**(n-2)),3) )
    for i in range(1,input_indexes.shape[0]):
        input_indexes[i,:] = input_indexes[i,:]+input_indexes[i-1,:]
    j=np.array([0,0,0])
    prev_ind = 0
    while prev_ind<10**n:
        next_ind = min(input_indexes[j,np.array([0,1,2])])
        for i in range(next_ind-prev_ind):
            ptr['x'] = [0,0,0]
            ptr['y'] = current_y
            ptr.append()
        mod_ind = np.equal(input_indexes[j,np.array([0,1,2])],next_ind)
        new_val = np.random.choice((-1,1),size=3)
        ptr['x'] = np.multiply(new_val,mod_ind)
        j += mod_ind
        current_y[np.where(mod_ind)] = new_val[np.where(mod_ind)]
        ptr['y'] = current_y
        ptr.append()
        prev_ind = next_ind
    table.flush
    return current_y

h5file = tables.open_file('/opt/DATA/train_test_arrays.h5', mode='w', title='data for
    training and testing 3-bit flop')

table = h5file.create_table('/', 'test_array', IODataPoint, 'array for testing 3-bit flop')
current_y = generateData(5,np.array([-1,-1,-1]),table)

```

```
table = h5file.create_table('/', 'train_array', IODataPoint, 'array for training 3-bit flop')
current_y = np.array([1,1,1])
for i in range(5):
    print(i)
    current_y = generateData(7,current_y,table)

h5file.close()
```

# E fcts_aff_anal.py

Useful functions for analysis and display.

```
# Importations

# In[1]:


import numpy as np
import scipy
from scipy import signal
from sklearn.decomposition import PCA
import pandas as pd

import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import matplotlib.animation as animation
from matplotlib.lines import Line2D
from mpl_toolkits.mplot3d import Axes3D
from cycler import cycler

from __future__ import print_function
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

import importlib


# In[2]:


np.random.seed(0)
colors = ('indianred','darkorange','gold','lime','turquoise','royalblue','blueviolet','magenta')
get_ipython().run_line_magic('matplotlib', 'notebook')


# In[3]:


from fcts_aff_anal import *


# Importations des donn es

# In[4]:


data_file = pd.HDFStore('/opt/DATA/train_test_arrays.h5').root.test_array.read()
x_test_array = data_file['x']
y_test_array = data_file['y']
test_array = pd.DataFrame({'x': x_test_array.tolist(), 'y': y_test_array.tolist()})


```

```
# In[5]:


random_seed = 7
time_delay = 8
nb_neurones = 100

data_file = pd.HDFStore('/opt/DATA/RNN.h5')
group = data_file.get_node('RNN_seed_'+str(random_seed)+'_delay_'+str(time_delay))

table = group.loss.read()
loss = table['loss']
table = group.training_steps.read()
w_train_array = table['w']
wout_train_array = table['wout']

table = group.testing_results.read()
states = table['states']
output = table['output']
test = pd.DataFrame({'states':states.tolist(), 'output':output.tolist()})

table = group.final_weights.read()
win = table['win'][0]
w = table['w'][0]
wout = table['wout'][0]

data_file.close()


# In[6]:


data_file = pd.HDFStore('/opt/DATA/analyse.h5')
data = data_file.root.data.read()
data_file.close()

min_q = data[np.where(np.all((data['random_seed']==random_seed,data['time_delay']==
    time_delay),axis=0))[0]]['min_q'][0]
q_type = data[np.where(np.all((data['random_seed']==random_seed,data['time_delay']==
    time_delay),axis=0))[0]]['q_type'][0]
q_type = q_type[np.where(np.any(min_q!=0,axis=1))[0]]
min_q = min_q[np.where(np.any(min_q!=0,axis=1))[0]]


# Analyse

# In[7]:


pca = PCA(n_components=3)
pca.fit(states)
pc_1, pc_2, pc_3 = pca.transform(states).T

out_type = np.array([np.argmax(tension_to_index(out)) for out in test['output']])
test['out_type'] = out_type


# In[8]:


center_array = []
stand_dev_array = []
for i in range(8):
    indexes = np.where(np.equal(out_type,i*np.ones(len(out_type))))[0]
    center, stand_dev = centerStandDev(states[indexes])
    center_array.append(center)
    stand_dev_array.append(stand_dev)
center_array = np.array(center_array)
```

```
stand_dev_array = np.array(stand_dev_array)


# In[9]:


vec = []
vec.append(center_array[np.argmax(tension_to_index((1,1,1)))]-center_array[np.argmax(
    tension_to_index((-1,1,1)))])
vec.append(center_array[np.argmax(tension_to_index((1,1,1)))]-center_array[np.argmax(
    tension_to_index((1,-1,1)))])
vec.append(center_array[np.argmax(tension_to_index((1,1,1)))]-center_array[np.argmax(
    tension_to_index((1,1,-1)))])
vec = np.array(vec)


# In[10]:


pca_all = PCA()
pca_all.fit(states)
cov_eigvals = pca_all.explained_variance_

np.sum(cov_eigvals)**2/np.sum(cov_eigvals**2)


# Dessin de l'analyse

# In[11]:


plotTrainResults(loss)


# In[12]:


min_q_pca = pca.transform(min_q).T
interact(plotPrincipalComponents, dots=False, lines=True, spheres=False, slowpts=True, vecs=
    True, nb=fixed(3000), pc_1=fixed(pc_1), pc_2=fixed(pc_2), pc_3=fixed(pc_3), out_type=
    fixed(out_type), min_q=fixed(min_q_pca), q_type=fixed(q_type), vecs_array=fixed(pca.
    transform(vec)), center_array=fixed(pca.transform(center_array)), stand_dev_array=fixed(
    pca.transform(stand_dev_array)))


# In[13]:


nb_before = time_delay+10
nb_after = 11
pattern = np.concatenate((np.ones(nb_before)*7,np.ones(nb_after)*3)).astype('int')
triggered_indexes = matchPattern(out_type,pattern)
triggered_indexes = np.array([np.arange(i,i+nb_before+nb_after) for i in triggered_indexes])
triggered_states = np.mean(states[triggered_indexes],axis=0)

triggered_pca = PCA(3)
triggered_pca.fit(triggered_states)
triggered_pc_1, triggered_pc_2, triggered_pc_3 = triggered_pca.transform(triggered_states).T


# In[14]:


plotPrincipalComponents(True,True,False,True,True,nb_before+nb_after, triggered_pc_1,
    triggered_pc_2, triggered_pc_3, pattern, triggered_pca.transform(min_q).T, q_type,
    triggered_pca.transform(vec))


```

```
# In[15]:


triggered_pc_1_2, triggered_pc_2_2, triggered_pc_3_2 = pca.transform(triggered_states).T
plotPrincipalComponents(True,True,False,True,True,nb_before+nb_after, triggered_pc_1_2,
    triggered_pc_2_2, triggered_pc_3_2, pattern, pca.transform(min_q).T, q_type, pca.
    transform(vec))


# In[16]:


triggered_pca_all = PCA()
triggered_pca_all.fit(triggered_states)

output_array = np.tanh(np.matmul(triggered_states,wout))
input_array = np.zeros(output_array.shape)
input_array[nb_before-time_delay] = (index_to_tension(pattern[-1])-index_to_tension(pattern
    [0]))/2
plotNeuronsActivity(triggered_pca_all.transform(triggered_states).T,input_array,output_array
    )


# Comparaison avec transformation de fourier

# In[17]:


T = 55
t = np.arange(-9,46)
y = np.concatenate((np.ones(5),-1*np.ones(50)))

n = np.arange(-50,50)
cn = np.array([np.sum(y*np.exp(-2j*np.pi*n_i/T*t))/T for n_i in n])

fig, (ax1,ax2,ax3) = plt.subplots(1,3)
ax1.plot(t,y)
ax2.plot(n,np.abs(cn))
ax3.plot(n,np.angle(cn))


# In[18]:


princ_comps = np.argsort(np.abs(cn))[-20:]
fd = np.array([cn[i]*np.exp(2j*np.pi*n[i]/T*t) for i in princ_comps]).real

fig, (ax1,ax2) = plt.subplots(1,2)
ax1.plot(t,fd.T)
ax2.plot(t,np.sum(fd, axis=0))


# In[19]:


fd = fd.T

pca_fourier = PCA()
pca_fourier.fit(fd[:15])
plotNeuronsActivity(pca_fourier.transform(fd[:15]).T,np.array([y,y,y]).T[:15],np.array([y,y,
    y]).T[:15])
```

# F 3-bit_flop_entrainement_rapide.py

Code for rapidly training many networks in the background.

```
# # Importation des librairies et donn es d'entra nement

# Importations

import tables
import numpy as np
import tensorflow as tf
import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
from mpl_toolkits.mplot3d import Axes3D
from cycler import cycler
from sklearn.decomposition import PCA


# D fintion des variables d'entra nement

nb_neurones = 100
backprop_len = 10
nb_batches = 300
nb_epochs = 15000

class TrainDataPoint(tables.IsDescription):
    w = tables.Float64Col((nb_neurones,nb_neurones))
    wout = tables.Float64Col((nb_neurones,3))

class LossDataPoint(tables.IsDescription):
    loss = tables.Float64Col()

class TestDataPoint(tables.IsDescription):
    states = tables.Float64Col(nb_neurones)
    output = tables.Float64Col(3)

class FinalWeights(tables.IsDescription):
    win = tables.Float64Col((3,nb_neurones))
    w = tables.Float64Col((nb_neurones,nb_neurones))
    wout = tables.Float64Col((nb_neurones,3))


# Chargement des donn es pour entra nement et test

input_file = tables.open_file('/opt/DATA/train_test_arrays.h5', mode='r')

table = input_file.root.test_array
x_test_array = table.read()['x']
y_test_array = table.read()['y']

train_array_table = input_file.root.train_array

if (backprop_len*nb_batches*nb_epochs)>train_array_table.nrows:
    print('Warning: Number of data requested larger than available data points. Train array
        will loop on available data.')

indexes = [[(i*nb_epochs*backprop_len+j*backprop_len)%(train_array_table.nrows-backprop_len)
     for i in range(nb_batches)] for j in range(nb_epochs)]


# # Entra nement du r seau

for time_delay in range(10):
    for random_seed in range(10):

        print('/RNN_seed_'+str(random_seed)+'_delay_'+str(time_delay))

        # Ouverture du fichier pour sauvegarde des r sultats

        # Uncomment to append to existing file
```

```
        output_file = tables.open_file('/opt/DATA/RNN.h5', mode='a')
        # Uncomment to override existing file
        #output_file = tables.open_file('/opt/DATA/RNN.h5', mode='w', title='RNN training
            progression, test data and final weights')

        if '/RNN_seed_'+str(random_seed)+'_delay_'+str(time_delay) in output_file:
            output_file.remove_node('/RNN_seed_'+str(random_seed)+'_delay_'+str(time_delay),
                recursive=True)

        group = output_file.create_group('/', 'RNN_seed_'+str(random_seed)+'_delay_'+str(
            time_delay), 'Training and testing data for RNN with random seed='+str(
            random_seed)+' and delay='+str(time_delay))
        group._f_setattr('random_seed',random_seed)
        group._f_setattr('time_delay',time_delay)
        group._f_setattr('nb_neurones',nb_neurones)
        group._f_setattr('backprop_len',backprop_len)
        group._f_setattr('nb_batches',nb_batches)
        group._f_setattr('nb_epochs',nb_epochs)


        # Architecture du r seau

        np.random.seed(random_seed)
        win = tf.constant(np.random.uniform(-np.sqrt(1./3), np.sqrt(1./3), (3, nb_neurones))
            , dtype=tf.float64)
        w = tf.Variable(np.random.uniform(-np.sqrt(1./nb_neurones), np.sqrt(1./nb_neurones),
             (nb_neurones, nb_neurones)), dtype=tf.float64)
        wout = tf.Variable(np.random.uniform(-np.sqrt(1./nb_neurones), np.sqrt(1./
            nb_neurones), (nb_neurones, 3)), dtype=tf.float64)


        # Architecture pour entra nement

        x_train = tf.placeholder(tf.float64, [backprop_len, nb_batches, 3])
        y_train = tf.placeholder(tf.float64, [nb_batches, 3])
        state_train_init = tf.placeholder(tf.float64, [nb_batches, nb_neurones])
        momentum = tf.placeholder(tf.float64)

        x_series = tf.unstack(x_train, axis=0)
        current_state_train = state_train_init
        for current_input in x_series:
            current_input = tf.reshape(current_input, [nb_batches, 3])
            current_state_train = tf.reshape(current_state_train, [nb_batches, nb_neurones])
            next_state_train=tf.tanh(tf.matmul(current_state_train,w)+tf.matmul(
                current_input,win))
            current_state_train = next_state_train

        pred_y_train = tf.tanh(tf.matmul(next_state_train,wout))


        loss = tf.reduce_sum(tf.square(tf.subtract(y_train,pred_y_train)))/nb_batches/3
        train = tf.train.AdamOptimizer(learning_rate=0.0005).minimize(loss)


        # Architecture pour test

        x_test = tf.placeholder(tf.float64, [3,])
        state_test_init = tf.placeholder(tf.float64, [1, nb_neurones])

        current_input = tf.reshape(x_test, [1, 3])
        current_state = tf.reshape(state_test_init, [1, nb_neurones])

        next_state = tf.tanh(tf.matmul(current_state,w)+tf.matmul(current_input,win))
        pred_y_test = tf.tanh(tf.matmul(next_state,wout))


        # Initialisation du r seau

```

```
126         sess = tf.Session()
127         sess.run(tf.global_variables_initializer())
128
129
130         # Entra nement du r seau
131
132         train_table = output_file.create_table(group, 'training_steps', TrainDataPoint, '
                progression of RNN weights during training')
133         train_ptr = train_table.row
134
135         loss_table = output_file.create_table(group, 'loss', LossDataPoint, 'progression of
                loss during training')
136         loss_ptr = loss_table.row
137
138         rnn_state = np.random.rand(nb_batches, nb_neurones)
139         nb_small_loss = 0
140
141         for epoch_index in range(nb_epochs):
142
143             _train = np.array([train_array_table.read(start=i,stop=i+backprop_len) for i in
                    indexes[epoch_index]]).swapaxes(0,1)
144             _x_train = _train['x']
145             _y_train = _train['y'][backprop_len-time_delay-1,:]
146
147             _state, _loss, _train, _w, _wout = sess.run(
148                 [current_state_train,loss,train,w,wout],
149                 feed_dict = {
150                     x_train : _x_train,
151                     y_train : _y_train,
152                     state_train_init : rnn_state
153                 })
154
155             rnn_state = _state
156
157             last_loss = _loss
158
159             if _loss < 0.02:
160                 nb_small_loss += 1
161             else:
162                 nb_small_loss = 0
163
164             if epoch_index>5000 and nb_small_loss >= 200:
165                 break
166
167             loss_ptr['loss'] = _loss
168             loss_ptr.append()
169             loss_table.flush()
170
171             if (epoch_index)%500 == 0:
172                 train_ptr['w'] = _w
173                 train_ptr['wout'] = _wout
174                 train_ptr.append()
175                 train_table.flush()
176                 print(str(epoch_index))
177
178         # Test du r seau
179
180         test_table = output_file.create_table(group, 'testing_results', TestDataPoint, '
                results of testing')
181         test_ptr = test_table.row
182         weights_table = output_file.create_table(group, 'final_weights', FinalWeights, '
                final state of RNN')
183         weights_ptr = weights_table.row
184         rnn_state = np.random.rand(1,nb_neurones)
185
186         for i in range(len(x_test_array)):
187             _output, rnn_state, _w, _win, _wout = sess.run(
188
```

```
                    [pred_y_test, next_state, w, win, wout],
                    feed_dict = {
                        x_test : x_test_array[i,:],
                        state_test_init : rnn_state
                    })
            test_ptr['output'] = _output
            test_ptr['states'] = rnn_state
            test_ptr.append()

        weights_ptr['win'] = _win
        weights_ptr['w'] = _w
        weights_ptr['wout'] = _wout
        weights_ptr.append()

        test_table.flush()
        weights_table.flush()

        output_file.close()

# Fermeture du fichier des r sultats

input_file.close()
```

## G 3-bit_flop_analyse_rapide.py

Code for rapidly analyzing many networks in the background.

```
# Importations

import numpy as np
import scipy
from scipy import signal
from sklearn.decomposition import PCA
import pandas as pd
import tables

import matplotlib as mpl
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import matplotlib.animation as animation
from matplotlib.lines import Line2D
from mpl_toolkits.mplot3d import Axes3D
from cycler import cycler

from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets

np.random.seed(0)

nb_neurones = 100


# Analyse

def tension_to_index(tension):
    index = np.zeros((8))
    tension = np.round(tension)
    index[int((tension[0]+1)/2+(tension[1]+1)+2*(tension[2]+1))] = 1
    return index

def index_to_tension(index):
    tension = np.zeros((3))
    tension[0] = (index%2)*2-1
    tension[1] = ((index >> 1)%2)*2-1
    tension[2] = ((index >> 2)%2)*2-1
```

```
39      return tension
40
41  def F(state):
42      return np.tanh(np.matmul(state,w))
43
44  I = np.diag(np.ones(nb_neurones))
45  def q(state):
46      return 1/2.0*np.linalg.norm(F(state)-state)**2
47
48  def centerStandDev(x):
49      center = np.mean(x,axis=1)
50      stand_dev = np.std(x,axis=1)
51      return center,stand_dev
52
53  def calculateSelectedOutput(selected_neurons):
54      s_win = win[:,selected_neurons]
55      s_w = w[selected_neurons,:][:,selected_neurons]
56      s_wout = wout[selected_neurons,:]
57
58      selected_out = []
59      prev_state = np.random.rand(len(selected_neurons))
60      next_state = np.empty(len(selected_neurons))
61
62      for x in x_test_array:
63          next_state = np.tanh(np.matmul(x,s_win)+np.matmul(prev_state,s_w))
64          selected_out.append(np.tanh(np.matmul(next_state,s_wout)))
65          prev_state = next_state
66
67      selected_out = np.array(selected_out)
68      return selected_out
69
70  def calculateShortOutput(s_w,s_wout,nb_inputs):
71      selected_out = []
72      prev_state = np.random.rand(nb_neurones)
73      next_state = np.empty(nb_neurones)
74
75      for x in x_test_array[0:nb_inputs]:
76          next_state = np.tanh(np.matmul(x,win)+np.matmul(prev_state,s_w))
77          selected_out.append(np.tanh(np.matmul(next_state,s_wout)))
78          prev_state = next_state
79
80      selected_out = np.array(selected_out)
81      return selected_out
82
83  def cov(tau,f,g):
84      return np.array([np.average(f*g) if t==0 else np.average(f[:-t]*g[t:]) for t in tau])
85
86
87  class analDataPoint(tables.IsDescription):
88      random_seed = tables.Int8Col()
89      time_delay = tables.Int8Col()
90      min_q = tables.Float64Col((54,nb_neurones))
91      q_type = tables.Int64Col(54)
92      dimension = tables.Float64Col()
93      eigvals_real = tables.Float64Col(nb_neurones)
94      eigvals_imag = tables.Float64Col(nb_neurones)
95      converges = tables.BoolCol()
96      train_output = tables.Float64Col((31,500,3))
97      max_cov_val = tables.Float64Col((3,nb_neurones))
98      max_cov_delay = tables.Int8Col((3,nb_neurones))
99
100
101 output_file = tables.open_file('/opt/DATA/analyse.h5', mode='w', title='Analyse des r seaux
        de neurones')
102 anal_table = output_file.create_table('/','data',analDataPoint,'data')
103 ptr = anal_table.row
104
105
```

```
# Importations des donn es

data_file = pd.HDFStore('/opt/DATA/train_test_arrays.h5').root.test_array.read()
x_test_array = data_file['x']
y_test_array = data_file['y']

for random_seed in range(10):
    for time_delay in range(10):

        print('seed: '+str(random_seed)+' delay: '+str(time_delay))

        ptr['random_seed'] = random_seed
        ptr['time_delay'] = time_delay

        data_file = pd.HDFStore('/opt/DATA/RNN.h5')
        group = data_file.get_node('RNN_seed_'+str(random_seed)+'_delay_'+str(time_delay))

        table = group.loss.read()
        loss = table['loss']
        table = group.training_steps.read()
        w_train_array = table['w']
        wout_train_array = table['wout']

        table = group.testing_results.read()
        states = table['states']
        output = table['output']

        table = group.final_weights.read()
        win = table['win'][0]
        w = table['w'][0]
        wout = table['wout'][0]

        data_file.close()

        ptr['converges'] = loss[-1]<0.02

        pca = PCA(n_components=3)
        pca.fit(states)
        x, y, z = pca.transform(states).T

        out_type = np.array([np.argmax(tension_to_index(out)) for out in output])
        center_array = []
        stand_dev_array = []
        for i in range(8):
            indexes = np.where(np.equal(out_type,i*np.ones(len(out_type))))[0]
            center, stand_dev = centerStandDev(np.stack((x[indexes],y[indexes],z[indexes])))
            center_array.append(center)
            stand_dev_array.append(stand_dev)
        center_array = np.array(center_array)
        stand_dev_array = np.array(stand_dev_array)

        max_state_val = max(np.sum(abs(states),axis=1))

        min_q = []
        for i in range(500):
            opt = scipy.optimize.minimize(q,(np.random.rand(nb_neurones)-0.5)*4*
                max_state_val,options={'maxiter':300})
            if opt['success'] == True:
                if len(min_q)==0 or not np.any(np.all((min_q-opt['x'])<0.01,axis=1)):
                    min_q.append(opt['x'])
        min_q = np.array(min_q)

        q_type = []
        for q_i in min_q:
            J = np.empty(w.shape)
            for i in range(nb_neurones):
                J[i,:] = np.dot(w[i,:],1/np.cosh(np.matmul(w[i,:],q_i))**2)-I[i,:]
            q_type.append(sum(np.linalg.eig(J)[0]>0))
```

```
        q_type = np.array(q_type,copy=True)

        min_q = np.copy(min_q)
        min_q.resize((54,nb_neurones))
        q_type.resize(54)
        ptr['min_q'] = min_q
        ptr['q_type'] = q_type

        pca_all = PCA()
        pca_all.fit(states)
        cov_eigvals = pca_all.explained_variance_

        ptr['dimension'] = np.sum(cov_eigvals)**2/np.sum(cov_eigvals**2)

        x_sp, y_sp, z_sp = pca.transform(min_q).T

        eigvals, eigvecs = np.linalg.eig(w)
        eigvals_train = []

        for i in range(w_train_array.shape[0]):
            eigvals_i, eigvecs_i = np.linalg.eig(w_train_array[i,:,:])
            eigvals_train.append(eigvals_i)
        eigvals_train = np.array(eigvals_train)

        eigvals, eigvecs = np.linalg.eig(w)

        ptr['eigvals_real'] = eigvals.real
        ptr['eigvals_imag'] = eigvals.imag

        train_output = np.zeros((31,500,3))
        i = 0
        for w_train,wout_train in zip(w_train_array,wout_train_array):
            train_output[i,:,:] = calculateShortOutput(w_train,wout_train,500)
            i += 1
        train_output[i,:,:] = calculateShortOutput(w,wout,500)

        ptr['train_output'] = train_output


        max_cov_val = np.empty((3,nb_neurones))
        max_cov_delay = np.empty((3,nb_neurones))

        for j in range(3):
            for i,s in enumerate(states.T):
                cov = signal.correlate(s,y_test_array[:,j],mode='full')/(s.std()*
                    y_test_array[:,j].std()*s.size)
                max_cov_val[j,i] = cov[np.argmax(abs(cov))]
                max_cov_delay[j,i] = np.argmax(abs(cov))-(cov.size//2)


        ptr['max_cov_val'] = max_cov_val
        ptr['max_cov_delay'] = max_cov_delay


        ptr.append()

output_file.close()
```