Text Generation Using Reservoir Computing

Advanced Computing 2017 project report

Francesco Cremonesi

I. Introduction

Reservoir Computing (RC) is a different approach to Recurrent Neural Networks (RNN). The benefit of using RC is that it simplifies considerably the training phase. The main idea behind RC is that supervised adaptation of all interconnection weights is not necessary, and simply training a memoryless supervised readout is sufficient.

There are two main approaches to Reservoir Computing, namely Echo State Machines (ESM) [4] and Liquid State Machines (LSM) [11]. Although now they are considered unified under the umbrella term Reservoir Computing, the main difference historically has been that ESM used *artificial neurons* in their reservoir (usually LSTM or GRU cells) whereas LSM used biologically-inspired models such as the leaky integrate-and-fire model.

In this project, I am going to attempt to apply the RC technique to the Text Generation Problem (TGP). In the TGP the network is trained with a sufficiently long and rich text. Although there are different approaches for *embedding* the input words into a lower-dimensional space, I will take here a very simple approach known as *character-based*, where every character in the text becomes directly an element of the input sequence. After training, the network is expected to be able to generate a new sequence, given a starting seed (one or more characters).

II. Reservoir Computing Formalism

In Reservoir Computing, the training of the recurrent part of the network is decoupled from the training of the readout neurons. In particular, usually the recurrent part is not trained at all, or sometimes can be trained using unsupervised methods. Figure 1 illustrates the main differences between a RNN and a RC approach. In this project, I will use a reservoir composed

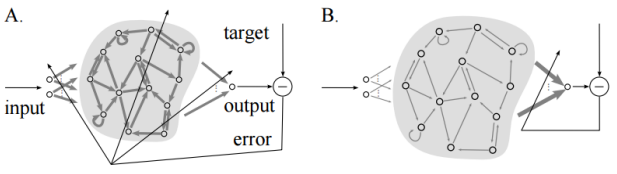

Fig. 1. **A.** A typical RNN approach, all the recurrent weights and input weights are co-trained with the readout weights. **B.** the RC approach, only the output-to-readout weights are trained. This Figure is taken from [9].

of *leaky integrate and fire* neurons (see Section III-A), a biologically inspired model of a spiking neuron.

A. Training the Readout Neurons

The recommened way of training a RC network is via Ridge Regression [8]. The algorithm is the following:

1) run all the simulations of the reservoir corresponding to the different input sequences;
2) aggregate all the outputs of the readouts into a matrix $\tilde{X}$;
3) stack the design matrix with an encoding of the input and a constant term, to create the design matrix $X = [\tilde{X}; U; 1]$
4) let Y be the aggregation of the desired outputs, use ridge regression to obtain the weight matrix

$$W = YX^T \left(XX^T + \beta I\right)^{-1}; \tag{1}$$

Therefore, if N_c is the number of possible characters in the sequence, N_s the total number of characters in all sequences, N_r the number of readout neurons we have:

$$W \in \mathbb{R}^{N_c \times (N_r+N_c+1)} \qquad Y \in \mathbb{R}^{N_c \times N_s}$$
$$X \in \mathbb{R}^{(N_r+N_c+1) \times N_s} \qquad I \in \mathbb{R}^{(N_r+N_c+1) \times (N_r+N_c+1)}$$

This training method implies that all the simulations are done at the beginning, and there is no *online* learning.

There are a few subtleties to training that I tried to understand from the literature, by combining information from [8], [9], [10], [5].

a) Providing input to the reservoir: is not a standardized procedure. In theory, it shouldn't matter how you do it as long as you guarantee a sufficient separation (i.e. not collapsing everything to 0). I choose to do it in a straightforward way: each character of the input sequence corresponds to a (randomly selected) neuron in the reservoir. When a character is observed in the sequence, I begin a two-phase stimulation protocol:

1) I stimulate the corresponding neuron for a fixed, small, period of time τ_{stim} with a very high frequency (100Hz) spike input from an external source;
2) I stop stimulating the neuron and wait a fixed period of time τ_{quiet}.

b) Filling the columns of X: requires making a conscious choice of what constitutes an input signal to the readout weights. Following [8] I use the time-averaged output from the readout neurons over an arbitrarily defined time window τ_{window}. In particular, since I keep the duration of the time window fixed, I simply count the number of spikes produced in that period and use that as my input signal.

c) The feedback loop: is necessary in a text generation task, in order to feed back the predicted character as input for the next phase. This feedback loop can introduce unwanted dependencies, in particular during training. For this reason, I use a technique called *teacher forcing*, where in the training phase I only use the correct predictions (i.e. the next element in the training sequence) as input to the network. On the other hand, during the inference phase I use the predicted character as input to the network.

III. The Experiment

A. The Leaky Integrate and Fire Neuron

In this work, I have decided to use one of the simplest *biologically plausible* neuron models in the literature: the *leaky integrate-and-fire* neuron [2]. This model is characterized by two states: a sub-threshold state in which the neuron acts as a leaky integrator, and an excited state where the neuron emits a spike and then enters a refractory period, during which its membrane potential is clamped to the resting state value. Therefore, the neuron's dynamics is governed by a differential equation in the sub-threshold phase, but everytime the threshold potential is crossed a spike is emitted and the refractory period is entered. Spikes are believed to be the primary way by which neurons communicate information between each other. Spikes are a binary, all-or-none event that occurs when certain threshold conditions are met, and correspond to a rapid increase in membrane potential (called action potential) that causes after a delay δ_{syn} the release of certain signalling molecules (neurotransmitters) which in turn cause a small flow of current in all the topological neighbors of the neuron. Denoting by V the membrane potential of the neuron, the complete mathematical description is given by:

$$\begin{cases} C_m \dfrac{dV}{dt} = -\dfrac{V - V_{rest}}{R_m} + I_{syn}(t) & \text{if } V(t) < V_{th} \\ V(t) = V_{rest} \text{ while } t - t_{sp} < t_{refrac} & \text{if } V(t_{sp}) = V_{th} \\ \dfrac{dI_{syn}}{dt} = -\dfrac{I_{syn}}{\tau_{syn}} & \\ I_{syn} \leftarrow I_{syn} + w_{ij} PSC & \text{if incoming spike} \end{cases} \tag{2}$$

where V_{th} is the spike threshold potential, V_{rest} is the resting potential (equilibrium), C_m, R_m are the membrane's capacitance and resistance, I_{syn} is the cumulative synaptic input currents, t_{sp} is the time of occurrence of the last spike, t_{refrac} is the duration of the refractory period, τ_{syn} is the synaptic time constant, w_{ij} is the connection weight between neurons i and j and PSC is a fixed, Post-Synaptic Current.

B. Reservoir Network Topology

I start with a small reservoir of 400 neurons. Moreover, I make a distinction between two kinds of neurons: excitatory and inhibitory. Excitatory neurons are characterized by the fact that they only make connections with positive weights in eq. 2, whereas inhibitory neurons are the opposite. The first network topology I consider is the balanced random network. In this case, neurons are randomly divided in excitatory (80%) and inhibitory (20%), and all the neurons make exactly 40 random connections. Random connections means any two neurons have equal probability of forming a connection.

I have fiddled around quite a lot with the neuron parameters in order to obtain reasonable behaviour. The goals I had in mind are the following:

- the network should converge to a quiet state if I stop giving it inputs;
- starting from a quiet network, the input corresponding to any character should be sufficient to elicit a response *not only in the stimulated neuron*;
- τ_{stim} and τ_{quiet} should be chosen such that, during a sequence, the next input is given while some activity (but not all of it) still lingers from the previous input.

This is a good moment to point out that these goals came from my mind, nor did I find any literature giving specific advice about this.

C. Simulation

For simulating the reservoir, I used the popular software package NEST [3] release 2.12.0. The advantage of NEST is that it has several built-in models, such as the leaky integrate-and-fire, and helper functions to easily setup a network of point neurons. The main disadvantage of NEST is that it is extremely memory-hungry: simulating a network of 10K neurons requires slightly more than 8GB of memory.

To simulate the model, a set of parameters that seemed good to me is the following:

V_{rest}	0.0	V_{th}	20.0
C_m	1.0	R_m	20.0
t_{refrac}	0.4	τ_{syn}	0.6
PSC	7.0	w_{ij}	exc: 1.0, inh: -5.0
τ_{stim}	1.0	τ_{quiet}	3.0
Δt	0.1	δ_{syn}	0.3

Figure 2 was generated using the stimulation protocol explained in the previous subsection for the first 250 ms of simulation (corresponding to the first 25 input characters), then was left quiet for another 70 ms. It is possible to identify the single neurons firing in correspondence to the different inputs, as well as the overall activity of the network. It shows how the chosen set of parameters satisfies strongly the first and third requirement of the previous subsection, and only weakly the second (this is the best I could find). Indeed, there seem to be some inputs that cause a global activation of the network, and others that can't do that.

D. Input Dataset

As input dataset I have used Italy's pride: *La Divina Commedia* by Dante Alighieri translated by LongFellow (from project Gutenberg's website [1]). The Divina Commedia is 113337 words (661393 characters) long, but for training I ended up only using 80% of the dataset (529113 characters). Figure 3 shows the histogram of the individual characters. Funnily enough, although this is an english text, the fact that it is a translation from italian seems to be reflected in the abundance of 'o' and 'a' characters, with the space character being only the third most common. To give an example, this

is the incipit of the text:

"Midway upon the journey of our life
I found myself within a forest dark,
For the straightforward pathway had been lost.

Ah me! how hard a thing it is to say
What was this forest savage, rough, and stern,
Which in the very thought renews the fear. "

A second interesting way to visualize the data is to see how many times a given character follows another. This information is presented in Figure 4, which shows that the space character (after the 't' in the labels) is followed by all characters equally likely, but almost always follows a full stop.

E. A Baseline

In this work, I have taken inspiration from A. Karpathy's blog [7]. In order to establish a baseline, I have downloaded a torch implementation of a very simple single-layer RNN that uses LSTMs from the github repository linked in his blog [6]. Among other things, Karpathy applied his RNN to Lev Tolstoy's *War and Peace*, which is what inspired me to use Dante's work for my own project.

Installing Lua and downloading the RNN implementation was fairly straightforward, although I did not even try to intall the GPU version. The github repository [6] comes with a tokenizer python script, which was very useful to me since I could reuse the same tokenised dataset for my own reservoir computing implementation later on. Training the CPU version for 50 epochs took roughly 4 hours on my work-desktop. After full training, this is what the I could sample from the network (the first sentence is my seed):

"Charon the demon, with the eyes of glede,
Thy strustened unto the pals that seemed us.

A shalt creeced and Refinalious
Think the wound know a said, bodionlass that,
Now there thinking so stall be art;
'Be theakind with arage that repear that thinks, by sted?" "

A. Karpathy's network has learned many things:

- words are separated by spaces;
- the words themselves seem english-like, although not all of them are actual english words;
- the structure of the text is of short sentences, and every once in a while a line break.

Although a human can easily tell the difference between the real thing and this, I still feel that this RNN did a fantastic job.

IV. Results

Although I wasn't able to bring the reservoir even close to the performance of the RNN, there are still a few interesting lessons to be learned.

A. One-Step Learning

In theory, the strength of the reservoir should be to maintain an *internal state* which acts as a kind of memory of past states. So, an important thing to ask oneself is *what would the method learn if there wasn't any reservoir?* Given that we just said the reservoir acts as a memory, I believe this would be equivalent to one-step learning, i.e. to learning, given a character, what would be the most-likely next one. Simulating the lack of a reservoir is pretty easy in our context: it can be achieved by setting the first rows of X to 0 (since I am using regularization, this process of setting whole rows to 0 doesn't cause numerical problems). Here's the result after feeding an input dataset of 10K characters, and turning off the reservoir:

"Charon the demon, with the eyes of glede,

the the the the the "

Actually, in almost all cases, without the reservoir the output consists of two return carriages, followed by an endless string of the sequence *the* . A very interesting exception to this is the apostrophe. Seeding the sampling simulation with the sequence *planet'* leads to the following result:

"planet's the the the the "

As you can see, the one-step learning has learnt that after an apostrophe there should always be an s.

B. Without Concatenating the Input to the Output

One muight also ask the reverse question: *what happens if the reservoir is left alone in predicting the next characters in the sequence?* The resulting W matrix is shown in Figure 5, where it is almost possible to see that the part corresponding to the input is now full of 0 values.

The output is quite strange:

"Charon the demon, with the eyes of glede, hhhh
ooeeeenoo "

An interesting phenomenon is happening here: the network is often repeating characters. This makes me afraid that Figure 2 actually presented a worse situation than I originally thought: it could be that characters that cause a full network blow-up end up taking over the inference process. So, I tried with shorter time constants to see what would happen. Setting τ_{stim} 0.5 τ_{quiet} 0.5 resulted in the following generation (still turning off the input): The output is quite strange:

"Charon the demon, with the eyes of glede,
rsssrsssshhhhseesiiii

i

ttttteeeee "

I'm not sure whether this is better or worse, but in any case it allowed a significant speedup.

C. Effect of Regularization Coefficient

I have found the regularization coefficient, i.e. β in eq. 1, to be quite important. In particular, it appears to play a double role: not only does it serve as regularizer thus ensuring that the matrix $XX^T + \beta I)$ is indeed invertible; it also acts as a knob to balance the two effects given above. I have found that a large β gives a behaviour more similar to the one-step learning described above, whereas a small β gives rise to a behaviour similar to the reservoir-only one. This is confirmed by Figure 6 where the W matrix is displayed.

Moreover, I have found that the range that defines small and large values of β varies with the length of the training sequence. Longer training sequences correspond to a shift of the range towards larger values. To be more clear, for a training sequence of length 1K characters, a large value of β would be around 10, and a small around 0.01. For a training sequence of length 10K characters, a large value of β would be around 100 and a small around 1. Thinking that this was somehow tied to the fact that the input is concatenated to the readout neurons, and then we compute the product of this matrix with its transpose, I thought that normalizing the values of the input from 1 to $\frac{1}{\text{len(input)}}$ or even $\frac{1}{\text{len(input)}^2}$ would help, but the few experiments I conducted didn't seem promising.

D. Standard Reservoir Training Results

Finally, I'd like to present some results from one of the largest training sessions that I was able to perform. I submitted to the network 50K characters, i.e. one tenth of the training data. Please note that this is very far from the 50 epochs that A. Karpathy's RNN received, but still corresponds to a couple of hours training on my work-desktop. The nice thing about reservoir learning is that, after the training is complete, one can still experiment with some parameters (typically β) without needing to re-run the whole simulation.

Here's what the reservoir generated after seeding it with the sequence *planet'*:

$\beta < 100$ " *planet'mttmmaa n hiii"*
$100 \leq \beta \leq 500$ " *planet's ttit thiiiimhhhh "*
$\beta > 500$ " *planet's the the the the"*

Here are more results, where I tried to increase the duration of $\tau_{stim}, \tau quiet$ from (0.5,0.5) to (5.0, 15.0) (using only 3K characters as input):

$\beta = 0.5$ " *planet'e the s me e s the thethe stherere"*
$\beta = 2$ " *planet'stherertha thathe the the the the the the "*
$\beta = 20$ " *planet'e the the the the the the "*

Just for fun, here's how the network would have helped Dante when he got stuck (in bold is the seed):

- " ***But after I had reached a mountain'****s sereshe he the the the the as here ere our r e e the the the the th an and athere fesesese e w"*
- *" **Charon the demon, with the eyes of glede,***
  thoth t t lourere hehe s the he an and the the the he s s the an an as the the athe me wathe the the tath"
- *" **"O Master, what so grievous is To these, that maketh them lament so sore?" He answered: "***
  the the the the theand and the the the me me the sed s he he the the th s s e e e e an t an thereand ad ath"
- *" **"Marvel thou not," she said to me, "because** oure the the the me s as e me me the the"*

V. Challenges and Future Perspectives

I encountered many challenges in this project. First, I had to decide a satisfactory set of parameters for the neurons and connections in the reservoir. Although this required some time, I did not spend too much effort on this since the theory behind reservoir computing and LSMs in particular is that it shouldn't matter too much, as long as the dynamics of the reservoir are sufficientlt rich and chaotic.

The main challenges I encountered, I would say, were defining a good mapping between input sequences and reservoir, and finding a good regularization coefficient. I believe the first issue is very important, because the particular dynamics of a balanced network are such that activity dies out after a certain period of time, so the stimulation arising from the new input must come quickly enough to prevent that. On the other hand, if the new input comes too quickly, the reservoir's state will be too mixed up and the network won't be able to distinguish between states. I believe this second situation could be what causes my network to repeat each character several times in the sampling phase, but I can't be completely certain. To tackle these issues in the future, one may consider exploring in a more systematic way the space of possible $\tau_{stim}, \tau_{quiet}$. However, this idea come swith the additional challenge of formalising an expected behaviour, something that is not trivial. Another possibility would be instead to tweak the balanced network's parameters such that, instead of dying out, it enters a self-sustained oscillatory state. Guaranteeing this for all possible inputs, however, seems pretty difficult.

References

[1] Dante Alighieri. La divina commedia, 2017. http://www.gutenberg.org/ebooks/1004.

[2] Wulfram Gerstner, Werner M Kistler, Richard Naud, and Liam Paninski. *Neuronal dynamics: From single neurons to networks and models of cognition*. Cambridge University Press, 2014.

[3] Marc-Oliver Gewaltig and Markus Diesmann. Nest (neural simulation tool). *Scholarpedia*, 2(4):1430, 2007.

[4] Herbert Jaeger. Echo state network. *Scholarpedia*, 2(9):2330, 2007.

[5] Herbert Jaeger. A quick introduction to reservoir computing. 2010. lecture slides.

[6] J. C. Johnson. https://github.com/jcjohnson/torch-rnn.git, 2017.

[7] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. *Andrej Karpathy blog*, 2015.

[8] Mantas Lukoševičius. A practical guide to applying echo state networks. In *Neural networks: Tricks of the trade*, pages 659–686. Springer, 2012.

[9] Mantas Lukoševičius and Herbert Jaeger. Reservoir computing approaches to recurrent neural network training. *Computer Science Review*, 3(3):127–149, 2009.

[10] Mantas Lukoševičius, Herbert Jaeger, and Benjamin Schrauwen. Reservoir computing trends. *KI-Künstliche Intelligenz*, 26(4):365–371, 2012.

[11] Wolfgang Maass, Thomas Natschläger, and Henry Markram. Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, 14(11):2531–2560, 2002.

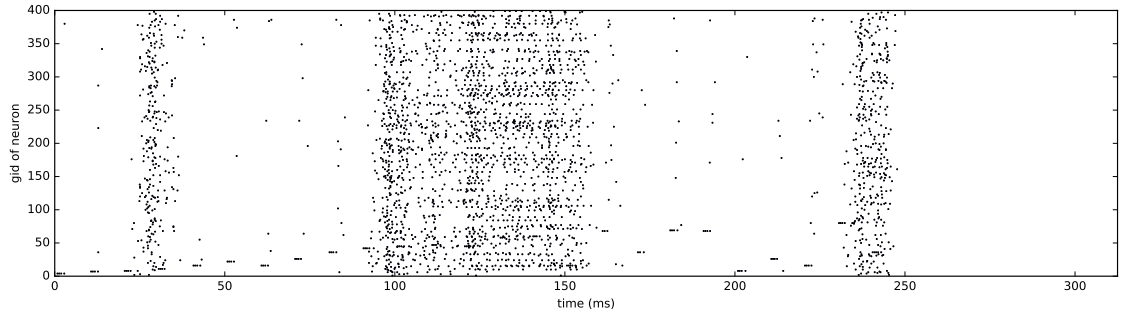

Fig. 2. Raster plot.

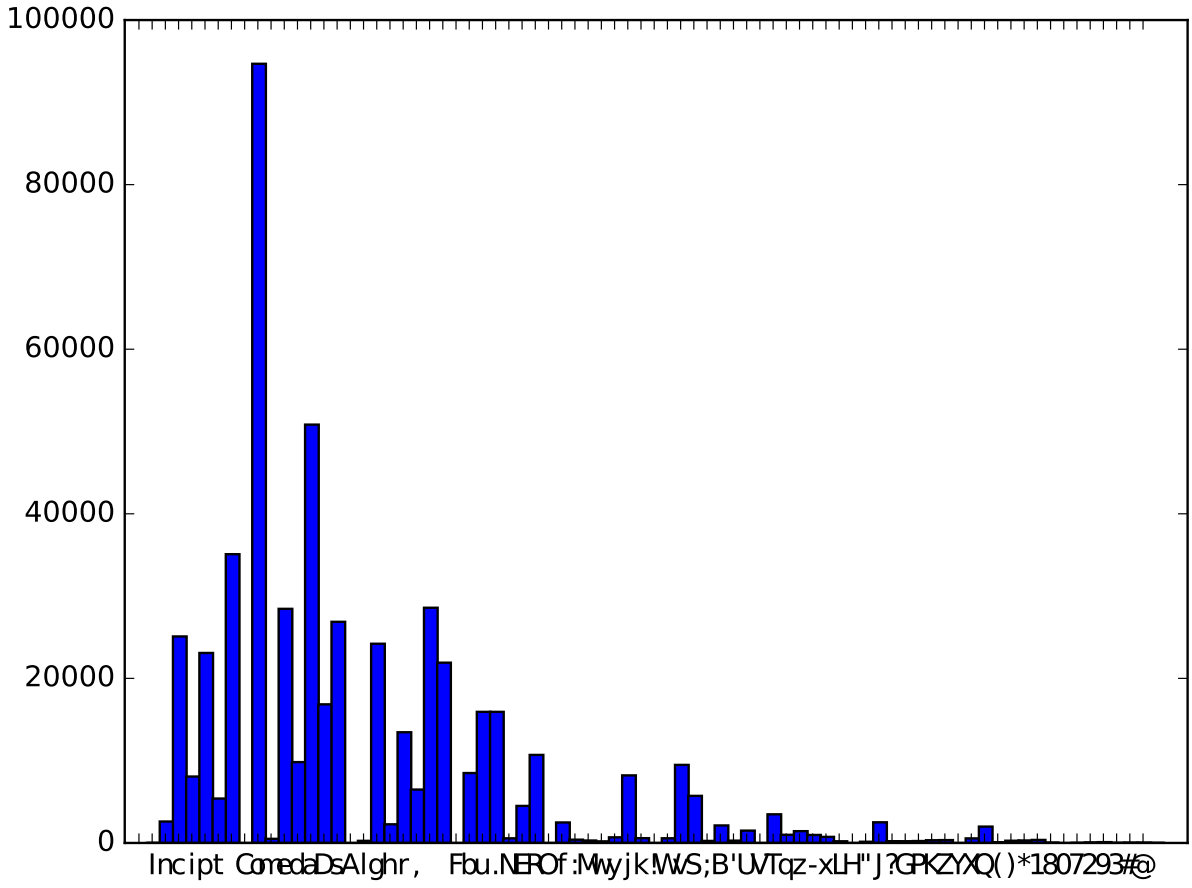

Fig. 3. Histogram of the number of occurrences of the individual characters.

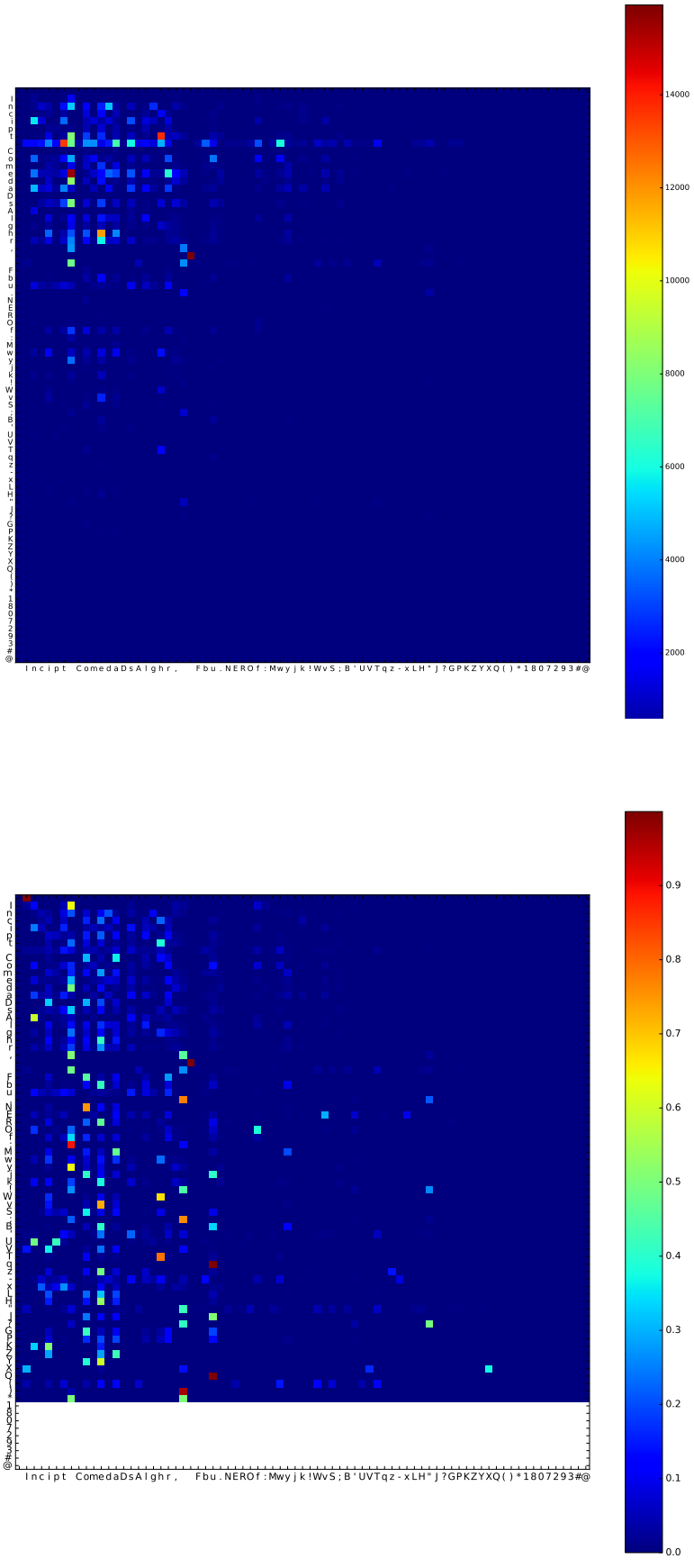

Fig. 4. Matrix of co-occurrence. The values in each row represent how many times the character of that row is followed by the character of the corresponding column. The second matrix is normalized *per row*.

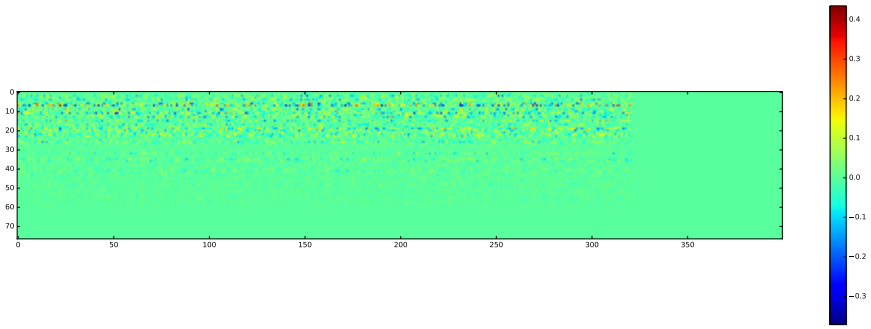

Fig. 5. W matrix, in the case where the input is not concatenated to the output in the ridge regression scheme. It is barely possible to see that the far-rigth side is only 0s, which is the part of the matrix corresponding to the input U.

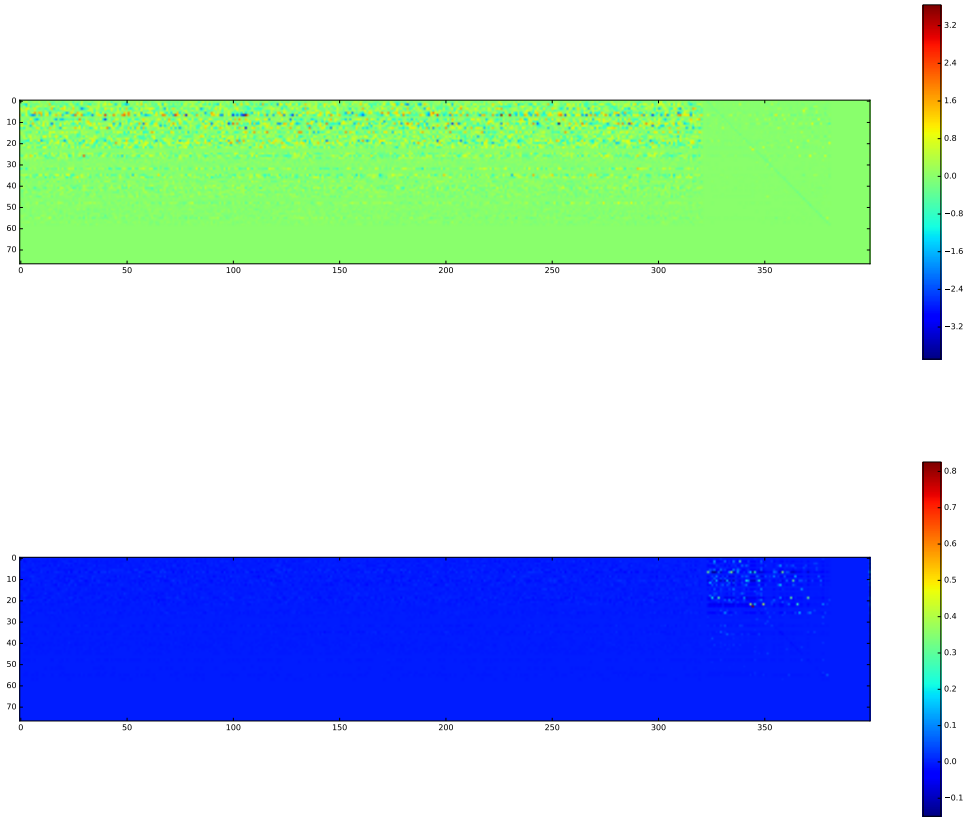

Fig. 6. W matrix for small regularization values (top) or large regularization values (bottom). The part corresponding to the inputs U are the columns on the far right. It is possible to see that their coefficients become small as β becomes small, thus giving more importance to the readouts.