RESCIENCE C

Replication / Computational Neuroscience

[Re] A Neurodynamical Model for Working Memory

Theophile Boraud[1,4] **and Anthony Strock**[1,2,3]

[1]INRIA Bordeaux Sud-Ouest, Bordeaux, France – [2]LaBRI, Université de Bordeaux, Institut Polytechnique de Bordeaux, Centre National de la Recherche Scientifique, UMR 5800, Talence, France – [3]Institut des Maladies Neurodégénératives, Université de Bordeaux, Centre National de la Recherche Scientifique, UMR 5293, Bordeaux, France – [4]Department of Computer Science, University of Warwick, Coventry, United Kingdoms

Edited by
Benoît Girard

Reviewed by
Vahid Rostami
Christian Jarvers

Received
24 October 2019

Published
09 March 2021

DOI
10.5281/zenodo.4590961

1 Introduction

In this work, we successfully replicated the results of the article *A Neurodynamical Model for Working Memory*[1] written by Razvan Pascanu and Herbert Jaeger. This replication has been done using Python, and the code is available on GitHub[2]. In their article, they propose (1) a way to implement working memory and (2) a way to characterise working memory states.

(1) First, they implemented a working memory model in the form of a Recurrent Neural Network (RNN), more precisely an Echo State Network (ESN). In their model, working memory corresponds to special output units trained to maintain information through feedback connections. Their model had been trained to predict the next character in a sequence randomly generated depending on a context, and to maintain this context in working memory in order to help the prediction. We replicated the same model, trained it to solve mostly the same task and obtained comparable results.

(2) They characterised the working memory states of their model by defining a notion of attractors for input driven dynamical systems. In autonomous dynamical systems the memory states could have been characterised for instance by the stable fixed points of the dynamic. However with inputs, these points are transformed into blobs stable against input that does not aim to change the memory state. In our replication we also obtained qualitatively the same result. However in the paper they go further and propose a method to automatically find the attractors of an input driven dynamical system. They applied this method only to a toy model and not to the working memory model so we did not implement that part.

In **Section 2** we give all the implementation details of the model and the tasks it aims to solve. We also highlight the few details that were missing in the paper and that we had to make a choice on in order to reproduce the results. Then, in **Section 3**, we compare the results we obtain to their results in the former article, to see whether or not we had been able to successfully replicate the experiments.

2 Methods

2.1 Model

Their model is a slightly modified Echo State Network, a special kind of Recurrent Neural Network where only the readout weights are trained. **Figure 1** illustrates and sums up the overall architecture of the model. It is composed of a recurrent internal layer of size **N**, a.k.a. reservoir, **K** input, **L** normal output units that are not fed back to the reservoir, and **WM** additional special output units. They call the latter the Working Memory units (or WM-units). These WM-units have a recurrent trainable connection to other WM-units (including themselves), and a feedback connection to the reservoir units.
In the following we use the following notations to describe the dynamic of their model:

- Input units activations **u**, with weights matrix $\mathbf{W}^{\mathbf{in}}$ for input connections;
- Internal units activations **x**, with weights matrix **W** for internal connections;
- Output units activations **y**, with weights matrix $\mathbf{W}^{\mathbf{out}}$ for output connections;
- WM-units activations **m**, with weights matrix $\mathbf{W}^{\mathbf{mem}}$ for connections from the input units, the reservoir and the WM-units to the WM-units;
- Weights matrix $\mathbf{W}^{\mathbf{b}}$ for the feedback connections from the WM-units to the reservoir.

Given these notations, the dynamic of their model is as follow:

- Eq. (1) represents the dynamic of the internal units. This dynamic involves at the same time the state of the internal units given their previous state, the previous state of the WM-units and the actual state of input units. We use f to subsume its activation functions, which are the hyperbolic tangent in this model.

$$x(n+1) = f(W^{in}u(n+1) + Wx(n) + W^{b}m(n)) \tag{1}$$

- Eq. (2) represents the dynamic of the WM-units. This dynamic involves the state of the WM-units given the actual state of input and internal units, and the actual state of the WM-units. The sharp threshold function $\mathbf{f}^{\mathbf{m}}$ is the activation function for the WM-units, and is given by Eq. (3), while $\mathbf{W}^{\mathbf{mem}}$ is computed by linear regression during the step 1 of the model *Training* phase as shown in Eq. (5).

$$m(n+1) = f^{m}(W^{mem}(u(n+1), x(n+1))) \tag{2}$$

- Eq. (3) is the sharp threshold function used as the activation function of the WM-units.

$$f^{m} = \begin{cases} -0.5 & x \leq 0. \\ +0.5 & x > 0. \end{cases} \tag{3}$$

- Eq. (4) represents the dynamic of the output units. This dynamic involves the state of the output units given the actual state of input and internal units. $\mathbf{f}^{\mathbf{out}}$ is the activation function of the output units, and is the Identity function, while $\mathbf{W}^{\mathbf{out}}$ will be computed by linear regression during the step 2 of the model *Training* phase as shown in Eq. (6).

$$y(n+1) = f^{out}(W^{out}(u(n+1), x(n+1))) \tag{4}$$

- Eq. (5) is the equation to compute the weights matrix for WM-units using linear regression. The activations of the reservoir, the input and the target of the WM-units are all stored in matrix **H**, $\mathbf{M}_{\mathbf{target}}$ is the target of the WM-units, and † stands for the pseudo-inverse.

$$W^{mem} = (H^{\dagger} \cdot M_{target})^{T} \tag{5}$$

- Eq. (6) is the equation to compute the weights matrix for output units using linear regression. The activations of the reservoir and the input units are all stored in matrix $\mathbf{G}$, $\mathbf{Y_{target}}$ is the target of the output units, and $\dagger$ stands as before for the pseudo-inverse.

$$W^{out} = (G^{\dagger} \cdot f^{out^{-1}}(Y_{target}))^T \tag{6}$$

When the model starts, the weights are initialized as stated:

- We have $\mathbf{K} = 13$, $\mathbf{N} = 1200$, $\mathbf{L} = 65$ and $\mathbf{WM} = 6$.
- $\mathbf{W^{in}}$ is of size $\mathbf{N} \times \mathbf{K}$, with 10% of which are +0.5, 10% are -0.5, and the rest is 0.
- $\mathbf{W}$ is of size $\mathbf{N} \times \mathbf{N}$, with only 12000 random non-zero connections, which will randomly either take the value +0.1540 or -0.1540.
- $\mathbf{W^b}$ is of size $\mathbf{N} \times \mathbf{WM}$, where every weight is either +0.4 or -0.4.

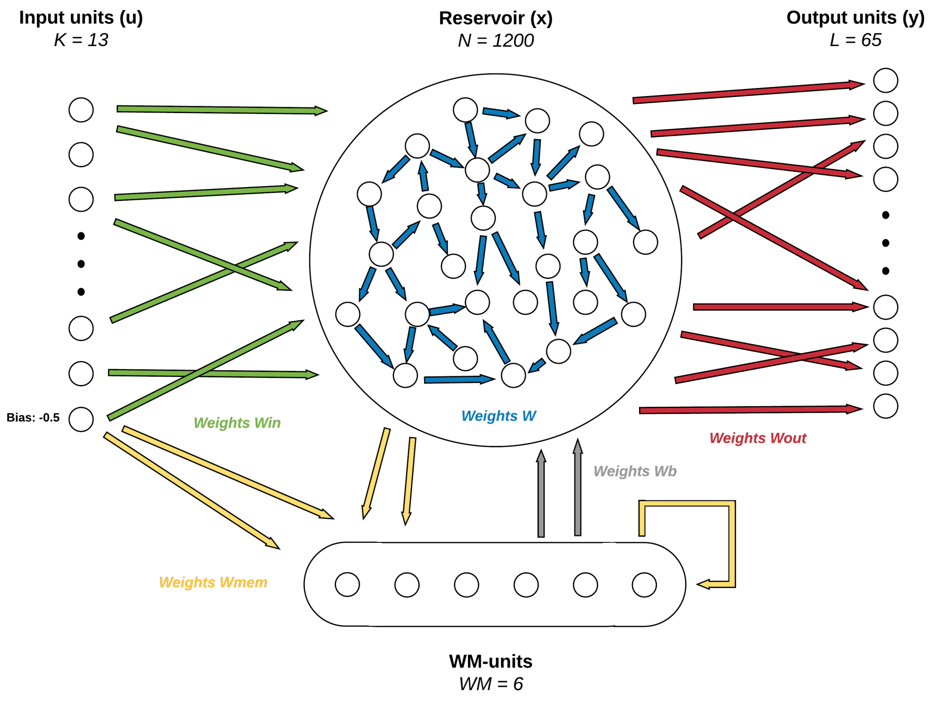

Figure 1. Model architecture. See **Section 2.1** for further details.

2.2 Input generation

Pascanu and Jaeger also describe the method they used to generate the input of the model. The input consists of a constant bias (-0.5) and a column of 12 pixels of an image. Each column of 12 pixels of the image is given as input to the network one after the other.

This image represents a sequence of characters, i.e. curly brackets (opened or closed) or 65 other ASCII symbols (e.g. letters in lower cases, numbers...). To each of these characters we associate a number that is its position in an alphabet sequence (see **Table 1**). For the sake of simplicity, in the following we say character i for the i-th character in that sequence. The sequence of characters is generated randomly using different 7 Markov Chains, and the Markov Chain used depends on the current curly bracket level. More precisely, the next character is either:

- an open (resp. closed) curly bracket thus increasing (resp. decreasing) the current bracket level by 1 to a maximum (resp. minimum) of 6 (resp. 0)
- or generated with a Markov chain given the last character that was not a curly bracket

Given that the next character is not a curly bracket, if the current bracket level is j, the next character after i will either be:

- $i + j + 1$ modulo 65 with probability 0.8;
- any of the 64 other characters from the alphabet with probability $\frac{0.2}{64} = 0.003125$.

During the training phase, the probability for the next character to be an open (resp. closed) curly bracket is 0.15. Whereas in the testing phase it is 0.03, which forces the WM-unit to maintain the current bracket level for a longer time.

The sequence is transformed into an image by displaying and concatenating each of its characters. The characters are displayed using a randomly selected font and then uniformly randomly stretched between 6, 7 or 8 pixels in width. Finally a salt-and-pepper noise of amplitude 0.1 and probability 1 is added to the whole image.

See **Table 2** to observe an example of input sequence generated from the alphabet in **Table 1** in **Section 2.6.2**.

2.3 Task

The aim of the WM-units is to keep track of the current bracket level. When the current bracket level is k, the k first WM-units should be at +0.5, whereas the remaining ones should be at -0.5.

The aim of the output units is to predict the next character in the sequence when the current character is not a curly bracket. The prediction for the next character should be made in the middle of receiving the current character. If character i should be predicted, then all the output units but the i-th one should be set to 0 whereas the i-th one should be 1.

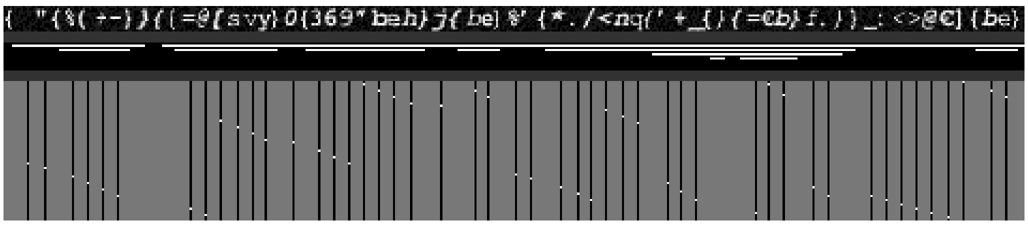

Figure 2. Input and target outputs of the network. The first row represents the image input sent to the network columns by columns; the second is the target values for WM-units, with each white rows representing the time steps on which the corresponding WM-units has a value of 0.5, and in black when its value is -0.5 (a.k.a. white when the corresponding bracket level has been reached, and black when not); the third one is the target output, with 65 black pixels (value 0) at the middle of the character representation, and the next predicted character represented by a white pixel (value 1) depending of its location in the alphabet (see **Table 1**).

2.4 Training and testing

As they did, we train W^{mem} and W^{out} separately with different input sequences. First we train W^{mem} using teacher forcing and then we train W^{out}. We use a sequence of 10000 characters to train W^{mem}, and of 49000 characters to train W^{out}. The teacher signal for training W^{mem} is defined at all times whereas for training W^{out} it is defined only in the middle of characters that are not curly brackets. Finally we test the model with a 35000 characters long sequence. As an error on the bracket level would propagate and create more errors, each time the model makes a mistake on the bracket level in the WM-units we count it and correct it. These errors are classified between *false positives* (i.e. the network detects a bracket when there is none) and *false negatives* (i.e. the network fails to detect a bracket). The errors made by the output units are also counted but not corrected. The error rate of prediction is then computed given the number of characters the network failed to predict correctly and the total length of the sequence.

2.5 Attractors

Following the instructions in the original article, we run the WM model 7 times, each time with a different sequence with no bracket, and forcing the WM-unit to a fixed bracket level ($k = 0$ for the first, $k = 1$ for the second, etc...). Each sequence has a length of 6500 characters, for about 45,000 network updates in total. We collect in each case both the reservoir states and the input units. Then, every sets of reservoir states (resp. input vectors) are concatenated, and their first principal components (or PCs) are computed. Finally, for each of the 7 original sets, the first PC of the inputs is plotted against the first two PCs of the reservoir states.

2.6 Implementation choices

In the original article very few parts were unclear. Pascanu and Jaeger briefly explain how they choose to generate the input, but did not specify the exact method and tools they used to implement it. This subsection aims at listing out the choices we made and the tools we used to obtain similar results than the original paper.

Language and libraries — Since the language used for the former model was not specified, we decide to use Python (3.7.4) to implement our model. The article did not specify which tools they used to transform their symbolic sequence into an image. Thus, we choose to use the libraries *PIL*, *FreeType* and *SciPy* for this purpose.

Input sequence alphabet — In the original paper Pascanu and Jageger did not specify which exact characters they used for their 65 symbols, and therefore not the exact order of this alphabet. For this, we decide to create our own, as shown on **Table 1**. We can observe in **Table 2** an input sequence example in order to further understand the behaviour of the input generation algorithm with our alphabet.

Alphabet sequence order
abcdefghijklmnopqrstuvwxyz0123456789 !”#$%&’()*+,.-_/:;<=>?@€|[]§

Table 1. Order of alphabet for data generation.

Font — We have not been able to find the fonts used in the original model (*FreeMono*, *FreeMono Bold*, *FreeMono Oblique* and FreeMono Bold Oblique of Gimp 2.3.6). Therefore, we try in the first place to use these *FreeMono* (classic, oblique, bold and bold oblique) fonts. However, as we encounter more errors than in the original paper, we use the *Incosolata* (regular and bold) fonts distributed by Google, which give far better results.

Input sequence example
abc{eg024{7{!osw0-}:={{fkp

Table 2. Example of input sequence. Each character in red corresponds to a randomly chosen character which does not depend on the current character and the memory states (with a probability of 20%). Use **Table 1** to understand how the input is selected 80% of the time depending on the bracket level (i.e. when the characters are in black).

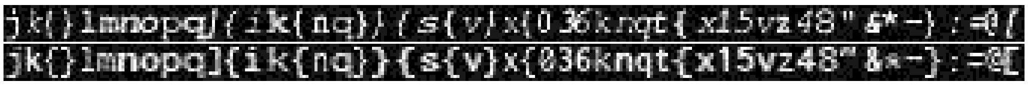
jk{}lmnopq]{ik{nq}}{s{v}x{036knqt{x15vz48" &*-}:=@[
jk{}lmnopq]{ik{nq}}{s{v}x{036knqt{x15vz48" &*-}:=@[

Figure 3. Images example of a sequence using 2 different set of fonts. The first image is computed using *FreeMono*, the second is using *Inconsolata*.

Noise — Pascanu and Jaeger did not specify in the original the probability of the salt-and-pepper noise. Thus, we tried a salt-and-pepper noise of amplitude 0.1, and probability 1.

Warm up in Principal Component Analysis — We can observe that for each memory state the first few points do not necessarily gather with their cluster. Thus, the first 100 time steps are removed from display only for clarity purposes.

3 Results

3.1 Network

According to the former article, the experiment has been run over 30 runs for the network. We assume that each time, it uses different randomly generated training and testing sequences, and reservoir, input and feedback weights. Those 30 runs are then used to compute mean and standard deviation of different performance measures. In the following Tables we use the color light green for the *FreeMono*[1] fonts (first row on result tables), dark green for *Inconsolata*[2] (second row), and black for their former results (third row).

First, as they did in the original paper we inspect the WM-units performance. We dissociate the errors made by the WM-units into false negative errors (when the network did not recognize a curly bracket as such) and false positive errors (when the network detects the character as a curly bracket while it is not). These errors are counted, and then transformed into three different percentages, depending on the number of curly brackets in the sequence, the number of characters, or the number of time steps (i.e. length of the input image in pixels). The **Table 3** shows the results obtained.

Then we look at the characters falsely identified as curly brackets. The **Table 4** shows how the characters highlighted in the original experiment are falsely identified as curly brackets.
As in the former article, we measure the average absolute value of W^{mem} weights over the 30 runs, which can be observed in **Table 5**.

[1]Seed used for results using *FreeMono*: *1639617780*
[2]Seed used for results using *Inconsolata*: *3939310522*

Type of error	Number of errors	Percentage of curly brackets (%)	Percentage of characters (%)	Percentage of time steps (%)
false negatives	54.7 ± 6.6	3.01 ± 0.35	0.16 ± 0.020	0.022 ± 0.003
	0.2 ± 0.4	0.01 ± 0.02	0 ± 0.001	0 ± 0
	7.2 ± 6.5	0.34 ± 0.30	0.02 ± 0.018	0.003 ± 0.002
false positives	1252.6 ± 48.5	68.97 ± 2.44	3.77 ± 0.148	0.511 ± 0.020
	97.2 ± 34.5	5.37 ± 1.89	0.29 ± 0.104	0.040 ± 0.014
	59.8 ± 21.6	2.84 ± 1.02	0.17 ± 0.061	0.024 ± 0.008
total	1307.3 ± 49.3	71.98 ± 2.44	3.94 ± 0.151	0.534 ± 0.020
	97.4 ± 34.6	5.37 ± 1.89	0.29 ± 0.104	0.040 ± 0.014
	67.0 ± 22.9	3.18 ± 1.09	0.19 ± 0.065	0.027 ± 0.009

Table 3. Number of erroneous WM states obtained by the ESN, averaged over 30 runs. Colour light green is used for results using *FreeMono* as font, dark green for *Inconsolata*, and black for the original results.

Character	Number of times the character is in testing sequences	Number of times the counter increased	Number of times the counter decreased
"("	508.7 ± 23.8	128.07 ± 14.08	3.8 ± 2.0
	514.2 ± 23.7	93.40 ± 34.50	0 ± 0
	499.5 ± 22.3	21.5 ± 10.1	0 ± 0
")"	516.7 ± 19.7	12.10 ± 4.21	3.9 ± 1.7
	516.8 ± 18.2	0 ± 0	0.3 ± 0.7
	502.4 ± 18.6	0 ± 0	0.5 ± 0.2
"["	513.1 ± 18.8	191.83 ± 28.71	3.8 ± 1.6
	507.9 ± 15.6	0 ± 0	0 ± 0
	496.2 ± 22.8	5.8 ± 5.1	6.0 ± 5.4
"]"	509.6 ± 18.3	9.53 ± 3.58	3.0 ± 1.4
	514.4 ± 18.3	0.13 ± 0.34	0 ± 0
	501.3 ± 15.1	0.05 ± 0.03	6.0 ± 5.4
"@"	507.0 ± 20.8	10.13 ± 2.84	3.9 ± 2.1
	508.8 ± 22.5	0 ± 0	0 ± 0
	492.7 ± 21.3	25.1 ± 14.1	0.2 ± 0.1
other	-	656.23 ± 30.19	226.2 ± 22.5
		3.13 ± 2.03	0.2 ± 0.6
		0.05 ± 0.04	0.6 ± 0.5

Table 4. Trigger characters for false positives, averaged over 30 runs. Colour light green is used for results using *FreeMono* as font, dark green for *Inconsolata*, and black for the original results.

As we can see in **Tables 3 and 4**, by using the *FreeMono* font we obtain way more errors in the WM-unit than in the original paper. However by using the *Inconsolata* font we obtain similar results.

Finally, we compute the average error rate for the prediction made in the output units. We count only the errors in the middle of the presentation of a character that is not a curly bracket, as the target is defined only during these time steps. Over the 30 runs, with the *Freemono* font we find an error rate of **26.02 $\pm$ 0.32%**, for **24.83 $\pm$ 0.27%** in the former article. With the *Inconsolata* font we obtain an error rate of **22.22 $\pm$ 0.25%**. These three results are very similar.

It is important to note that, during our various testing, and as in the original model,

considered weights	average absolute value
input to WM-units weights	0.3395 ± 0.2975
	0.2834 ± 0.2445
	0.2327 ± 0.1813
reservoir to WM-units weights	0.0971 ± 0.0920
	0.0852 ± 0.0782
	0.0667 ± 0.0591
WM-units to WM-units weights	0.9217 ± 0.6893
	0.7003 ± 0.5436
	0.5825 ± 0.5627

Table 5. Average learned output weights of the ESN over 30 runs. Colour light green is used for results using *FreeMono* as font, dark green for *Inconsolata*, and black for the original results.

the network never changes the WM-units to an invalid state or by increasing or decreasing the counter by more than one, attesting their assumption that any error would be the result of misclassification of a character (as expected), and not any other subprocesses of the WM-units.

3.2 Attractors

As in the original paper, for both *Freemono* and *Inconsolata* font we can see clusters of points in the first two principal components(see dark projection in **Figure 4**). Each of this cluster seems to be an attractor (as they defined in the original paper) associated to a memory state. It is stable against the characters that are not curly brackets, and the curly brackets makes it move from one to the other.

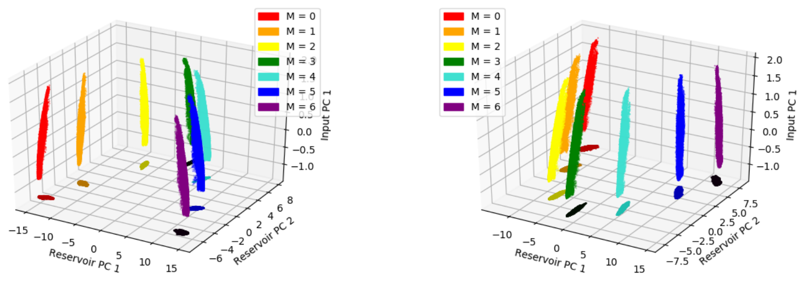

Figure 4. Principal Component Analysis results. Each coloured column represents the several reservoir states for each memory states in the form of attractors, with *FreeMono* results on the left, and *Inconsolata* on the right. For each memory states we do not display the first 100 points. We can observe that the points for the different working memory states are clustered much more clearly than in the original article, spanning a much larger range. However, being only a quantitative difference, the results stay very similar from the original

4 Conclusion

Considering the original tools were not sourced, we needed to use our own tools, whether it be for the language, the fonts but also for the input generation, which could easily justify the differences between the original results and the ones found in this replication. However, these differences also highlight the fact that this model is really modular and consistent, giving similar results despite the differences in fonts and error rates.

In the end, we consider that we have been able to successfully replicate the result obtained by Pascanu and Jaeger in *A Neurodynamical Model for Working Memory*. Very few information were missing in the original paper, such as the input generation methods from string to image they used, a more precise definition of the noise they added or the ASCII characters alphabet they used.

References

1. R. Pascanu and H. Jaeger. "A Neurodynamical Model for Working Memory." In: **Neural Networks** 24 (2 Mar. 2011), pp. 199–207.
2. T. Boraud and A. Strock. **theoboraud/ESN: Release 1.1 (ReScience submission)**. Version 1.1. Oct. 2019.