A first step in the direction to learn Saturated SRN

M1 MLDM Internship - Report

Charles FRANCHI[1], Volodimir MITARCHUK[2], and Rémi EYRAUD[3]

[1]University Jean-Monnet, Saint-Etienne, `charles.franchi@etu.univ-st-etienne.fr`
[2]Laboratoire Hubert Curien, Saint-Etienne, `volodimir.mitarchuk@univ-st-etienne.fr`
[3]Laboratoire Hubert Curien, Saint-Etienne, `remi.eyraud@univ-st-etienne.fr`

April 2024 - July 2024

Abstract

The vanishing gradient is a notorious problem when it comes to training Recurrent Neural Networks (RNNs), particularly for Simple Recurrent Networks (SRNs), the first differentiable RNN architecture to be introduced. By design, RNNs recursively process sequential data of variable size, and it is exactly the recursive nature of RNNs that is at the root of gradient problems. The vanishing gradient phenomenon makes a large part of the parameter space inaccessible for SRNs. Nevertheless, a theoretical result proves that these inaccessible regions contain parameters of great interest, notably functions directly related to Deterministic Finite states Automaton (DFA), an expressive and interpretable class of functions. DFA-related SRNs belong to the class of saturated SRNs (SSRNs), that will be introduced in the article. In this work, we have explored possible modifications to the gradient descent algorithm in order to extend as far as possible the horizon of what SRNs can learn in finite precision. Starting from a theoretical result which proves that, in finite precision, it is impossible to learn a SSRN using gradient descent algorithms with a bounded learning step, our strategy focused on how to boost the gradient to avoid the vanishing gradient issue[1].

Keywords: RNN, SRN, Deep Learning, DFA.

1 Introduction

Nowadays, explicability of neural networks model is one of the main drawbacks of Deep Learning. The reason of the final decision of the network is often unclear, and a lot of efforts have been deployed to try to understand it deeply. Therefore, understand the behavior of a neural network remains an important topic in the field of Deep Learning.
Among all Neural Network types, Recurrent Neural Network, and precisely Simple Recurrent Network (SRN) [Elman, 1990] has been of great interest. Recurrent Neural Networks are neural networks based architectures designed to process sequential data, one element at a time, by updating an internal state called the hidden vector. The hidden vector is, in theory, designed to save the important information of what the SRN have seen. A hidden vector is computed recursively with the previous hidden vector and the next input from the sequence.
SRNs key feature - such that they can deal with variable sequence length - is that the parameters are shared all along the process of the sequence. This advantage is also their main drawback: the output has been produced using the same weights matrices at each iteration. Therefore, there is a significant probability to have vanishing or exploding gradients, a phenomenon already well studied in the 90's

[1]Here you can find the code related to all experiments conducted in this work.

[Bengio et al., 1994], [Hochreiter, 1998] .

This problem is particularly visible in the case of Saturated SRNs (SSRN). As defined by Mitarchuk et al. [2023], β-Saturated SRNs are SRNs with a sigmoidal function as activation function, that produces hidden vectors such that all the coordinates of the hidden vectors are closed to the activation function boundaries. In finite precision, that is to say when we operate with floating point numbers[2], a high saturation lead to the boundaries of the sigmoidal function, and the possible values of the hidden vectors are the sigmoidal's boundaries them-self [Mitarchuk et al., 2023].
There are several good aspects to SSRNs : first, they are more stable to noise perturbations. The saturation bring guarantees, under some conditions, that the impact of the noises is negligible [Mitarchuk et al., 2023]. Secondly, Merrill [2019] proved SSRNs have the same expressive power as Deterministic Finite states Automata (DFA), and we already know some SSRN's parameters to simulate the exact behavior of a DFA. It seems possible to find a way to do the other direction[3], and thus have a clear understanding of the SSRN.

Therefore, we have a great interest in using SSRNs. The problem is the saturated region is also the region where the gradient vanished the most. For this reason, we cannot learn a SSRN with a classical gradient descent algorithm. More precisely, we need the set of step of the gradient descent to be unbounded [Mitarchuk et al., 2024].
The main topic of this report focus on our attempt to learn a SSRN. In section 2, we set the context of the internship out. We introduce several formal definitions of the principal objects we work with in section 3. In section 4, we detail the problem of the vanishing gradient in SRN optimization. We then explain two methods, HFO and K-FAC, respectively in section 5 and 6, in our attempt to overcome the vanishing gradient phenomenon. In section 7, we present several important properties of SSRNs and detail the heuristic exploration we made. Next, we present the different datasets used for our experiments in section 8 before talking about the experiments them-self in section 9. Finally, after a short discussion about the work done in section 10, we conclude in section 11.
In appendix A, you can find the list of the abbreviations and notations used all along the report. The proof of Property 1 is moved to Appendix B for reasons of readability.

2 Context of the project

This project was led in the Data Intelligence team of the public research Laboratory Hubert Curien (LabHC) in Saint-Etienne[4]. The Data Intelligence team have the goal to explore and understand data to create knowledge and tools with meaning. It centers its activity in 3 main topics : Machine Learning, Data Mining and Information Retrieval. This project is part of the Machine Learning team, and is aligned with the PhD subject of V. Mitarchuk, working on Saturated SRN [Mitarchuk et al., 2023] [Mitarchuk et al., 2024].
I join this project in the context of a first year internship of the Machine Learning and Data Mining Master's degree in University Jean Monnet of Saint-Etienne. The original subject was about embedding recurrence in Transformers, but we move on the actual subject because we found it more interesting.

3 Objects of study

In this section, we introduce the different mathematical objects we work with: SRN, SSRN and DFA.

3.1 Simple Recurrent Network

We present here a formal definition of SRNs based on the work of Elman [1990]:

Definition 1 *Let u, o, d be 3 integers : u is the input size, o the output size, and d the hidden size. Let $U \in \mathbb{R}^{d\times u}$, $W \in \mathbb{R}^{d\times d}$ and $V \in \mathbb{R}^{d\times o}$ be 3 matrices. Let $b \in \mathbb{R}^d$ and $c \in \mathbb{R}^o$ be 2 vectors. Let σ be a non linear*

[2]All computers use finite precision.
[3]Given a SSRN, find the corresponding DFA.
[4]Laboratoire Hubert Curien, UMR CNRS 5516, Bâtiment F 18 Rue du Professeur Benoît Lauras, 42000 Saint-Etienne

function.
The SRN $\mathcal{R}_P$ with $P = (U, W, V, b, c)$ is the network which, for every $(X_t)_{1\leq t\leq \mathcal{T}}$ sequence of vector in $\mathbb{R}^u$, and every $h_0 \in \mathbb{R}^d$ initial hidden state, recursively compute :

$$\begin{aligned}
&\forall t \in [\![1, \mathcal{T}]\!], \\
&\qquad h_t = \sigma\left(U.x_t + W.h_{t-1} + b\right) \\
&\qquad \hat{y}_t = \sigma\left(V.h_t + c\right)
\end{aligned}$$

$(h_t)_{1\leq t\leq \mathcal{T}}$ are the hidden states, and $(\hat{y}_t)_{1\leq t\leq \mathcal{T}}$ are the output vectors.
For simplicity, we note $\hat{y} := \hat{y}_{\mathcal{T}}$ the last output.

Usually, σ is the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ or the tanh.

3.2 Saturated Simple Recurrent Network

We base our definition of SSRNs on the one given by Mitarchuk et al. [2023]:

Definition 2 *Let $\mathcal{R}_P$ be a SRN with hidden dimension d and a sigmoidal type activation function $\sigma : \mathbb{R} \to [a, b]$ with $a < b \in \mathcal{R}$. We set m to be the middle point of $[a, b]$.*
$\mathcal{R}_P$ is a β-Saturated SRN if there exists $\beta \in [m, b]$ such that for all sequences of vectors $(X_t)_{1\leq t\leq \mathcal{T}}$, we have:

$$\begin{gathered}
\forall k \in [\![1, \mathcal{T}]\!], \quad \forall i \in [\![1, d]\!] \\
|h_k[i] - m| \geq \beta - m
\end{gathered}$$

with $h_k[i]$ the i^{th} coordinates of the vector h_k.
For the purposes of notations, we say $\mathcal{R}_P$ is a Saturated SRN if $\beta = b$.

This definition describes an SRN that produces hidden vectors whose coordinates are close to the bounds of the activation function. For example is the activation function is the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ then $a = 0$, $b = 1$ and $m = 0.5$. If the activation function is tanh, then $a = -1$, $b = 1$ and $m = 0$.

3.3 DFA

In this section, we express a formal definition of a DFA.

Definition 3 *Let Σ be an alphabet of size $|\Sigma|$.*
Let $Q = (q_i)_{0\leq i\leq |Q|-1}$ be a set of size $|Q|$. We call each q_i a state.
Let $\delta : (Q \times \Sigma) \to Q$ a function of transitions between states.
Let $F \subset Q$ be a subset of accepting states.
The Deterministic Finite states Automaton (DFA) $(\Sigma, Q, q_0, \delta, F)$ is the machine going from Σ^* to $\{0, 1\}$ that, giving a word $w \in \Sigma^*$, will recursively compute:

$$\begin{cases}
a_1 = q_0 \\
a_{i+1} = \delta(a_i, w_i) \quad \forall i \in [\![2, |w|]\!]
\end{cases}$$

and returns 1 if $a_{|w|} \in F$, 0 otherwise.
$q_0 \in Q$ is called the initial state.

3.4 Loss function

To optimize a SRN, we use the next format of loss function:
Let $S = \{(\mathbf{X}^{(i)}, y^{(i)}) \,|\, 1 \leq i \leq m\}$ be a dataset with $|S| = m$, $\mathbf{X}^{(i)} = (X_t^{(i)})_{1\leq t\leq \mathcal{T}^{(i)}}$ a set of sequence of vectors and $y^{(i)} \in \{0, 1\}$ the label.
Let $\mathcal{R}_P$ be a SRN. We note $\theta = \mathbf{vec}(P)$ and n the number of parameters of $\mathcal{R}_P$. We define the loss function:

$$\begin{aligned}
L : ([0, 1] \times \{0, 1\}) \times \mathbb{R}^n &\longrightarrow \mathbb{R} \\
(\hat{y}^{(i)}, \quad y^{(i)}), \qquad \theta &\longmapsto L(\hat{y}^{(i)}, y^{(i)}; \theta)
\end{aligned}$$

with $\hat{y}^{(i)}$ the output of $\mathcal{R}_P$ as defined in definition 1.

4 Vanishing Gradient phenomenon

As mentioned in the introduction, the vanishing gradient is a common problem of RNNs. The main structure of the network induce it, that's why it is a difficult problem to get through. This is one of the reason SRNs are much less widely used than more complex recurrent network, like Gated Recurrent Unit (GRU) [Gao and Glowacka, 2016], Long Short Term Memory (LSTM) [Hochreiter and Schmidhuber, 1997] or Transformers [Vaswani et al., 2017]. GRUs and LSTMs use gates to make the network more complex, with more processing for a single element of sequences. Transformers use attention mechanism, which store the hidden vector at each iteration to use all of them for the final prediction. With those techniques, the gradient generally does not vanish, or at least the problem is less significant [Noh, 2021]. However, the price to be paid for getting rid of gradient-related problems is in terms of loss of interpretability. Indeed, the proposed architectures are far more complex than SRNs. Hence our focus on SRNs.

To keep SRNs, we have to deal with the vanishing gradient issue more directly. Along all the approaches tested to overcome this problem, several of them add the curvature information of the loss as well as the gradient, like using the Hessian or the Gauss-Newton approximation [Martens et al., 2010].
These kind of approaches have to face two issues. First, the size of these curvature matrices is the number of parameters squared (n^2), and since a neural network can be composed of thousand of parameters, these matrices are not usable in practice. Secondly, most of the techniques with the curvature, like Newton's Method, need to inverse the curvature matrix. In practice, inverting a big matrix is costly and the gain of the method does not worth it.

We decide to focus on two methods with two different approaches.
The Hessian-Free Optimization (HFO) add the curvature information to the gradient as a term.
The Kronecker-factored Approximate Curvature (K-FAC) boost the gradient by multiplying it with the inverse of the curvature.
We detailed both in the next sections.

5 Hessian-Free Optimization

In this section we sum up the principle of the Hessian-Free Optimization (HFO), and explained its main drawbacks for our project.

The HFO is a loss optimizer, originally presented by Pearlmutter [1994] and improved by Martens et al. [2010], that takes into account the Hessian during the optimization steps. Martens and Sutskever [2011] studied it for the case of SRNs.

This method is based on the Conjugate Gradient (CG), which is an optimization method known to work very well for quadratic function. Suppose we want to minimize a function f :

$$f(x) = \frac{1}{2}x^T Ax + b^T x + c$$

with $A \in \mathbb{R}^{n\times n}$ symmetric and Positive Semi-Definite (PSD), $b \in \mathbb{R}^n$ and $c \in \mathbb{R}$. Thus, the CG method compute conjugate directions of ∇f and optimize the function in each direction. Therefore, we obtain the minimum of f when we have explored all the dimensions, that is to say in n step.

One important hypothesis is that A is PSD. This implies f is convex, and thus the global minimum exist.

The idea of HFO is to take the second order Taylor approximation of the loss, and to find the best direction to optimize.
Let L be the loss function. For convenience, regarding the definition in section 3.4, we write the loss $L(\theta)$ instead of $L(\hat{y}, y; \theta)$, as we do not do specific operations on the other terms. The second order approximation is :

$$\begin{aligned} L(\theta_n + \delta_n) \approx L(\theta_n) + \nabla L(\theta_n).\delta_n \\ + \delta_n^T.H_L(\theta_n).\delta_n \end{aligned}$$

Therefore, instead of directly minimizing L, we can minimize this function :

$$\delta_n = \underset{\delta}{\arg\min}\, L(\theta_n) + \nabla L(\theta_n).\delta + \delta^T.H_L(\theta_n).\delta$$

$$\theta_{n+1} = \theta_n + \delta_n$$

We then use the CG method to find δ_n, with $A = 2H_L(\theta_n)$, $b = \nabla L(\theta_n)$ and $c = L(\theta_n)$.

This method was claimed to be able to overcome the two main problems detailed at section 4. Of course, we do not need the inverse of the Hessian here. For the first problem however, their idea to overcome it is to evaluate the Hessian in the specific desire direction at each step of the CG algorithm. Since we can see the Hessian as the gradient of the Jacobian, we can approximate it by:

$$H_L(\theta_n)\delta_n \approx \frac{\partial}{\partial \epsilon}\nabla L(\theta_n + \epsilon\delta_n)$$

with ϵ close to 0. In that way, we only compute a vector at each iteration instead of a full matrix: we win time and space complexity.

For the CG to be efficient, we have 2 conditions on H_L : it has to be PSD and symmetric. Usually, since all function used in a SRN are infinitely differentiable, the Hessian is Symmetric. However, we cannot assure it is PSD: generally, a neural network is not convex. To overcome this problem, Martens et al. [2010] used the Gauss-Newton matrix instead of the Hessian, which is an approximation of it with several benefits, like the fact it is PSD.

Finally, they add some extra elements to the method, like a regularization argument to the Hessian λId. Also, for the SRN's version, Martens and Sutskever [2011] used a more complex version. For example, as well as a λId term, they add the distance between the hidden states of two consecutive parameters updates for regularization. It helps to control the amplitude of each step.

Regarding the experiments given by Martens and Sutskever [2011], this method should have been interesting. However, several problems occur:
First, the way HFO deal with the vanishing problem does not lead to an unbounded gradient step method. Generally, if the gradient is bounded, the Hessian is too. This is always the case at the limits of a bounded function (the region we are interested in with SSRNs), so to add the Hessian as a simple term to the gradient will not "unbound" the step, and will not help to learn a SSRN in that way.
Moreover, even if this algorithm perform a better step regarding the same amount of running time for a Stochastic Gradient Descent, it has to use a fix batch during all the process of Conjugate Gradient (CG). Thus, it does a better step but in a very specific direction. That means we have a more chaotic directions in which we optimize, or we at least have to re-think the batch size to be adapted to this new criterion.
Another problem is the number of steps of CG. To be efficient, we have to perform the all descent: we reach the actual minimum only at the last step, and we cannot assure to have a good approximation beforehand [Martens et al., 2010]. The number of steps is the number of parameters of the network, which can be huge. It makes the computation way slower, and we cannot do a systematic good approximation by stopping it earlier [Martens and Grosse, 2015].
Finally, in terms of pure performance, HFO does not seem to be more efficient than a well-tuned SGD with momentum [Martens and Grosse, 2015].

For all this points, and especially the first one, we decide to not go further with this method.

6 Kronecker-factored Approximate Curvature

In this section, we sum up the Kronecker-factored Approximate Curvature (K-FAC) optimization method for a classic neural network and for a SRN, and discussed its drawbacks.

The K-FAC is an approximation first developed by Heskes [2000], improved by Martens and Grosse [2015], and then adapted for SRNs by Martens et al. [2018]. The main goal is to approximate the inverse of the Fisher matrix such that we can use the Natural Gradient Descent (NGD) optimizer. We will now detailed this two concepts.

The Fisher matrix catches the curvature information of a loss function, which can be seen as the covariance of the score, or the log likelihood curvature matrix [Amari, 1998]. Let L be a loss function. Like in section 5, for convenience, regarding the definition of section 3.4, we write the loss $L(\theta)$ instead of $L(\hat{y}, y; \theta)$, as we do not do specific operations on the other terms. The Fisher matrix is defined by :

$$\mathcal{I}_L(\theta) := \mathbb{E}[\left(\frac{\partial}{\partial\theta}\log L(\theta)\right)\left(\frac{\partial}{\partial\theta}\log L(\theta)\right)^T]$$

with element-wise Expected value. We can notice this matrix has the same size as the Hessian. Furthermore, there is a relationship between this matrix and the Generalized Gauss-Newton matrix [Martens and Grosse, 2015].

The Fisher matrix is used in the Natural Gradient Descent (NGD). NGD is an optimization algorithm similar to the Newton's method, but which uses the Fisher matrix instead of the Hessian. A step of the NGD is defined by :

$$\theta_{n+1} = \theta_n - \eta_n . \mathcal{I}_L(\theta)^{-1} . \nabla L(\theta)$$

with θ_0 a chosen initialization, and $(\eta_n)_n$ a sequence of adaptive[5] learning rates.

In practice, NGD can be really efficient on several scenario over some conditions on the network [Zhang et al., 2019]. However, it suffers of the same problems as other 2^{nd} order methods, that is to say the problems mentioned at the beginning of section 4. K-FAC get rid of these two problems.

6.1 K-FAC for a classic Neural Network

Based on the notations of Martens and Grosse [2015], let consider a neural network of ℓ layers, with each layer i defined as :

$$\begin{aligned} s_i &:= W_{i-1} a_{i-1} \\ a_i &:= \sigma(s_i) \end{aligned}$$

with W_i a matrix, a_i expressed in homogeneous coordinates[6] and a_0 the input vector X. Thus, we can see the set of parameters θ as $(\mathbf{vec}(W_1)^T \dots \mathbf{vec}(W_\ell)^T)^T$. We also define :

$$g_i = \frac{\partial \log L(\theta)}{\partial s_i}$$

Therefore we have:

$$\frac{\partial \log L(\theta)}{\partial W_i} = g_i a_{i-1}^T$$

The first idea of K-FAC is to express the Fisher matrix $\mathcal{I}$ as a $\ell \times \ell$ block-matrix:

$$\begin{aligned} &\forall (i,j) \in [\![1, \ell]\!]^2 \\ \mathcal{I}_{i,j} &= \mathbb{E}[\left(\frac{\partial \log L(\theta)}{\partial W_i}\right)\left(\frac{\partial \log L(\theta)}{\partial W_j}\right)^T] \\ &= \mathbb{E}[\mathbf{vec}(g_i a_{i-1}^T)\mathbf{vec}(g_j a_{j-1}^T)^T] \\ &= \mathbb{E}[(a_{i-1} \otimes g_i)(a_{j-1} \otimes g_j)^T] \\ &= \mathbb{E}[a_{i-1} a_{j-1}^T \otimes g_i g_j^T] \\ \mathcal{I}_{i,j} &\approx \mathbb{E}[a_{i-1} a_{j-1}^T] \otimes \mathbb{E}[g_i g_j^T] \end{aligned}$$

with $\otimes$ the Kronecker product [Van Loan, 2000].

According to Martens and Sutskever [2011], this final approximation is good in practice and lead to accurate results.

The second main approximation of the K-FAC method comes when we inverse the Fisher matrix : if we make the assumption the $\frac{\partial \log L(\theta)}{\partial W_i}$ are related to $\frac{\partial \log L(\theta)}{\partial W_{i+1}}$ with a Gaussian transformation, we can express the Fisher matrix with a simpler structure and compute it efficiently with the parameters of these Gaussian transformations [Martens and Grosse, 2015].

This second approximation seems actually good in practice [Martens and Grosse, 2015].

Furthermore, to make K-FAC really efficient, they also add momentum, and a $(\lambda + \eta_n)Id$ to the Fisher matrix as regularization coefficients. The λ correspond to the l_2-norm regularization, and η_n is a coefficient re-scaled at each step if the current step is too big. Thus, if we continue doing big steps, we increase the model complexity.

6.2 K-FAC for SRNs

In the case of a SRN, things get more complicated, due to the fact that the hidden state transition matrix W[7] is applied through all iterations. Therefore, when applied to the parameter of W, the K-FAC method has to be adapted to the recurrence. However, the original method can be applied as it is to the others parameters of the SRN, that is to say the matrices U, V and the bias vectors b and c.

The major ideas are the same as for a classic neural network, but the approximations are quite different. We define :

$$\begin{aligned} \tilde{s}_i &:= U x_i + W h_{i-1} + b \\ \tilde{g}_i &:= \frac{\partial L(\theta)}{\partial \tilde{s}_i} \end{aligned}$$

and thus we can notice that:

$$\frac{\partial L(\theta)}{\partial W} = \sum_{i=1}^{\mathcal{T}} \tilde{g}_i h_i^T$$

[5] $(\eta_n)_n$ is generally decreasing.

[6] Vectors extended with a last dimension, with value 1. That way, we include the bias in W_i.

[7] We use the notations of definition 1.

Therefore, the Fisher matrix can be expressed:

$$\mathcal{I} = \mathbb{E}[\mathcal{I}_{\mathcal{T}}]$$

$$\begin{aligned}\mathcal{I}_{\mathcal{T}} &= \sum_{i=-\mathcal{T}}^{\mathcal{T}} (\mathcal{T} - |d|)\mathbb{E}[h_k h_{|i-k|}^T \otimes \tilde{g}_k \tilde{g}_{|i-k|}^T] \\ &\approx \sum_{i=-\mathcal{T}}^{\mathcal{T}} (\mathcal{T} - |i|)\mathbb{E}[h_k h_{|i-k|}^T] \otimes \mathbb{E}[\tilde{g}_k \tilde{g}_{|i-k|}^T]\end{aligned}$$

with $k = 0$ if $i \geq 0$ and $k = \mathcal{T}$ if $i < 0$ [Martens et al., 2018].

To make this approach correct and efficient, we have to make a lot of hypotheses - in this case "the derivatives" refer to the elements $(\tilde{g}_i h_i^T)_{1 \leq i \leq \mathcal{T}}$:

1. The values of the derivatives through times have to be independent with the sequence length $\mathcal{T}$.

2. The derivatives at time t and at time s must only be related by the time passed $i = t - s$.

3. To compute the inverse, we suppose the derivatives are structured with a Linear Gaussian Graphical Model (that means, for all i, $\tilde{g}_i h_i^T = \Psi \tilde{g}_{i-1} h_{i-1} + \epsilon_i$, with Ψ a square matrix and ϵ_i some noises)[8].

4. We have to make a last assumption on the temporal dependency : the original paper propose two options, the only one possible in our case is to suppose the sequence could be infinitely long.

Some of this assumption are not aligned for what we need. Since we know SSRNs are equivalent to DFA, we can already deduce some drawbacks for some hypotheses:

- To simulate an automaton, the network have to remember the current state of the equivalent automaton when he read a word. Thus, the hidden vector have to contain this information. In some cases, the current state depend of the sequence length in general, whether it due to a specific sequence size, or because some parity of the DFA. Therefore, in practice, the hidden vector h_i could be dependant of $\mathcal{T}$. The first hypothesis supposes this independence, and seems therefore not compatible with SSRNs.

- For similar reason as before, depending of the current states, the next possible states are not the same, and thus we cannot easily supposed the derivatives $(\tilde{g}_i h_i^T)_{1 \leq i \leq \mathcal{T}}$ depend only on the passed time and not the current position.

These problems do not seem easy to fix. It implies deeper research to find a way to not use them. It should be a way to explore, but this is a full stand-alone subject because of its difficulty.
Unfortunately, it makes high possibility the method will not work efficiently.

We tried to find an implementation of this method, to test it and confirm or infirm our hypotheses, but the only one we found dealing with SRNs was outdated and we do not manage to make it work in a reasonable time.

7 Saturated Simple Recurrent Network

In this section, we explained the main result about Saturated Recurrent Neural Network (SSRN) in the context of finite precision.

7.1 Stability of SSRN

In finite precision, the number of values that can take the sigmoidal function is limited, and thus due to overflow and underflow phenomenon, this function has a binary output for values large enough. For instance, with 32-bits float numbers, we can observe that for all $x \in \mathbb{R}_+$ such that $x \geq 89$ $\sigma(-x) = 0$ and $\sigma(x) = 1$ [Mitarchuk et al., 2024].
Therefore, if we reach this region of the sigmoidal function, the parameters can change a bit without having any impact on the behavior of the SSRN, resulting in a SRN stable to input and parameter noises.
However, we do not need to be as specific to obtain a good stability. In Theorem 3.3 in the paper of Mitarchuk et al. [2023], they expose some conditions on the β-Saturated SRN to obtain a full region where to add noises to the network's weights does not impact the hidden

[8]This is a similar assumption as for the original K-FAC method.

vectors to much, and thus the final prediction. One important condition to have this stability is : $\beta > \sqrt{1 - \frac{1}{||W||}}$ with W as defined in definition 1. Consequently, we do not especially need the extreme case $\beta = 1$.

7.2 DFA to SSRN

We know there is an equivalence in terms of expressive power between SRN and DFA: Merrill [2019] prove the class of languages that can simulate a SSRN - and more generally that can simulate a SRN in finite precision - are the Regular Languages, which is precisely the class of the Languages that can simulate a DFA. That means, in theory, from a specific DFA, we can construct a SSRN, and conversely.
The first way - DFA to SSRN - have been proved by Mitarchuk et al. [2024] where the activation function used is the sigmoid.

We now detail this construction. Please note we use the Python convention for matrix, where the indexation start at 0.
Let $\sigma(X) = \frac{1}{1+e^{-X}}$ be the sigmoid function. Since we are in finite precision, there exist a number J such, $\forall x \geq J$, $\sigma(x) = 1$ and $\sigma(-x) = 0$ [Mitarchuk et al., 2024].
Let $(\Sigma, Q, q_0, \delta, F)$ be a DFA. We define $\mathcal{R}_P$ the SSRN with σ as activation function, with the hidden size $d = |Q| \times |\Sigma|$, the output size $o = 1$ and the input size $|\Sigma|$.
To make an input word readable by the SSRN, every letters of the word are one-hot encoded, that is to say, given the i^{th} letter of Σ, we create a vector of size $|\Sigma|$ where the i^{th} coordinate is 1, and the other coordinates are 0.

We construct a SSRN with a binary behavior, that means we reach the region of the activation function such that the hidden vectors can only be composed of 0 and 1. They are of size $d = |Q| \times |\Sigma|$ and every group of $|\Sigma|$ coordinates represent a single state. We thus define the following weights :

- U is
$$2J \times \begin{pmatrix} Id_{|\Sigma|} \\ \vdots \\ Id_{|\Sigma|} \end{pmatrix}$$
with $|Q|$ times the Id.

- W is a square matrix of size $d \times d$. Every columns represent the transitions from a specific state. Thus, the columns are identical by group of $|\Sigma|$ successive columns, like the hidden vectors. If $W_{i,j}$ represent a valid transition - the line j representing the letter seen -, we set $W_{i,j} = -J$. If not, we set $W_{i,j} = -3J$.

- V is a line vector with also the columns equal by group of $|\Sigma|$ consecutive columns. For every states, we set J as the value of V if the corresponding state is accepted, and $-J$ else.

- The biases b and c are 0 vectors.

This construction make h_t have only one coordinate equal to 1, and all the other equal to 0. The current hidden vector represent the column of W used, thus the current state of the equivalent DFA. Ux_t is a vector corresponding to the letter seen. Finally, the only positive element of the vector $Ux_t + Wh_{t-1}$ is the one where we find $2J - J$, thus the only position where we have seen the correct letter as input, and where we use the correct transition.
When we applied the activation function, we then obtain the hidden vector, with only one coordinate being a 1 and the others being 0.

For clarity, let's take an example.

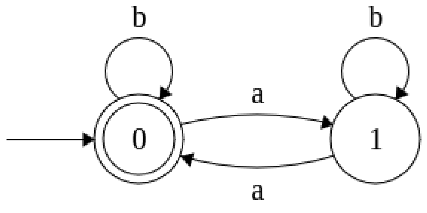


Figure 1: An example of DFA: "all the words with an even number of 'a'". The final state is represented by the double circle.

The DFA of Figure 1 is built on an alphabet Σ of size 2, with $|Q| = 2$ states. It is well defined since every transitions is explicitly given for every letter and states[9].
The hidden vector $(1, 0, 0, 0)$ and $(0, 1, 0, 0)$ represent the state q_0, whereas $(0, 0, 1, 0)$ and $(0, 0, 0, 1)$ represent the state q_1.

[9]In a case some transitions are not explicitly given, we have to add an extra rejected "bin" states, which is a sink where all these unknown transitions go.

The equivalent SSRN is, using the previous construction, defined with :

$$U = J\begin{pmatrix}2 & 0\\0 & 2\\2 & 0\\0 & 2\end{pmatrix}, W = -J\begin{pmatrix}3 & 3 & 1 & 1\\1 & 1 & 3 & 3\\1 & 1 & 3 & 3\\3 & 3 & 1 & 1\end{pmatrix},$$
$$V = J\begin{pmatrix}1 & 1 & -1 & -1\end{pmatrix}$$

For instance, suppose we have a current word w ="aba". After having read "ab", the original automaton is in state 2, and it means our SSRN current hidden vector is $h_2 = (0,0,0,1)$. We finally read an "a", so the corresponding one-hot encoded input vector is $x_3 = (1,0)$. The next hidden vector is thus :

$$\begin{aligned}h_3 &= \sigma\left(Ux_3 + Wh_2\right)\\ &= \sigma(J\left(\begin{pmatrix}2\\0\\2\\0\end{pmatrix} - \begin{pmatrix}1\\3\\3\\1\end{pmatrix}\right)) = \sigma(J\begin{pmatrix}1\\-3\\-1\\-1\end{pmatrix})\\ &= \begin{pmatrix}1\\0\\0\\0\end{pmatrix}\end{aligned}$$

It means we are in state 0, this is correct regarding the original DFA.

Since this is the last letter of the current word, we then look at the acceptance :

$$y = \sigma(Vh_3) = \sigma(J \times 1) = 1$$

So "aba" is accepted.

Mitarchuk et al. [2024] proved this construction is well defined and actually simulate any DFA accurately. In the paper, they precised it is possible to use the tanh instead of the sigmoid, but do not explicit the method. Since we will need it in section 7.3, we explain it after.

Let $\mathcal{R}_P$ with $P = (U, W, V, 0, 0)$ be a SRN simulating a DFA with the sigmoid function. Using the tanh instead of the sigmoid with the current construction result in changing the value of the hidden vectors: instead of having 0 - 1 vectors, we now have (-1) - 1 vectors. This will impact the recursive steps:

$$\begin{aligned}h_t &= \tanh\left(U.x_t + W.h_{t-1} + b\right)\\ \hat{y}_t &= \tanh\left(V.h_t + c\right)\end{aligned}$$

Thus, we have to change W and V.

In fact, we need to find back 0-1 vector, so we want to go from (-1) - 1 vectors to 0 - 1 vectors. These coordinates modification of the hidden vectors is the same as a change of base between the base $\mathcal{B}_{\text{tanh}}$:

$$\mathcal{B}_{\text{tanh}} = \begin{pmatrix}1 & -1 & \dots & -1\\-1 & 1 & & -1\\ \vdots & & \ddots & \\ -1 & -1 & \dots & 1\end{pmatrix}$$

and the canonical base Id, which are the 0-1 vectors we need.
Well, as the base change from Id to $\mathcal{B}_{\text{tanh}}$ is $\mathcal{P}_{Id\to\text{tanh}} = \mathcal{B}_{\text{tanh}}$, the base change from $\mathcal{B}_{\text{tanh}}$ to Id is $\mathcal{P}_{\text{tanh}\to Id} = \mathcal{B}_{\text{tanh}}^{-1}$.

Property 1 *Let $n \neq 2$ be an integer. We define:*

$$\mathcal{B}_{\text{tanh}} = \begin{pmatrix}1 & -1 & \dots & -1\\-1 & 1 & & -1\\ \vdots & & \ddots & \\ -1 & -1 & \dots & 1\end{pmatrix}$$

the matrix with 1 on the diagonal and -1 otherwise.
Therefore, we have :

$$\mathcal{B}_{\text{tanh}}^{-1} = \frac{1}{2}\begin{pmatrix}1-a_n & -a_n & \dots & -a_n\\-a_n & 1-a_n & & -a_n\\ \vdots & & \ddots & \\ -a_n & -a_n & \dots & 1-a_n\end{pmatrix}$$

the matrix with $\frac{1-a_n}{2}$ on the diagonal and $\frac{-a_n}{2}$ otherwise, with $a_n = \frac{1}{n-2}$.

We prove this property in appendix B.
We finally obtain the equations:

$$\begin{aligned}h_t &= \tanh\left(U.x_t + W.\mathcal{B}_{\text{tanh}}^{-1}.h_{t-1} + b\right)\\ \hat{y}_t &= \tanh\left(V.\mathcal{B}_{\text{tanh}}^{-1}.h_t + c\right)\end{aligned}$$

Thus, the new parameters of the network such that the SSRN simulate the same DFA with the tanh function instead of the sigmoid are $\tilde{P} = (U, W\mathcal{B}_{\text{tanh}}^{-1}, V\mathcal{B}_{\text{tanh}}^{-1}, 0, 0)$.

To conclude this part, it is important to notice there is no clear construction to go back from any SSRN to a DFA until now. Some theoretical trails will be discussed in section 10.

7.3 Use heuristic to learn SSRN

In this part, we focus on the heuristic knowledge we have about SRNs and SSRNs to create an other approach to learn a SSRN. The objective is to use this knowledge to change the behavior of the gradient descent.

Since we know a way to simulate a DFA with a SSRN (see section 7.2), we know what its shape could be. Our idea is to add some strong constraints on the SRN to incite the network to learn similar parameters as the DFA to SSRN construction, such that we can interpret it and understand its behavior.

Let $\mathcal{R}_P$ with $P = (U, W, V, b, c)$ be a SRN. Then:

- We force U to be:

$$u \times \begin{pmatrix} Id_{|\Sigma|} \\ \vdots \\ Id_{|\Sigma|} \end{pmatrix}$$

with u a learnable parameter.

- We force W to have equal columns by group of $|\Sigma|$ columns. We thus decrease the number of parameters of W by a factor $\frac{1}{|\Sigma|}$.

- We force all the initial parameters of W to be negative.

- We set the biases b and c to 0.

It is important to note we can easily recover Σ looking at the dataset: we can extract all the different symbol used along the train set, and suppose any new symbol encounter further does not belong to the language.
We then train $\mathcal{R}_P$ as a classical SRN.

In section 10, we present all the other ideas we had no time to experiment yet.

We implemented this method in PyTorch. Since the sigmoid function is not implemented for SRN in PyTorch, we have to apply this constraints in the case of the tanh activation function.
However, this is not a problem here: with the very regular shape of $\mathcal{B}_{\text{tanh}}^{-1}$, if W has equal columns by group of $|\Sigma|$ columns, $W\mathcal{B}_{\text{tanh}}^{-1}$ has them too. So it finally does not change anything about our constrained SRN, whether it be the sigmoid or the tanh function.

8 Datasets

In this part, we focus on the datasets used to experiment our models.
Since SSRNs are equivalent to DFA, learning SSRNs from data reduces to learning DFA. Therefore, to conduct our experiments we need samples of formal languages.

8.1 Create our own data

The first idea was to find a generic way to create a dataset corresponding to any DFA needed. In that way, we should be able to try to learn any DFA we need.

To do so, we create a class of object named `DFA`[10], with only 2 principals arguments - we use the notation of definition 3:

- δ, as a $|Q| \times |\Sigma|$ size matrix. The states are thus symbolized by numbers from 0 to $|Q| - 1$[11], and we take the first $|\Sigma|$ letters of the classic alphabet order for the DFA alphabet. The transition $(q_i, \Sigma_j) \rightarrow q_k$ is represented by $\delta_{i,j} = k$

- F, as a $|Q|$ size vector, with, for every states q_i, $F_i = 1$ if q_i is an accepted final state, 0 else.

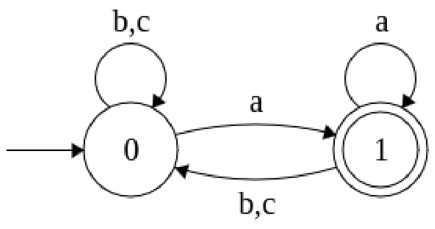


Figure 2: An other example of DFA : "all the words ending with an 'a'".

[10] The code is available on GitHub here.
[11] As we use the Python convention for matrix, indexation start at 0.

As an example, we propose to consider the DFA illustrated in Figure 2. The corresponding parameters are:

$$\delta = \begin{array}{c} \\ q_0 \\ q_1 \end{array} \begin{array}{ccc} \text{a} & \text{b} & \text{c} \\ \end{array}$$

$$\delta = \begin{array}{c|ccc} & \text{a} & \text{b} & \text{c} \\ q_0 & 1 & 0 & 0 \\ q_1 & 1 & 0 & 0 \end{array}$$

$$F = \begin{array}{c} q_0 \\ q_1 \end{array} \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

For instance, $\delta_{1,2} = 0$ is equivalent to the transition $(q_1, \text{c}) \to q_0$, and $F_0 = 0$ indicate q_0 is rejected.

Then, from this class, we designed different probabilities distribution of the transitions δ. This allows us to randomly generate words as follows:

- The length of the generated word follow a Poisson law of hyper-parameter λ.
- We start at the initial state 0. Then, for each step, we choose randomly a new letter according to the probability distribution of the current state, and go to the corresponding new state.
- When we reach the desired length, we look if the current state is accepted or rejected regarding F.

For example, if the probability for the DFA 2 is:

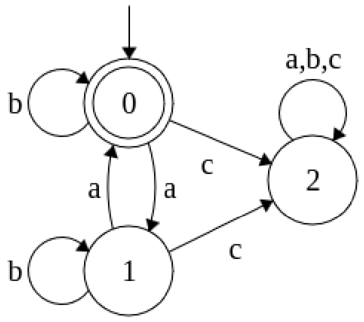
$$\mathbb{P} = \begin{pmatrix} 1/2 & 1/4 & 1/4 \\ 1/4 & 1/4 & 1/2 \end{pmatrix}$$

Then a common generated word with the hyper-parameter $\lambda = 5$ will be "acaca" since it has an high probability to occur.

We have 2 types of predefined probability distribution:

1. The "equal" distribution : given a state, every letter has the same probability to occur. In fact, this distribution does not depend of the DFA anymore: with this, we generate pure random words.
2. No obvious negative : if we have in a DFA a "bin", a non-final sink for a lot of transitions, we do not want to generate too much negative elements going to this sink. It often express obvious negative for the language, and we want our dataset to contain relevant data.
   We do so by putting a 0 probability for a maximum of transitions going to it.

For example, if we add the letter "c" to the alphabet of the DFA 1, we can construct :



Obviously, any word with a "c" will be rejected. We thus want to generate relevant words. The "no obvious negative" probabilities described are :

$$\mathbb{P} = \begin{pmatrix} 1/2 & 1/2 & 0 \\ 1/2 & 1/2 & 0 \\ 1/3 & 1/3 & 1/3 \end{pmatrix}$$

That way, we are insured to not generate words with "c" in a first time[12].

Finally, with the selected probability distribution, we generate words and labels according to the DFA.
After setting the training data set size to N, we randomly generate words until the number of positive labeled words reaches $N/2$[13]. Based on the difficulty of generating positive examples, we either remove negative elements or modify the probability distributions on the DFA edges to generate more negative examples, with the aim of obtaining $N/2$ words with negative labels.

From the algorithmic point of view, the method of generating synthetic training data we described above is efficient. On average a training set of 10,000 elements was generated

[12]The last row cannot be reach, so the equal probabilities put are just for "safety".
[13]$N/2$ or the desire proportions if we do not want a balanced distribution.

in seconds. This ease of use and low algorithmic cost has enabled us to train SRNs on data from several DFA. We then found that some DFA were learnable by SRN with gradient descent, while others were not. For example we did not succeed in learning the DFA 1, but the DFA 2 was learned in less than 10 epochs. This observation led us to work with the MLRegTest benchmark, which we describe in detail in the following paragraph.

8.2 MLRegTest

[Van der Poel et al., 2023] brings a double contribution, presenting a hierarchical classification of regular languages - the class of languages that can simulate a DFA -, and providing the community a benchmark of 1,800 samples of regular languages, spanning the different complexities. This benchmark was specifically designed for machine learning applications, as every sample is composed of a well balanced training set, a validation and tests sets.
In addition to synthetic data, the benchmark is also providing the DFA associated with every language data set.

This dataset correspond to our requirements. We therefore adopt it for our experiments.

9 Experimentation

In this part we detailed the experiments we made to test the constrained SRN.

We train two SRNs at the same time, one constrained, and one not. From both, we extract several metrics all along the training:

- The ℓ_2 norm and the ℓ_∞ norm of the vectorized parameters gradient of the loss.
- The distance from the current parameters to the target SSRN. The target is, given the original DFA of the generated dataset, the corresponding SSRN constructed as described in section 7.2.
- The distance from the current parameters to the line between the target SSRN and the point 0 of the parameters space.

  This statistic is to check if we learn in the good direction: even if we get closer to the parameters expected, it could go in the wrong directions. By checking the distance to this line, we assure we stay near the good "parameters' line".
- The loss values.
- The accuracy on the test set.

We selected 9 language sample from the 1,800 of MLRegTest benchmark, that faithfully span the different language complexity[14] and use different alphabet's size.
On those data, we test several hyperparameters: we tried different learning rates, different batch sizes, different optimizers (Adam and SGD), and different losses: the Binary Cross Entropy (BCE), and 2 custom losses - BCE & the Inverse of the parameters' norm, and what we called the Nth Root Loss.

BCE & the Inverse of the parameters' norm is the BCE loss with $\lambda\frac{1}{||\theta||}$ as a regularization term, with θ the vector of all the parameters of the network, and λ an hyperparameter. It is supposed to help the network to learn big parameters and thus to become saturated.

The Nth Root Loss is similar to the BCE loss, but instead of using the log function, we use the n^th root:

$$L(\hat{y}, y; \theta) = (1-y)\sqrt[n]{\hat{y}} + y\sqrt[n]{1-\hat{y}}$$

We do not have any guarantees about this loss. Nevertheless the intuition was to design a loss which penalizes even small deviations. In Figure 3, we display the behavior of different loss functions as n^{th} root by varying $n \in \{1/2, 1, 2, 3, 4, 5, 6\}$

[14]The complexity defined by MLRegTest itself.

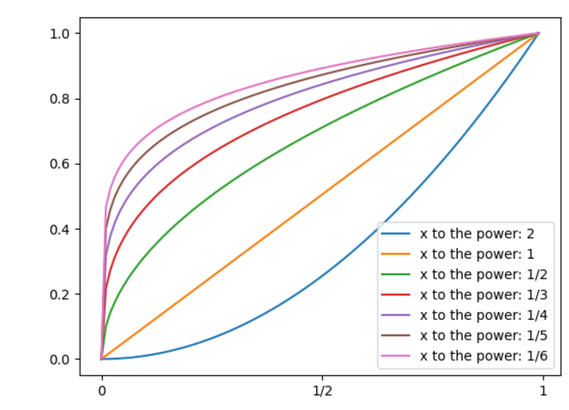


Figure 3: Plot of different Nth Root Loss functions on the interval $[0, 1]$.

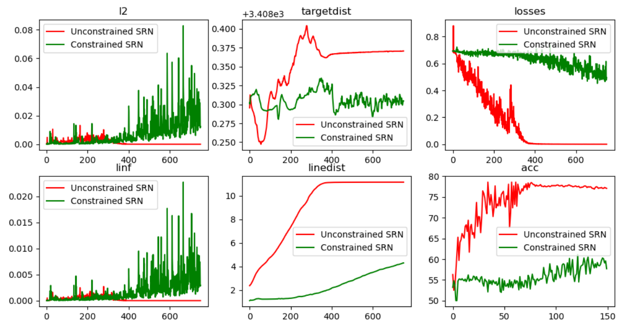


Figure 4: Test on 16.16.SP.2.1.0 from ML-RegTest. Parameters : learning rates = 0.01 ; Number of epoch = 150 ; Batch size = 200 ; optimizer = Adam ; loss = BCE.

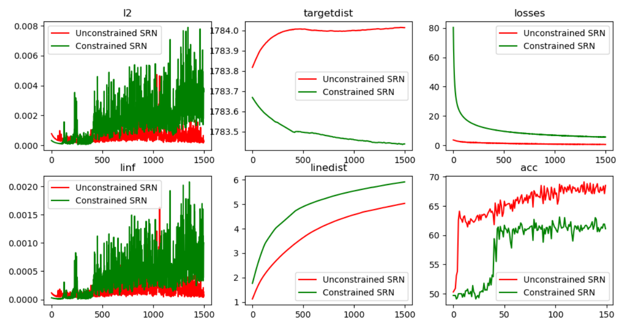


Figure 5: Test on 04.03.TLTT.4.2.3 from ML-RegTest. Parameters : learning rates = 0.001 ; Number of epoch = 150 ; Batch size = 100 ; optimizer = Adam ; loss = BCE & Inverse of the parameters' norm (with a coefficient $\lambda = 1$).

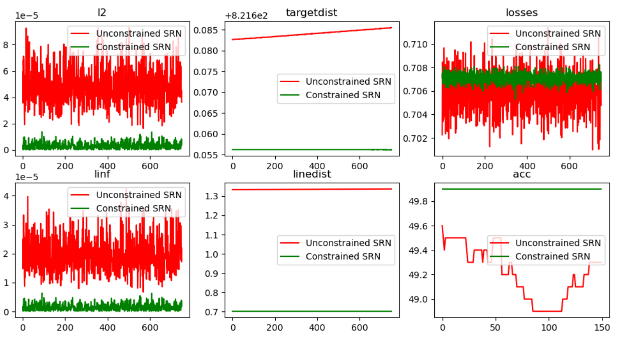


Figure 6: Test on 04.04.Zp.2.1.0 from ML-RegTest. Parameters : learning rates = 0.005 ; Number of epoch = 150 ; Batch size = 200 ; optimizer = SGD ; loss = nth Root Loss (with n=2).

Until now, no result have been conclusive. With all the tests done, we cannot extract a clear behavior of the different metrics. From the examples above, we can understand the irregularity of these metrics.

However, we can see some similarities between the tests.
Firstly, the accuracy of the constrained SRN is nearly always worst than the not constrained SRN. It should be because we do not reach the region of big parameters, so we will never be saturated and thus the constrained may not be relevant at this step.
Then, we do not observe the vanishing gradient phenomenon in our tests. As we can see with the ℓ_2 and ℓ_∞ norm, neither the constrained SRN nor the unconstrained one have a norm tending towards 0. This is another hint that we do not go in the desire direction.
Next, the distance to target is generally better for the constrained SRN than for the unconstrained one, but if we look at the values, both are just faraway from the target. In addition, if we look at the distance to the target line (linedist), we never get to the good directions. It should be because we do not learn big parameters yet.
Finally, sometimes the constrained SRN does not learn anything. It can happen the not constrained SRN begin to have a good accuracy (going higher than 80%), during the constrained RNN staying at 50%. It may mean the constrained are too strong, and we should implement it in a softer way.

Even if all these results are not good, we have other trails to explore. We detailed them in the next section.

10 Discussion

All along this paper, we discussed two major axes: resolving the vanishing gradient problem, and learn SSRN with heuristic processes. There is a lot of tracks we did not explore yet, and we have some issues in our own experiments.

We can make some remarks about the attempts we have done for the vanishing gradient phenomenon.
We do not use HFO until know, but it could lead in learning the parameters in the direction of the target, even if we will not reach the desire region of parameters[15].
For K-FAC, it should have been really hard to test it because we have not found a valid implementation for SRNs. However, we have not test our hypothesis to check if the method is indeed not efficient for our case.
They must have other methods to use the curvature information in the optimization process. We may find a more suitable one for our specific scenario.

For the heuristic part, there is still some issues we have not fix yet.
First, to make the two SRN really comparable, it seems important to have the same initialisation for the parameters. For now on, it is not the case.
We also already know this method cannot standalone, because it does not result in an unbounded gradient. Therefore, we know we cannot directly learn a SSRN and we will have to add more steps to finally get the desire saturation.
Also, we have not prove the uniqueness of the relation between DFA and SSRN, and it is not unique: we can shuffle the name of the states of a DFA without changing the DFA itself. These permutations result in rearranged matrices of the SSRN. Thus, we have at most an all class of SSRN that can simulate this DFA. That means the distance to target is not that relevant, neither the distance to the line target.

[15]Since the gradient step is still bounded.

The number of epochs may be not relevant too. Our experiments were done on 150 epochs per SRN. The loss was still really high at the end of the training in some cases. We may learn other directions if we do more steps. The loss could also contained more constraints. Instead of force directly the SRN to have the good shape, we should penalize the network in the loss if it does not look like what we expected. For example, instead of specify we want equal columns, we can add to the loss the difference between a group, such that it encourage it to be close to 0.
Finally, we have trained on DFA without regarding the DFA's complexity. We should train the networks per level of complexity first, to potentially find different behavior regarding the complexity classes.

We will take all these points into account in a future work.

11 Conclusion

Finally, we do not succeed in learning a SSRN for now. We do not overcome the vanishing gradient problem, as the methods found to do so are not suitable for our project. The heuristic method does not lead to any advancement for now too, but we will try other approaches as discussed in section 10. We have to explore the complexity of the DFA as a main axis, to see if learning a DFA of less complexity is easier. We also want to make more theory on the equivalence between a specific DFA and a class of SSRN, such that we can have a clear idea of the target space we are looking for during training.

This project was interesting and motivating. A lot of tracks have still to be explored, and we hope to find some conclusive results until the end of the internship.

References

S.-I. Amari. Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276, 1998.

Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 1994.

J. L. Elman. Finding structure in time. *Cognitive science*, 1990.

Y. Gao and D. Glowacka. Deep gate recurrent neural network. In *Asian conference on machine learning*, pages 350–365. PMLR, 2016.

T. Heskes. On "natural" learning and pruning in multilayered perceptrons. *Neural Computation*, 12(4):881–901, 2000.

S. Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9 (8):1735–1780, 1997.

J. Martens and R. Grosse. Optimizing Neural Networks with Kronecker-factored Approximate Curvature. *23th ICML*, 2015.

J. Martens and I. Sutskever. Learning recurrent neural networks with hessian-free optimization. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 1033–1040, 2011.

J. Martens, J. Ba, and M. Johnson. Kronecker-factored curvature approximations for recurrent neural networks. In *International Conference on Learning Representations*, 2018.

J. Martens et al. Deep learning via hessian-free optimization. In *Icml*, volume 27, pages 735–742, 2010.

W. Merrill. Sequential neural networks as automata. *In Proceedings of the Workshop on Deep Learning and Formal Languages: Building Bridges*, 2019.

V. Mitarchuk, C. Lacroce, R. Eyraud, R. Emonet, A. Habrard, and G. Rabusseau. Length Independent PAC-Bayes Bounds for Simple RNNs. *International Conference on Artificial Intelligence and Statistics*, 2023.

V. Mitarchuk, R. Eyraud, R. Emonet, and A. Habrard. On the limit of gradient descent for Simple Recurrent Neural Networks with finite precision. In *Journal of Machine Learning Research 1–50*, 2024.

S.-H. Noh. Analysis of Gradient Vanishing of RNNs and Performance Comparison. Information. *MDPI*, 2021. doi: 10.3390/info12110442.

B. A. Pearlmutter. Fast exact multiplication by the hessian. *Neural computation*, 6(1): 147–160, 1994.

S. Van der Poel, D. Lambert, K. Kostyszyn, T. Gao, R. Verma, D. Andersen, J. Chau, E. Peterson, C. S. Clair, P. Fodor, et al. Mlregtest: A benchmark for the machine learning of regular languages. *arXiv preprint arXiv:2304.07687*, 2023.

C. F. Van Loan. The ubiquitous kronecker product. *Journal of computational and applied mathematics*, 123(1-2):85–100, 2000.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

G. Zhang, J. Martens, and R. B. Grosse. Fast convergence of natural gradient descent for over-parameterized neural networks. *Advances in Neural Information Processing Systems*, 32, 2019.

Appendix A Abbreviations and notations

BCE : Binary Cross Entropy. A common loss function.

CG : Conjugate Gradient. An optimization method very efficient on quadratic function that go in the conjugate direction of the gradients at each step to guarantee a fast convergence (see section 5).

DFA : Deterministic Finite states Automaton. Automaton with a finite number of states and a fully predictable behavior.

HFO : Hessian-Free Optimization. A neural network optimizer algorithm which take into account the Hessian through a second order approximation of the loss (see Section 5).

K-FAC : Kronecker-factored Approximate Curvature. An approximation method of the Fisher Matrix to obtain some curvature information. We often use it to refer to the K-FAC optimizer, a neural network optimizer algorithm which use the K-FAC to compute a NGD (see Section 6).

NGD : Natural Gradient Descent. A neural network optimizer algorithm that use the Fisher matrix to add curvature information to the optimization steps (see section 6).

PSD : Positive Semi-Definite. A matrix M is PSD if, for all vector X, $X^T M X \geq 0$.

RNN : Recurrent Neural Network. A type of network dealing with sequences in a recursive way.

SRN : Simple Recurrent Network. The first form introduced of a Recurrent Neural Network. We denote a SRN by $\mathcal{R}_P$, with P the parameters of the network (see definition 1).

SSRN : (β-)Saturated Simple Recurrent Network. Specific type of SRN which used sigmoidal activation function and tend toward a binary behavior at its extrema (see Mitarchuk et al. [2023]).

$[\![a, b]\!]$: the set of integers between a and b, boundaries included.

$||a||$ and $||A||$: with a a vector, $||a||$ is the ℓ_2 norm of a. With A a matrix, $||A||$ is the spectral norm of A.

$|a|$ or $|A|$: the number of dimensions of the vector a, the number of elements of the set A.

Please note, for simplicity and consistency all along this paper, we use the Python convention for indexing matrix and thus start by index 0.

Appendix B Proof of property 1

If $n = 1$, both basis are the one-element vector (1), so the inverse is trivial.
If $n = 2$, the matrix of the hidden vector of tanh is:

$$\begin{pmatrix} 1 & -1 \\ -1 & 1 \end{pmatrix}$$

and this is not a base, as the first column is (-1) times the second one. There is no solution to the inverse.
If $n \geq 3$, we check the product $\mathcal{B}_{\tanh}\mathcal{B}_{\tanh}^{-1} = Id_n$.

$\forall (i,j) \in [\![1,n]\!]^2, i \neq j$:

$$\begin{aligned}
\left(\mathcal{B}_{\text{tanh}}\mathcal{B}_{\text{tanh}}^{-1}\right)_{i,j} &= \frac{1}{2}\left(-1 \times (1 - \frac{1}{n-2}) - 1 \times \frac{1}{n-2} + \sum_{k=3}^{n} -1 \times (-\frac{1}{n-2})\right) \\
&= \frac{1}{2}\left(-1 + \frac{1}{n-2} - \frac{1}{n-2} + \sum_{k=3}^{n} \frac{1}{n-2}\right) \\
&= \frac{1}{2}\left(-1 + (n-3+1)\frac{1}{n-2}\right) \\
&= \frac{1}{2}(-1+1) \\
&= 0
\end{aligned}$$

$\forall i \in [\![1,n]\!]$:

$$\begin{aligned}
\left(\mathcal{B}_{\text{tanh}}\mathcal{B}_{\text{tanh}}^{-1}\right)_{i,i} &= \frac{1}{2}\left(1 \times (1 - \frac{1}{n-2}) + \sum_{k=2}^{n} -1 \times (-\frac{1}{n-2})\right) \\
&= \frac{1}{2}\left(1 - \frac{1}{n-2} + \sum_{k=2}^{n} \frac{1}{n-2}\right) \\
&= \frac{1}{2}\left(1 + \sum_{k=3}^{n} \frac{1}{n-2}\right) \\
&= \frac{1}{2}(1+1) \\
&= 1
\end{aligned}$$

Therefore, we have $\mathcal{B}_{\text{tanh}}\mathcal{B}_{\text{tanh}}^{-1} = Id_n$
Since both matrices are symmetric, we have:

$$\begin{aligned}
&\mathcal{B}_{\text{tanh}}\mathcal{B}_{\text{tanh}}^{-1} = Id_n \\
\Leftrightarrow &\left(\mathcal{B}_{\text{tanh}}\mathcal{B}_{\text{tanh}}^{-1}\right)^T = Id_n^T \\
\Leftrightarrow &\mathcal{B}_{\text{tanh}}^{-T}\mathcal{B}_{\text{tanh}}^{T} = Id_n^T \\
\Leftrightarrow &\mathcal{B}_{\text{tanh}}^{-1}\mathcal{B}_{\text{tanh}} = Id_n
\end{aligned}$$

Therefore, the inverse is respected from left and right, and is the global inverse.