ReScience

[Re] Robust timing and motor patterns by taming chaos in recurrent neural networks

Julien Vitay[1]

1 Professorship for Artificial Intelligence, Department of Computer Science, Chemnitz University of Technology, D-09107 Chemnitz, Germany

julien.vitay@informatik.tu-chemnitz.de

Editor
Nicolas P. Rougier

Reviewers
Xavier Hinaut
Pierre Enel

Received Jul, 1, 2016
Accepted Oct, 7, 2016
Published Oct, 7, 2016

Competing Interests:
The authors have declared that no competing interests exist.

A reference implementation of

→ Laje, R. and Buonomano, D.V. (2013). Robust timing and motor patterns by taming chaos in recurrent neural networks. Nat Neurosci. 16(7) pp 925-33. doi:10.1038/nn.3405.

Introduction

Recurrent neural networks have the capacity to exhibit complex spatio-temporal patterns that can be used to produce motor patterns. The research field of *reservoir computing* [2, 4] focuses on forcing recurrent networks to produce a desired read-out output through supervised learning. In the high-gain regime, recurrent networks can even exhibit rich and complex patterns in the absence of stimulation, but they become chaotic, as they can not reproduce twice the same trajectory in the presence of noise [5]. In Laje and Buonomano [3], the authors proposed a method to *tame* the chaos in such networks, by training the recurrent weights to minimize the error between a desired trajectory - spontaneously generated by the network in an initial trial - and the current trajectory. By selectively creating stable dynamic attractors, this method allows to benefit at the same time from the rich dynamics of a chaotic network and from the reproducibility of a deterministic one.

Methods

The network proposed by Laje and Buonomano [3] is a pool of 800 rate-coded neurons sparsely connected with each other with a fixed probability. They receive random inputs from specific impulse neurons, which are only used to shortly force the network into a deterministic state at the beginning of a trial or to perturb its state. One or more output neurons (called *read-out* neurons) receive inputs from the recurrent units and learn to produce a specific target activation through supervised learning. 60% of the recurrent weights are plastic and evolve according to the recursive least squares (RLS) algorithm [1], as well as all the read-out weights between the recurrent units and the read-out ones.

The training procedure is split into three separate phases: 1) an initial trajectory is acquired by giving an input impulse and recording the activation of all recurrent units for a given duration *in the absence of noise*, 2) the recurrent weights are trained for 20 or 30 trials to reproduce this initial trajectory using RLS in the presence of noise,

3) the read-out weights are trained for 10 trials to generate a desired pattern (e.g. a delayed impulse, see the Results section), using the recurrent units as inputs.

The Python code for the network simulation and training is provided in the file `RecurrentNetwork.py`. It is based on both the original article and the Matlab code provided by the authors as supplementary material[1]. Contrary to the original code where the network is defined as a script (obligatory with Matlab), the network is here implemented as a Python class, allowing to reuse the same code when some parameters change between two simulations (e.g. the number of output neurons). The computations are mostly standard linear algebra operations on vectors and matrices, so the network is simply defined by a set of Numpy arrays stored as attributes. This class is then instantiated by each script reproducing the figures (`Fig1.py`, `Fig2.py`, `Fig3.py`), with different inputs and/or training procedures. The rest of this section describes the network in more details and presents the implementation.

Structure of the network

Recurrent network

The recurrent network is composed of $N = 800$ neurons, receiving inputs from a variable number of input neurons N_i and sparsely connected with each other. Each neuron's firing rate $r_i(t)$ applies the tanh transfer function on an internal variable $x_i(t)$ which follows a first-order linear ordinary differential equation (ODE):

$$\tau \cdot \frac{dx_i(t)}{dt} + x_i(t) = \sum_{j=1}^{N_i} W_{ij}^{in} \cdot y_j(t) + \sum_{j=1}^{N} W_{ij}^{rec} \cdot r_j(t) + I_i^{noise}(t)$$

$$r_i(t) = \tanh(x_i(t))$$

The weights of the input matrix W^{in} are taken from the normal distribution, with mean 0 and variance 1, and multiply the rates of the input neurons $y_j(t)$. The variance of these weights does not depend on the number of inputs, as only one input neuron is activated at the same time. The recurrent connectivity matrix W^{rec} is sparse with a connection probability $pc = 0.1$ (i.e. 64000 non-zero elements) and existing weights are taken randomly from a normal distribution with mean 0 and variance $g/\sqrt{pc \cdot N}$, where g is a scaling factor. It is a well known result for sparse recurrent networks that for high values of g, the network dynamics become chaotic. $I_i^{noise}(t)$ is an additive noise, taken randomly at each time step and for each neuron from a normal distribution with mean 0 and variance I_0. I_0 is chosen very small in the experiments reproduced here ($I_0 = 0.001$) but is enough to highlight the chaotic behavior of the recurrent neurons (non-reproducibility between two trials).

The `build()` method of the `Network` class initializes all these elements as Numpy arrays and stores them as attributes. For clarity, we present the code here as a regular script:

```
# Input
I = np.zeros((Ni, 1))

# Recurrent population
x = np.random.uniform(-1.0, 1.0, (N, 1))
r = np.tanh(x)
```

[1] http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3753043

```
# Weights between the input and recurrent units
W_in = np.random.randn(N, Ni)

# Weights between the recurrent units
W_rec = np.random.randn(N, N) * g/np.sqrt(pc*N)

# The connection pattern is sparse with p=0.1
connectivity_mask = np.random.binomial(1, pc, (N, N))
connectivity_mask[np.diag_indices(N)] = 0
W_rec *= connectivity_mask
```

Everything is straightforward here: population variables are vertical vectors and weights are represented by matrices. They are initialized using Numpy's random library. In order to ensure the sparseness of the recurrent weights. `connectivity_mask` is a binary mask of size $N * N$ with ones where synapses should exist and zeros otherwise, generated using the binomial distribution. The diagonal is removed to avoid self-connections. The weight matrix is a regular Numpy matrix here, but the `scipy.sparse` library could have been used instead.

To perform the simulation, the ODE of the recurrent neurons is discretized using the forward Euler method, with an implicit time step dt of 1 ms:

```
for t in range(duration):
    x += (np.dot(W_in, I) + np.dot(W_rec, r)
            + Io * np.random.randn(N, 1) - x)/tau
    r = np.tanh(x)
```

Read-out neurons

The read-out neurons simply sum the activity of the recurrent neurons using a matrix W^{out}:

$$z_i(t) = \sum_{j=1}^{N} W_{ij}^{out} \cdot r_j(t)$$

The read-out matrix is initialized randomly from the normal distribution with mean 0 and variance $1/\sqrt{N}$:

```
# Read-out population
z = np.zeros((No, 1))
# Output weights
W_out = np.random.randn(No, N) / np.sqrt(N)
```

The firing rate of the read-out neurons is simply the product between the weight matrix and the rates of the recurrent neurons at each time step:

```
z = np.dot(W_out, r)
```

Learning rule

Recursive least squares (RLS)

The particularity of the reservoir network proposed by Laje and Buonomano [3] is that both the recurrent weights W^{rec} and the read-out weights are trained in a supervised manner. More precisely, in this implementation, only 60% of the recurrent neurons have plastic weights, the 40% others keep the same weights throughout the simulation.

Learning is done using the recursive least squares (RLS) algorithm [1]. It is a supervised error-driven learning rule, i.e. the weight changes depend on the error made by each neuron: the difference between the firing rate of a neuron $r_i(t)$ and a desired value $R_i(t)$.

$$e_i(t) = r_i(t) - R_i(t)$$

For the recurrent neurons, the desired value is the recorded rate of that neuron during an initial trial (to enforce the reproducibility of the trajectory). For the read-out neurons, it is a function which we want the network to reproduce (e.g. handwriting as in Fig. 2).

Contrary to the delta learning rule which modifies weights proportionally to the error and to the direct input to a synapse ($\Delta w_{ij} = -\eta \cdot e_i \cdot r_j$), the RLS learning uses a running estimate of the inverse correlation matrix of the inputs to each neuron:

$$\Delta w_{ij} = -e_i \sum_{k \in \mathcal{B}(i)} P^i_{jk} \cdot r_k$$

Each neuron i therefore stores a square matrix P^i, whose size depends of the number of weights arriving to the neuron. Read-out neurons receive synapses from all N recurrent neurons, so the P matrix is $N * N$. Recurrent units have a sparse random connectivity ($pc = 0.1$), so each recurrent neuron stores only a $80 * 80$ matrix on average. In the previous equation, $\mathcal{B}(i)$ represents those existing weights.

The inverse correlation matrix P is updated at each time step with the following rule:

$$\Delta P^i_{jk} = -\frac{\sum_{m \in \mathcal{B}(i)} \sum_{n \in \mathcal{B}(i)} P^i_{jm} \cdot r_m \cdot r_n \cdot P^i_{nk}}{1 + \sum_{m \in \mathcal{B}(i)} \sum_{n \in \mathcal{B}(i)} r_m \cdot P^i_{mn} \cdot r_n}$$

Each matrix P^i is initialized to the diagonal matrix and scaled by a factor $1/\delta$, where δ is 1 in the current implementation and can be used to modify implicitly the learning rate [6].

Implementation

Linear algebra: For an efficient implementation in Python, the previous weight-specific update rules have to be turned into matrix/vector operations. Unfortunately, for the recurrent units, each matrix P^i has a different size ($80 * 80$ on average), so we will still need to iterate over all post-synaptic neurons. If we note $\mathbf{W}$ the vector of weights coming to a neuron (80 on average), $\mathbf{r}$ the corresponding vector of firing rates (also 80), e the error of that neuron (a scalar) and $\mathbf{P}$ the inverse correlation matrix (80*80), the update rules become:

$$\Delta \mathbf{W} = -e \cdot \mathbf{P} \cdot \mathbf{r}$$

$$\Delta \mathbf{P} = -\frac{(\mathbf{P} \cdot \mathbf{r}) \cdot (\mathbf{P} \cdot \mathbf{r})^T}{1 + \mathbf{r}^T \cdot \mathbf{P} \cdot \mathbf{r}}$$

In Haykin [1], it is shown that $\mathbf{P}$ converges towards the inverse correlation matrix of the inputs to the neuron, plus a regularization term:

$$\mathbf{P} = (\sum_t \mathbf{r}(t) \cdot \mathbf{r}^T(t) - \delta \mathbf{I})^{-1}$$

However, by looking at the original Matlab code, one notices that the weight update $\Delta\mathbf{W}$ is also normalized by the denominator of the update rule for $\mathbf{P}$:

$$\Delta\mathbf{W} = -e \cdot \frac{\mathbf{P} \cdot \mathbf{r}}{1 + \mathbf{r}^T \cdot \mathbf{P} \cdot \mathbf{r}}$$

Removing this normalization from the learning rule impairs learning completely, so we kept this variant of the RLS rule in our implementation.

Initialization: The RLS rule is applied to both recurrent and read-out weights. For the recurrent neurons, we need to first build a list of the actual inputs to each neuron:

```
N_plastic = int(0.6*N)
W_plastic = [list(np.nonzero(connectivity_mask[i, :])[0])
                              for i in range(N_plastic)]
```

Here, only 60% of the recurrent neurons have plastic synapses, so we restrict learning to the first 480 neurons (the exact indexes are irrelevant). We use the non-zero elements of `connectivity_mask` to find out which synapses actually exist.

We can then initialize the P matrices for each recurrent neuron, depending on how many weights they receive:

```
P = [np.identity(len(W_plastic[i]))/delta for i in range(N_plastic)]
```

Initializing the P matrix for the read-out neurons is easier, as they all receive exactly N weights from the recurrent neurons:

```
P_out = [np.identity(N)/delta for i in range(No)]
```

Training: Once the P matrices are initialized to the correct value, implementing the learning rule only consists of getting the algebraic operations right. One particularity of the original Matlab code is that the learning rules are only applied every 2 time steps during the training window instead of every step. As the computations are heavy, we kept this implementation trick which does not change much the results.

For the recurrent weights, we need to build a vector `r_plastic` for each neuron that contains the firing rates of the neurons connected to it. Once we have this vector, we can apply the RLS rule. The error is computed as a vector of shape $(800, 1)$ (one scalar value per recurrent unit), but in practice only used for the first `N_plastic` units:

```
# Compute the error of the recurrent neurons
error = r - target
```

```
# Apply the RLS learning rule to the recurrent weights
for i in range(N_plastic): # for each plastic post neuron
    # Get the rates from the plastic synapses only
    r_plastic = r[W_plastic[i]]
    # Multiply with the inverse correlation matrix P*R
    PxR = np.dot(P[i], r_plastic)
    # Normalization term 1 + R'*P*R
    RxPxR = (1. + np.dot(r_plastic.T,  PxR))
    # Update the inverse correlation matrix
    # P <- P - ((P*R)*(P*R)')/(1+R'*P*R)
    P[i] -= np.dot(PxR, PxR.T)/RxPxR
    # Learning rule W <- W - e * (P*R)/(1+R'*P*R)
    W_rec[i, W_plastic[i]] -= error[i, 0] * (PxR/RxPxR)[:, 0]
```

The weight update uses the error of the neuron `i` (a scalar) and a vector corresponding to `(P*R)/(1+R'*P*R)`. The shape of `W_rec[i, W_plastic[i]]` is (80,) on average, but `PxR/RxPxR` is (80, 1), so we need to slice this array to the correct shape using `[:, 0]`. The implementation for the read-out weights is similar, except we can use directly `r` instead of `r_plastic` in the update rules.

Training procedure

The training procedure is split into different trials, which differ from one experiment to another (Figs 1, 2 and 3). Each trial begins with a relaxation period of `t_offset` = 200 ms, followed by a brief input impulse of duration 50 ms and variable amplitude. This impulse has the effect of bringing all recurrent neurons into a deterministic state (due to the `tanh` transfer function, the rates are saturated at either +1 or -1). This impulse is followed by a training window of variable length (in the seconds range) and finally another relaxation period. In Figs 1 and 2, an additional impulse (duration 10 ms, smaller amplitude) can be given a certain delay after the initial impulse to test the ability of the network to recover its acquired trajectory after learning.

In all experiments, the first trial is used to acquire an innate trajectory for the recurrent neurons in the absence of noise (I_0 is set to 0). The firing rate of all recurrent neurons over the training window is simply recorded and stored in an array without applying the learning rules. This innate trajectory for each recurrent neuron is used in the following trials as the target for the RLS learning rule, this time in the presence of noise ($I_0 = 0.001$). The RLS learning rule itself is only applied to the recurrent neurons during the training window, not during the impulse or the relaxation periods. Such a learning trial is repeated 20 or 30 times depending on the experiments. Once the recurrent weights have converged and the recurrent neurons are able to reproduce the innate trajectory, the read-out weights are trained using a custom desired function as target (10 trials).

Results

Fig. 1 shows that an initially chaotic network can become deterministic after training. The target function for the single read-out neuron is a smooth impulse peaking 2 second after the initial impulse. Before training, two successive trials do not produce similar dynamics in the recurrent population, so the read-out neuron cannot reproduce the desired function. However, after training, the recurrent dynamics become identical, even in the presence of noise, allowing the read-out neuron to reproduce the target. Moreover, a short perturbation impulse to the network given 500 ms after the initial

impulse only transiently perturbs the network's dynamics, which quickly recover the learned trajectory.

Note that in the original article, the recurrent weights are trained for 20 trials, while here we use 30 trials. Only a small percentage of initial configurations (i.e. initial value of the connectivity matrix) converge after 20 trials. In our experiments, 30 trials allow for an almost systematic convergence of the weights. As shown in the original article, the longer the network is trained, the more stable the trajectories will be. During learning, the code displays the averaged mean-square error over a learning trial to show how well the weights have converged.

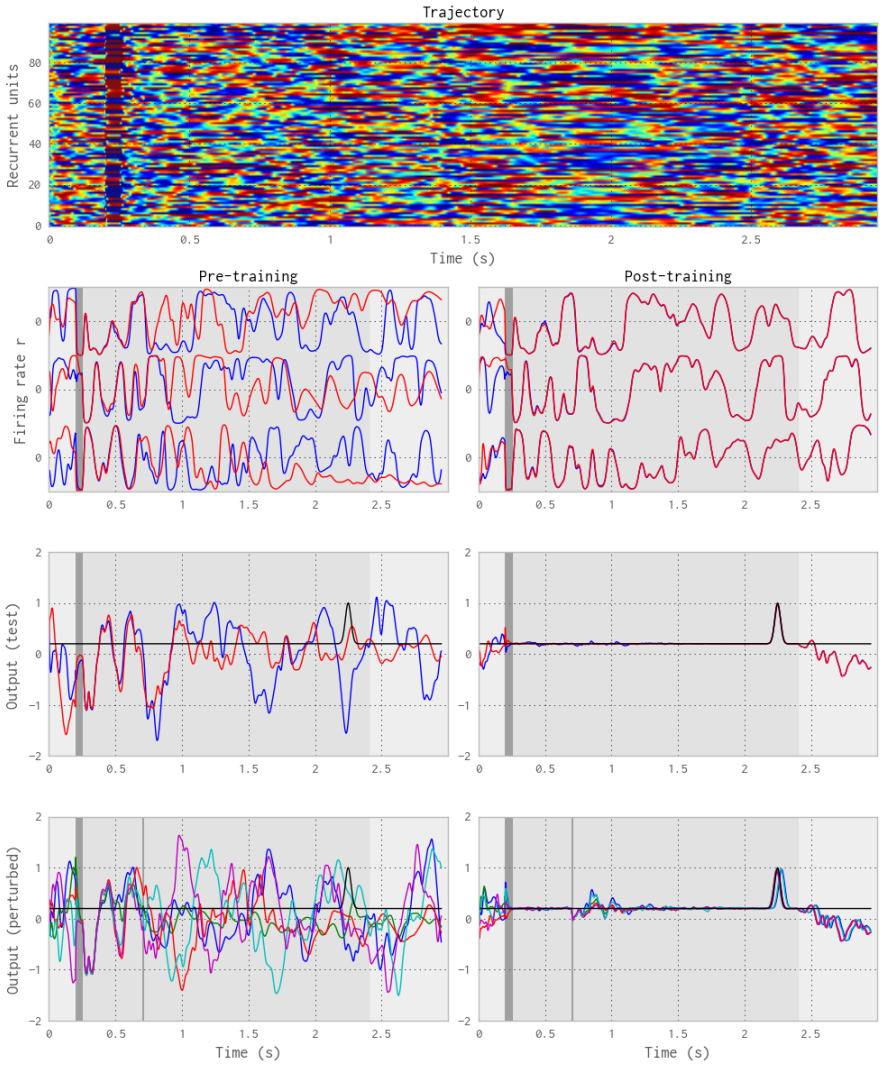

Figure 1: Complexity without chaos. Top: Temporal evolution of the firing rates of some recurrent neurons during the innate trajectory. Left: The innate trajectory (blue) superposed with a new noisy trial (red) before training. Right: The same after training. The top panel depicts the firing rate of 3 randomly chosen neurons, the middle one the firing rate of the read-out neuron, the bottom one the firing rate of the read-out neuron when a perturbation impulse is given 500 ms after the initial impulse.

Fig. 2 shows that using two read-out neurons, the network can robustly learn trajectories in the 2D space, such as written words ("chaos" and "neuron") acquired as sequences of points. The setup was similar as for Fig. 1, although 4 input neurons were used (2 per word: one for the initial impulse, one for the perturbation) and

2 read-out ones (x and y axes of the word). The training window was different for the two words (1.322 and 1.234 s respectively). As in the original paper, the scaling factor g was chosen smaller than in the previous experiment (1.5 instead of 1.8) to accelerate training: the network is less chaotic and it becomes easier to find values for the recurrent weights that lead to stable trajectories. Using $g = 1.8$ would require more training trials. Despite the complexity of the target output, the network is able to robustly learn the two trajectories. Perturbations (of small amplitude) during the sequence reproduction only briefly impair the trajectory.

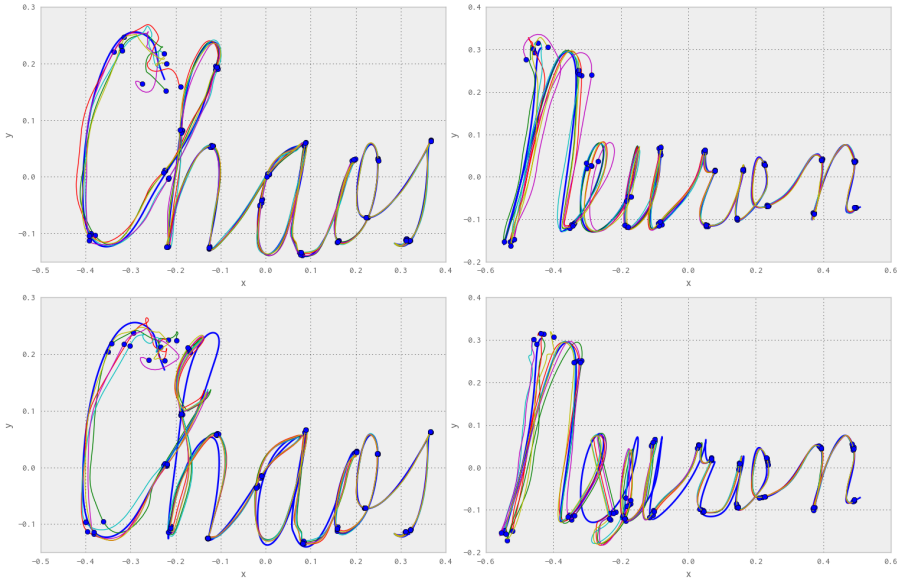

Figure 2: Generation and stability of complex spatiotemporal motor patterns. Left and right: two different trajectories in the read-out space learned by the same network. Top: without perturbation. Bottom: with perturbation. Each figure is the superposition of 5 different test trials, showing the robustness of the trajectories to initial conditions and perturbations. Equidistant time points are represented as blue circles.

Fig. 3 investigates the timing capacity of the recurrent network, i.e. the maximal duration of the target function that can be successfully learned by the network in a limited number of trials (here 20). For this experiment, the target function is chosen analog to Fig. 1: a flat function (value of 0.2) with a Gaussian peak after a certain duration. The position of the peak within the training window is systematically varied from 250 ms to 8 s, what primarily influences the total duration of a trial. The exact shape of the output function is not very important: only the length of the training window actually matters as we want the recurrent units to exhibit stable trajectories long enough. Other functions with the same total duration would lead to similar results.

For each duration, 10 randomly initialized networks were trained on the corresponding target function with the same protocol as in Fig. 1, and the correctness of the read-out activity in a test trial after learning is measured using the Pearson correlation coefficient $\mathcal{R}^2$. We observe that durations smaller than 4 seconds are perfectly learned by all networks, while this performance decreases (with a high variance) for longer durations. Longer durations actually necessitate more training trials to allow the convergence of the recurrent weights (the read-out weights are not difficult to train as long as the recurrent neurons are stable).

Note that the protocol differs slightly from the original one: in Laje and Buonomano [3], the same 10 initial networks were used to learn the different durations, while here 10 new networks are re-created every time. This may explain why in our results

learning a duration of 8 seconds leads to a better performance than for 7 seconds (luckier initializations). As running the simulations for this figure already took three days on a standard computer, we decided not to correct the learning protocol and leave the figure as is.

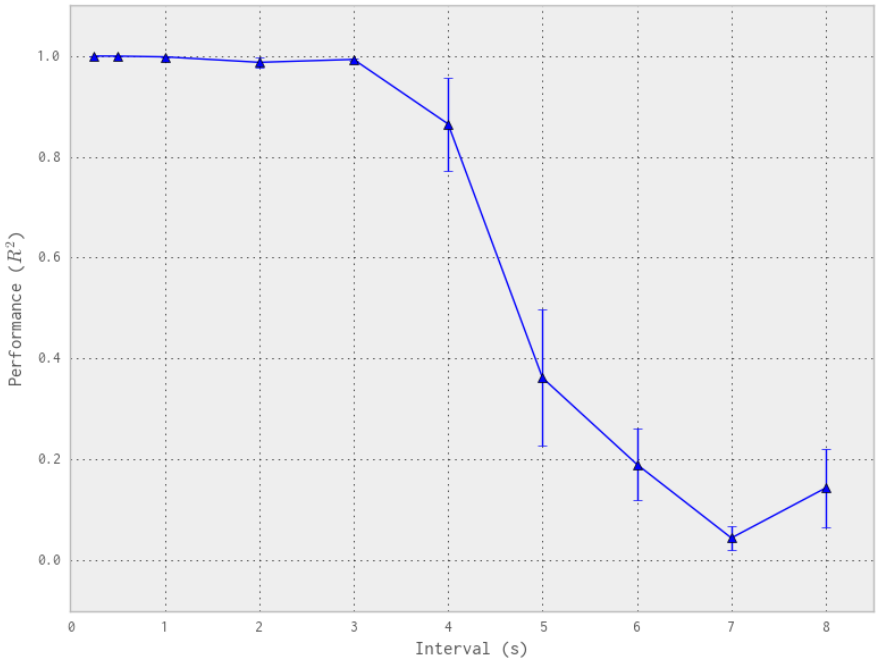

Figure 3: Improved "timing" capacity. A peak of variable duration (as in Fig. 1) is learned by different networks. The fit between the target and the reproduction is measured using the Pearson correlation coefficient between the two time series. The reproduction capacity decreases with the duration of the training window, suggesting a limited timing capacity for a single network.

The rest of the figures in Laje and Buonomano [3] focuses on a more specific analysis of the computational properties of the model (effect of noise, Lyapunov exponents, distribution of weights…). As the match between the reproduction and the original model is already high on the three first figures, we did not perform those experiments.

Conclusion

The reproduction of the model proposed by Laje and Buonomano [3] was successful. Although the original article is already quite detailed and self-explanatory, the fact that the authors released the original Matlab code on Pubmed Central allowed to resolve some small ambiguities and speed up the reproduction process:

1. The initialization of the inverse correlation matrices P to the identity matrix is not specified in the manuscript, only in the original code.
2. The RLS learning rule is described in a weight-specific way (Δw_{ij}), but Numpy (as well as Matlab) is much more efficient when using matrix/vector operations. Having the code already in this format saved a lot of time and mistakes.
3. In the implementation, the weight change is again normalized by $1 + \mathbf{r}^T \cdot \mathbf{P} \cdot \mathbf{r}$, which is not mentioned in the article. As the network does not converge without this normalization, the re-implementation would not have been possible.
4. The amplitude of the perturbations in Figs. 1 and 2 was not specified in the article nor in the code. The reproduction uses wild guesses for theses values.

This highlights the importance of open-sourcing the implementation of computational models for a correct reproducibility. As reproducibility is the most important quality of a computational model, this suggests that all national funding agencies should require their researchers to provide freely their articles (as the NIH does with PMC) together with the corresponding data (whether it is recordings or source code) in order to encourage the development of high-quality science.

References

[1] Simon S. Haykin. *Adaptive filter theory*. Prentice Hall, 2002, p. 989. ISBN: 013322760X.

[2] Herbert Jaeger. *The "echo state" approach to analysing and training recurrent neural networks*. Tech. rep. German National Research Center for Information Technology, 2001, GMD Report 148. URL: http://www.faculty.jacobs-university.de/hjaeger/pubs/EchoStatesTechRep.pdf.

[3] Rodrigo Laje and Dean V Buonomano. "Robust timing and motor patterns by taming chaos in recurrent neural networks." In: *Nat. Neurosci.* 16.7 (July 2013), pp. 925–33. ISSN: 1546-1726. DOI: 10.1038/nn.3405. URL: http://www.ncbi.nlm.nih.gov/pubmed/23708144.

[4] Wolfgang Maass, Thomas Natschläger, and Henry Markram. "Real-time computing without stable states: a new framework for neural computation based on perturbations." In: *Neural Comput.* 14.11 (Nov. 2002), pp. 2531–60. ISSN: 0899-7667. DOI: 10.1162/089976602760407955. URL: http://www.ncbi.nlm.nih.gov/pubmed/12433288.

[5] H. Sompolinsky, A. Crisanti, and H. J. Sommers. "Chaos in Random Neural Networks". In: *Phys. Rev. Lett.* 61.3 (July 1988), pp. 259–262. ISSN: 0031-9007. DOI: 10.1103/PhysRevLett.61.259. URL: http://link.aps.org/doi/10.1103/PhysRevLett.61.259.

[6] David Sussillo and L F Abbott. "Generating coherent patterns of activity from chaotic neural networks." In: *Neuron* 63.4 (Aug. 2009), pp. 544–57. ISSN: 1097-4199. DOI: 10.1016/j.neuron.2009.07.018. URL: http://www.ncbi.nlm.nih.gov/pubmed/19709635.