LICENCE D'INFORMATIQUE
École Normale Supérieure de Lyon
Université Claude Bernard Lyon I

Stage 2015–2016
STÉRIN Tristan
L3 informatique

Réseaux de neurones récurrents et mémoire : application à la musique

Tristan Stérin

tristan.sterin@ens-lyon.fr

Résumé : *Les réseaux de neurones récurrents sont des modèles aptes à apprendre et à générer des séquences temporelles. Ces réseaux se déclinent en plusieurs variantes dont les deux principales sont Vanilla et LSTM. À travers un exemple concret d'inférence grammaticale on constate la faiblesse des Vanilla à exploiter des dépendances temporelles longues. Sur la base de ces résultats expérimentaux on remet en cause l'importance de la raison la plus souvent invoquée dans la littérature pour expliquer cette faiblesse. On propose une autre explication que l'on conjuge avec l'introduction d'une mesure de la capacité de mémoire d'un modèle récurrent. On confronte cette mesure à la théorie des Echo States Networks qui aborde ces questions de mémoire différemment. Forts de ces expériences on applique les techniques décrites à la génération de musique à travers les chorals de Bach.*

Mots clefs : *RNNs, Mémoire, Echo States Networks, Musique*

Stage encadré par :
Nicolas FARRUGIA
nicolas.farrugia@telecom-bretagne.eu
Télécom Bretagne
Technopôle Brest-Iroise - CS 83818 - 29238 Brest Cedex 3
http://www.telecom-bretagne.eu/

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Lab-STICC

06 Septembre 2016

Remerciements

Je tiens à remercier très sincèrement mon tuteur Nicolas Farrugia et son collègue Vincent Gripon qui m'ont accueilli avec beaucoup d'attention dans leur équipe. Très présents et toujours soucieux de l'avancement de mon travail, ils m'ont beaucoup apporté. Je tiens notamment à les remercier pour leur invitation aux conférences NeuroSTIC qui se sont déroulées à l'Inria Grenoble pendant mon stage, j'ai beaucoup appris. J'ai aussi pu assister à beaucoup de présentations de travaux, thèses, séances de réflexions interdisciplinaires très enrichissantes. Je souhaite aussi remercier tous les doctorants, post-doctorants, ingénieurs, chercheurs du laboratoire dont Bastien, Eliott, Ala, Jean-Charles, Philippe, Max, Carlos, Deok-Hee, Olivier, François, Claude Berrou, qui n'ont jamais hésité à m'accorder de leur temps ou à échanger avec moi ainsi que mes camarades stagiaires Jules, Francis, Ghouti, Hippolyte, Emma et Martin. Je remercie enfin Télécom Bretagne pour m'avoir permis de découvrir le monde de la recherche à travers un stage qui m'a passionné.

Table des matières

Introduction 1

1 Réseaux de neurones récurrents 1
- 1.1 Réseaux de neurones . . . 1
- 1.2 Réseaux de neurones récurrents . . . 3
  - 1.2.1 Vanilla RNNs . . . 3
  - 1.2.2 LSTM, GRU . . . 4
- 1.3 Apprentissage de séquences et modèle génératif . . . 6

2 RNNs et mémoire à court terme 7
- 2.1 La grammaire de Reber embarquée . . . 7
- 2.2 Séquences indistinguables . . . 10
- 2.3 L'approche Echo State Networks . . . 12

3 Application à la musique 15
- 3.1 Les chorals de Bach . . . 15
- 3.2 Méthode . . . 16
- 3.3 Résultats . . . 17

Conclusion 18

Références 19

Appendices 20

Annexe A Principaux éléments de code 20
- A.1 Implémentation des RNNs Vanilla . . . 20
- A.2 Implémentation des LSTMs (GRU) . . . 22
- A.3 Boucle d'apprentissage . . . 25

Annexe B Contexte institutionnel 26

Introduction

Les réseaux de neurones récurrents – RNNs – ont été introduit en *machine learning* afin de pouvoir traiter des données séquentielles. Pour modéliser ces données et les éventuelles dépendances temporelles qui y apparaissent les RNNs conservent un état caché qui résume leur historique. Au cours de ce stage nous nous sommes demandés à quel point ils tirent profit de cet historique pour orienter leurs décisions. Au travers d'expériences nous avons réalisé que le premier modèle de RNNs inventé souffre d'une très faible mémoire court-terme. Les articles qui abordent cette question font souvent appels à des arguments de calcul numérique lors de la phase d'apprentissage pour expliquer ce phénomène. Nous montrons que ces arguments ne paraissent pas être prédominants dans notre contexte et en proposons d'autres en lien avec la théorie dite des *Echo States Networks*. Enfin, à travers les chorals de Bach, nous nous intéressons à la musique représentant un excellent exemple de données séquentielles hautement dépendantes de leur contexte.

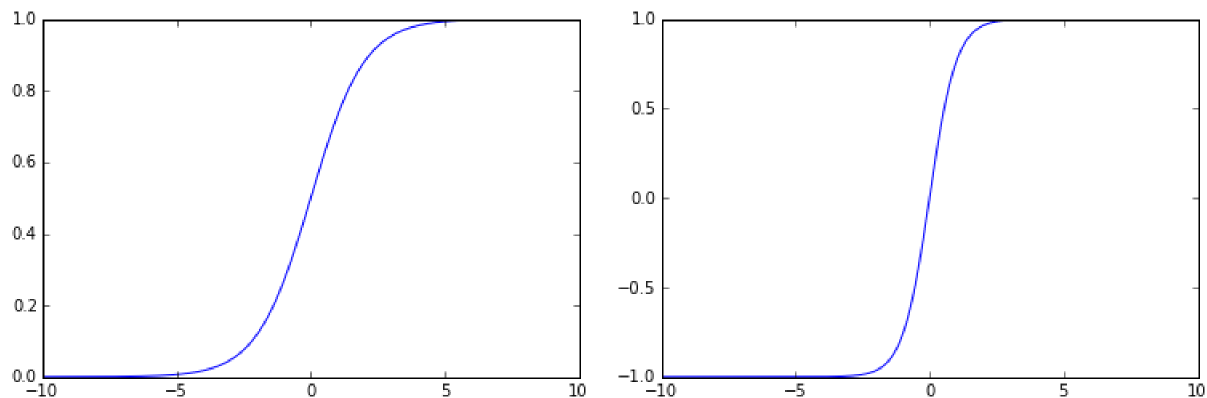


Figure 1 : Exemples de sigmoïde : logistique et tanh

1 Réseaux de neurones récurrents

Les réseaux de neurones récurrents sont une extension des réseaux de neurones traditionnels, nous les réintroduisons donc au préalable.

1.1 Réseaux de neurones

Les réseaux de neurones ne sont pas un modèle nouveau. En effet ils ont été introduits dès les années 50 – le perceptron est présenté en 1958, [1]. Cependant ils sont tombés en désuétude dans les années 70 car trop coûteux en calculs. Ils redeviennent un sujet d'intérêt dans les années 90 et connaissent actuellement leur heure de gloire grâce aux architectures profondes – *deep learing* – et à la puissance de calcul proposée par les GPU.

Définition 1.1 (Sigmoïde)**.**
Une sigmoïde est une fonction de $\mathbb{R}$ dans $\mathbb{R}$:
— Continue
— Monotone
— Admettant des limites finies en $\pm\infty$

Définition 1.2 (Neurone artificiel)**.**
Soit $n \in \mathbb{N}$ et $\boldsymbol{\omega} \in \mathbb{R}^n$ et $a : \mathbb{R} \to \mathbb{R}$ une fonction, le neurone artificiel paramétrisé par $\boldsymbol{\omega}$ avec fonction d'activation a est la fonction $f_{\boldsymbol{\omega}} : \mathbb{R}^n \to \mathbb{R}$ définie par :

$$f_{\boldsymbol{\omega}}(\boldsymbol{x}) = a(\boldsymbol{x} \bullet \boldsymbol{\omega})$$

Avec $\bullet$ le produit scalaire usuel.
On appelle le paramètre $\boldsymbol{\omega} = (\omega_1, \ldots, \omega_n)$ le vecteur de **poids**.
On appelle l'argument $\boldsymbol{x} \bullet \boldsymbol{\omega}$ **activation** du neurone.
Ici on ne considérera que des fonctions d'activation **sigmoïdes** ou **linéaires**.

Le plus souvent on utilise comme sigmoïde la fonction logistique $\sigma(x) = \frac{1}{1+e^{-x}}$ ou bien la tangente hyperbolique voir la figure 1. Une manière plus visuelle et utile pour la suite de représenter un neurone artificiel est donnée par le schéma suivant :

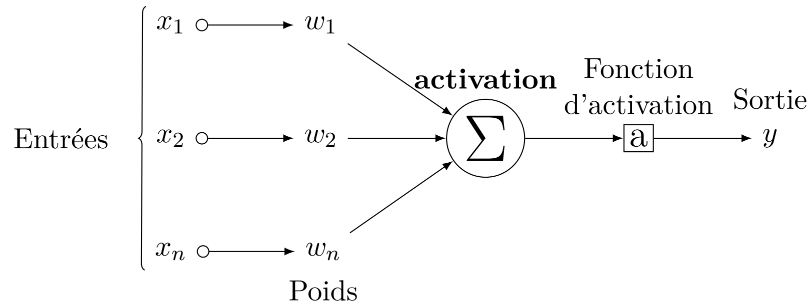


Figure 2 : Schéma d'un neurone artificiel

Définition 1.3 (Couche de neurones).
Une couche de neurones est un ensemble de neurones artificiels ayant tous le même nombre d'entrées. On peut ainsi regrouper leur paramètre $\boldsymbol{\omega}$ par ligne et former une matrice de connexion W.

Définition 1.4 (Réseau de neurones).
Un réseau de neurones est une succession de couches de neurones telles que le nombre de neurones sur chaque couche est égal au nombre d'entrées des neurones de la couche suivante.

— La première couche contient des neurones factices qui transmettent les entrées fournies au réseau, c'est la **couche d'entrée**.

— La dernière couche est appelée **couche de sortie** et fournit le résultat du réseau.

— Les autres couches sont appelées **couches cachées**.

Ici on considère une architecture particulière de réseau de neurones :

Définition 1.5 (Perceptron). Un perceptron, [1], est un réseau de neurones aux propriétés suivante :
— Une couche cachée dont tous les neurones possèdent la même activation sigmoïde
— Une couche de sortie linéaire
Il est donc caractérisé par :
— $n \in \mathbb{N}$ son nombre d'entrées
— $k \in \mathbb{N}$ son nombre de neurones cachés et leur sigmoïde
— $p \in \mathbb{N}$ son nombre de sortie
— $W \in \mathcal{M}_{k,n}$ qui regroupe les paramètres de la couche cachée
— $O \in \mathcal{M}_{p,k}$ qui regroupe les paramètres de la couche de sortie
Pour $\boldsymbol{x}$ en entrée il opère la transformation :

$$R(\boldsymbol{x}) = Os(W\boldsymbol{x})$$

Avec $s : \mathbb{R} \to \mathbb{R}$ une sigmoïde quelconque étendue composante par composante. Les matrices O et W sont les paramètres du réseau, ses **poids**.

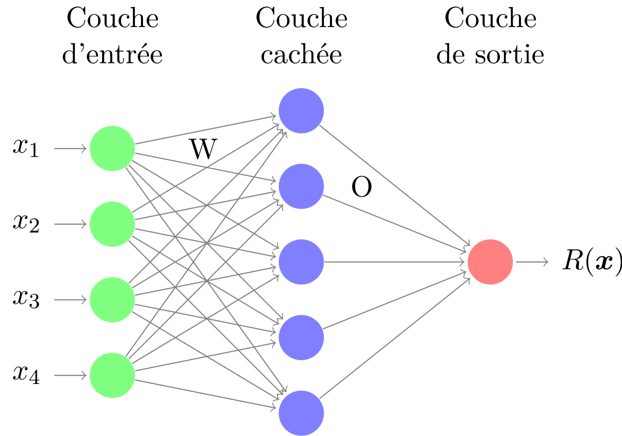


Figure 3 : Perceptron avec $n = 4$, $k = 5$ et $p = 1$

Les perceptrons peuvent se visualiser à l'aide d'un graphe du type de la figure 3.

Les perceptrons forment une classe de fonctions intéressantes car il s'agit d'approximateurs universaux [2] : étant donnés suffisamment de neurones et des paramètres O et W convenables il est possible d'interpoler à précision arbitraire toute fonction Borel-mesurable. De plus grâce aux non-linéarités introduites par σ ils sont parcimonieux [3], [4] : ils requièrent peu de paramètres par rapport à d'autres méthodes.
Si on connaît une fonction T au travers d'un ensemble d'exemples $B = \{(\boldsymbol{x}, T(\boldsymbol{x}))\}$ on peut ajuster les paramètres d'un réseau R pour l'approximer. Pour cela on minimise la fonction de coût :

$$J(O, W) = \frac{1}{|B|} \sum_{\boldsymbol{x} \in B} ||R_{O,W}(\boldsymbol{x}) - T(\boldsymbol{x})||^2$$

Le réseau étant entièrement différentiable on peut effectuer cette minimisation à l'aide d'algorithmes de descente de gradient, le plus utilisé étant la descente de gradient stochastique avec rétropropagation [5].

1.2 Réseaux de neurones récurrents

Les réseaux récurrents – RNNs dans la suite – permettent de traiter des données séquentielles, une suite d'entrées $\boldsymbol{x}_0 \dots \boldsymbol{x}_m$. En effet au temps t ils calculent leur sortie en fonction de l'entrée $\boldsymbol{x}_t$ mais aussi de l'état de la couche cachée au temps précédent. Ainsi ils font évoluer un état interne qui fait office de mémoire à court terme – *short-term memory* – et qui permet de prendre en compte les dépendances temporelles que manifestent les entrées.

1.2.1 Vanilla RNNs

Les RNNs les plus simples se décrivent de la manière suivante :

Définition 1.6 (Réseaux de neurones récurrents simples, **Vanilla RNNs**).
Soient :
- $n, k, p \in \mathbb{N}$
- $\boldsymbol{x}_0 \dots \boldsymbol{x}_m$ avec $\boldsymbol{x}_t \in \mathbb{R}^n$ une suite d'entrées
- $W \in \mathcal{M}_{k,n}$
- $W_h \in \mathcal{M}_{k,k}$
- $O \in \mathcal{M}_{p,k}$
- $\boldsymbol{h}_{-1} \in \mathbb{R}^k$

Alors la dynamique du RNN associé est décrite par :

$$\boldsymbol{h}_t = \sigma(W\boldsymbol{x}_t + W_h\boldsymbol{h}_{t-1})$$
$$\boldsymbol{y}_t = O\boldsymbol{h}_t$$

Avec $\boldsymbol{h}_t \in \mathbb{R}^k$ l'état de la couche cachée et $\boldsymbol{y}_t \in \mathbb{R}^p$ l'état de la couche de sortie.
En pratique on prend souvent $\boldsymbol{h}_{-1} = \boldsymbol{0}$.
Dans la suite on utilisera indifféremment RNN pour *Vanilla* RNN.

On peut de même visualiser ces RNNs sous forme de graphe :

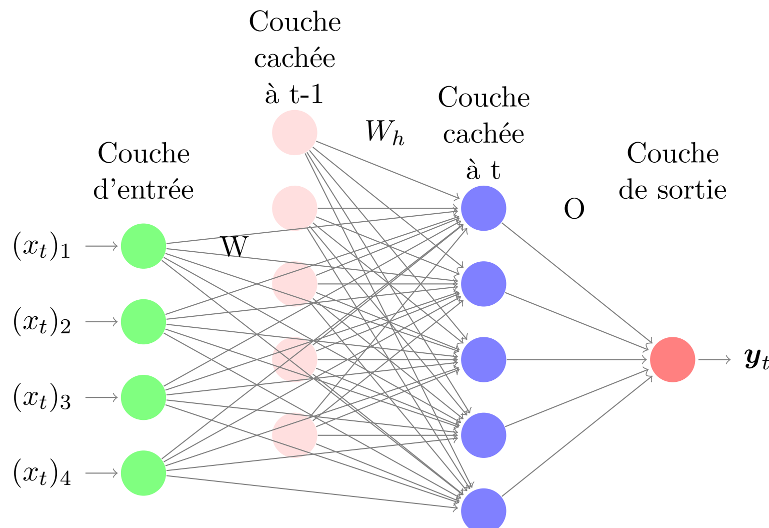


Figure 4 : RNN avec $n = 4$, $k = 5$ et $p = 1$

Théoriquement ces RNNs forment un modèle très puissant : ils sont Turing Complet [6], on peut implémenter n'importe quel algorithme avec cette architecture. L'auteur de [6] propose une machine de Turing universelle à partir d'un RNN à 886 neurones cachés.
Cependant en pratique il est difficile de les entraîner et d'atteindre leur puissance théorique. L'une des raisons est présentée dans [7] : la descente de gradient adaptée aux RNNs, *Backpropagation Through Time* – BTT ou BPTT [8]–, rencontre des difficultés de calcul numérique. Les gradients explosent ou s'écrasent numériquement – *Vanishing and exploding gradient.*
Ainsi ces modèles ne prennent en compte que des dépendances très court terme comme cela sera illustré dans la deuxième partie du rapport. C'est pour remédier à ces problèmes qu'a été introduit en 1997 le modèle *Long Short-Term Memory* dans [9] – LSTM dans la suite.

1.2.2 LSTM, GRU

L'architecture d'un module LSTM est plus complexe à présenter. Dans ce rapport nous présenterons simplement un schéma d'ensemble ainsi que les équations qui gouvernent l'ensemble et une description rapide. D'avantage de détails seront exposés lors de la présentation orale.

2. http://colah.github.io/posts/2015-08-Understanding-LSTMs/

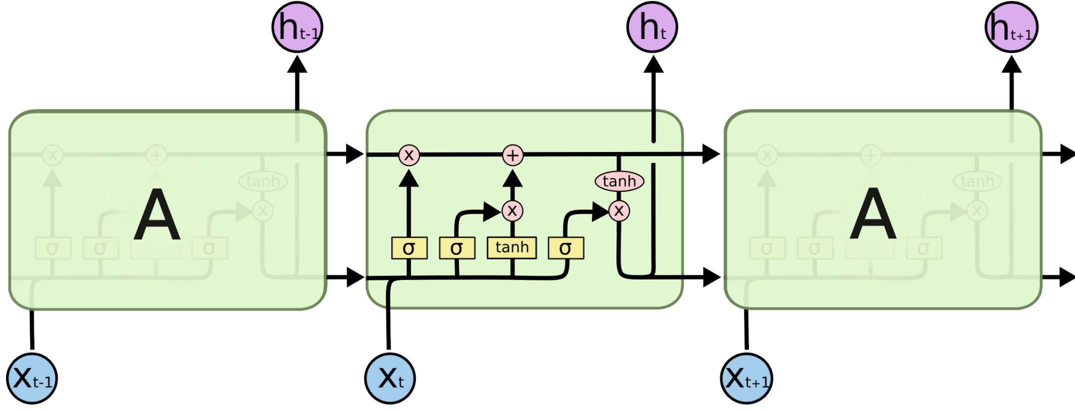


FIGURE 5 : Un module LSTM déplié dans le temps[2]

Les équations de la dynamique de ce modèle sont les suivantes :

$$\begin{aligned}
\boldsymbol{i} &= \sigma(U^i \boldsymbol{x}_t + W^i \boldsymbol{h}_{t-1}) \\
\boldsymbol{f} &= \sigma(U^f \boldsymbol{x}_t + W^f \boldsymbol{h}_{t-1}) \\
\boldsymbol{o} &= \sigma(U^o \boldsymbol{x}_t + W^o \boldsymbol{h}_{t-1}) \\
\tilde{\boldsymbol{c}} &= tanh(U^{\tilde{c}} \boldsymbol{x}_t + W^{\tilde{c}} \boldsymbol{h}_{t-1}) \\
\boldsymbol{c}_t &= \boldsymbol{c}_{t-1} \circ \boldsymbol{f} + \tilde{\boldsymbol{c}} \circ \boldsymbol{i} \\
\boldsymbol{h}_t &= tanh(\boldsymbol{c}_t) \circ \boldsymbol{o} \\
\boldsymbol{y}_t &= O \boldsymbol{h}_t
\end{aligned}$$

Avec $\boldsymbol{i}, \boldsymbol{f}, \boldsymbol{o}, \tilde{\boldsymbol{c}}, \boldsymbol{c}_t, \boldsymbol{h}_t \in \mathbb{R}^k$ et $\boldsymbol{y}_t \in \mathbb{R}^p$. Ici σ est spécifiquement la fonction logistique et $\circ$ représente la multiplication élément par élément.
Le vecteur $\boldsymbol{c}_t$ est la mémoire de la cellule LSTM et $\boldsymbol{y}_t$ la sortie. L'idée est de contrôler la mémoire à l'image du modèle RAM. On gère la lecture et l'écriture mémoire à l'aide des vecteurs $\boldsymbol{f}$, $\boldsymbol{i}$ et $\boldsymbol{o}$. Ce modèle résout les problèmes de calcul numérique évoqués plus haut. Il possède également la puissance de pouvoir apprendre la manière dont il gère ses accès mémoires. En effet $U^i, W^i, U^f, W^f, U^o, W^o$ font partie de ses paramètres. C'est autour de cette idée qu'ont été réalisées les *Neural Turing Machines* qui apprennent des algorithmes à partir d'états mémoire rencontrés lors d'exécutions [10], [11].
Avec l'avènement des architectures profondes LSTM est très utilisé aujourd'hui dans le cadre de l'apprentissage de séquences et a démontré une grande puissance expérimentale, comme recensé dans [12]. Cependant cette architecture est sujette à beaucoup de modifications, elle comporte presque autant de variations que de papiers qui l'utilise.
Récemment – 2014 – une version simplifiée de LSTM nommée GRU (*Gated Recurrent Unit*) a été introduite dans [13]. Elle a l'avantage d'être moins lourde en calculs car possédant moins de paramètres et d'équations :

$$\begin{aligned}
\boldsymbol{z} &= \sigma(U^z \boldsymbol{x}_t + W^z \boldsymbol{h}_{t-1}) \\
\boldsymbol{r} &= \sigma(U^r \boldsymbol{x}_t + W^r \boldsymbol{h}_{t-1}) \\
\tilde{\boldsymbol{h}} &= tanh(U^{\tilde{h}} \boldsymbol{x}_t + W^{\tilde{h}} (\boldsymbol{h}_{t-1} \circ \boldsymbol{r}) \\
\boldsymbol{h}_t &= (1 - \boldsymbol{z}) \circ \tilde{\boldsymbol{h}} + \boldsymbol{z} \circ \boldsymbol{h}_{t-1} \\
\boldsymbol{y}_t &= O \boldsymbol{h}_t
\end{aligned}$$

Les auteurs de [14] démontrent qu'elle a des performances équivalentes au modèle LSTM de base. Dans la suite on confondra les deux architectures mais les calculs ont été réalisés avec GRU.

1.3 Apprentissage de séquences et modèle génératif

On se place dans le contexte d'apprentissage de séquences sur un alphabet fini $\mathcal{A}$. On se munit de deux symboles spéciaux : $\odot$ et $\otimes$, début et fin de séquences. Les exemples de la base de données sont des suites finies $\boldsymbol{x}_0 \dots \boldsymbol{x}_m$ avec $\boldsymbol{x}_t \in \mathcal{A}$ et $\boldsymbol{x}_0 = \odot$, $\boldsymbol{x}_m = \otimes$. La traduction vectorielle des éléments de $\mathcal{A}$ peut, par exemple, consister à prendre un vecteur de base par lettre, ce qui donne selon les notations précédentes $n = |\mathcal{A}| + 2$.
On réalise alors une tâche d'apprentissage supervisé avec les couples $(\boldsymbol{x}_t, \boldsymbol{x}_{t+1})$, l'état interne du RNN gardant trace de l'historique de la séquence.
Par exemple, prenons un alphabet binaire **0**, **1**. On aura $n = 4$ avec :

$$\odot = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \quad \mathbf{0} = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \quad \mathbf{1} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \quad \otimes = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

On peut échantillonner le langage $(\mathbf{0}|\mathbf{1})\mathbf{01}^*$ et avoir des séquences du type :

$$\begin{array}{r} \odot\mathbf{0011}\otimes \\ \vdots \\ \odot\mathbf{101111}\otimes \end{array}$$

Comme, étant donné un symbole, on attend que le RNN nous prédise le suivant on a $p = n = 4$. Le paramètre k est celui que l'on peut ajuster pour obtenir les meilleurs résultats. En fin d'apprentissage on attend du réseau le comportement suivant :

- Lorsque qu'on présente en entrée le symbole $\odot$, les neurones les plus activés de la couche de sortie – *i.e* avec la valeur la plus élevée – doivent être le deuxième et le troisième qui correspondent à **0** et **1**.
- Lorsqu'on présente $\odot$ puis **0** ou **1**, un seul neurone de sortie doit être majoritairement activé, le deuxième qui correspond à **0**.
- Lorsqu'on présente $\odot$, **0** ou **1** , **0** le seul neurone activé doit être le troisième pour **1**.
- Enfin, au terme de l'opération, seuls doivent être activés les deux derniers pour **1** et $\otimes$.

Pour obtenir un modèle génératif on peut transformer systématiquement la couche de sortie du réseau en distribution de probabilité et, partant du symbole $\odot$ choisir le suivant en fonction de cette distribution et l'insérer en entrée. On répète le processus jusqu'à l'obtention du symbole $\otimes$.
La transformation en distribution de probabilité peut s'accomplir à l'aide d'une opération de softmax. Si on appelle $(\boldsymbol{y}_t)_i$ la ième sortie on considère la probabilité :

$$(\tilde{\boldsymbol{y}}_t)_i = \frac{e^{-\beta(\boldsymbol{y}_t)_i}}{\sum_{l=1}^{p} e^{-\beta(\boldsymbol{y}_t)_l}}$$

Avec β le facteur de température qui contrôle l'allure de la distribution.
En fait de tels modèles calculent :

$$Pr(\boldsymbol{x}_{t+1}|\boldsymbol{x}_0 \dots \boldsymbol{x}_t)$$

2 RNNs et mémoire à court terme

Dans le vocabulaire des RNNs, on oppose mémoire à court terme et mémoire à long terme. La première est celle qui évolue au court du traitement, par exemple $\boldsymbol{h}_t$ pour un *Vanilla*. La seconde est celle issue de l'apprentissage, ancrée dans les poids du réseau. C'est la mémoire à court terme, qui détient les éléments de contexte, que l'on étudie ici. Nous présentons d'abord cette mémoire à court terme à travers un exemple confrontant RNNs et LSTM pour ensuite essayer de comprendre pourquoi LSTM est plus puissant.

2.1 La grammaire de Reber embarquée

Dans l'article qui introduit LSTM, [9], les auteurs présentent un ensemble de problèmes au travers desquels les *Vanilla* sont très médiocres par rapport aux LSTM. L'un d'eux est celui de la grammaire de Reber embarquée. On introduit d'abord la grammaire de Reber qui est l'automate suivant :

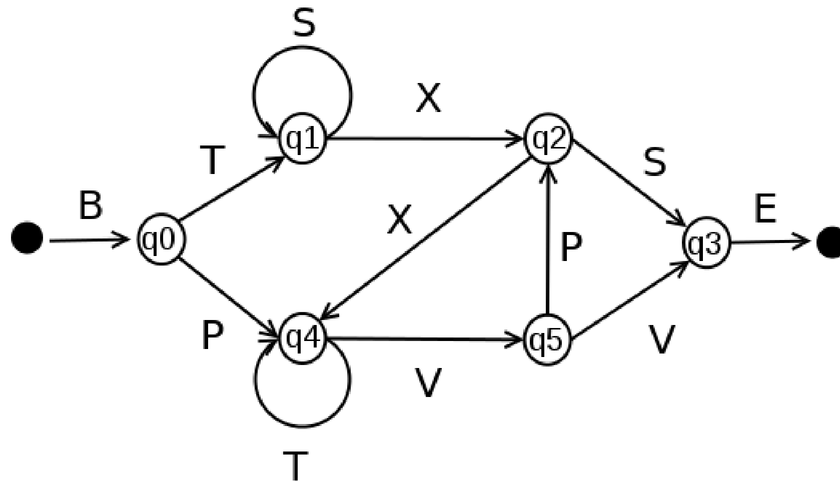


Figure 6 : Grammaire de Reber

Le but étant, après présentation d'un ensemble de séquences terminales d'avoir un RNN génératif qui ne produit que des séquences terminales. Tous les exemples générés dans la suite ont été codés à l'aide du module theano de python, les parties principales du code sont en annexe.
Une manière simple de tester un modèle entraîné est d'afficher sous forme de heatmap ses activations de sorties pour différentes séquences en entrée. En effet ici chaque neurone de sortie correspond à une lettre de l'alphabet. L'entraînement a été effectué avec 256 exemples de mots de Reber et un RNN simple à 5 neurones cachés.

Figure 7 : État de la couche de sortie d'un RNN Vanilla pour différentes séquences en entrée

On constate sur l'exemple de la figure 7 que le RNN simple suit fidèlement la structure de l'automate proposant toujours les bonnes transitions.
Il est aussi intéressant de représenter l'état de la couche cachée ($\boldsymbol{h}_t$), donc de la représentation interne du RNN, pour différentes séquences menant au même état. On peut ainsi constater si le RNN a inféré la structure de l'automate.

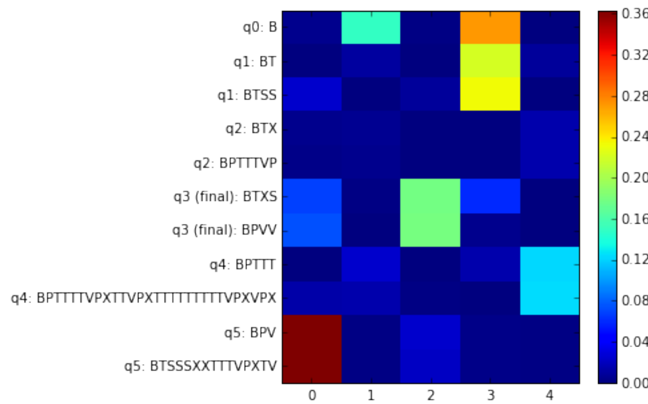


Figure 8 : État de la couche cachée pour différentes séquences menant 2 à 2 aux mêmes états

On constate avec la figure 8 que la structure de l'automate rejailli, notamment pour les états $q2$ et $q5$ qui sont identifiés malgré des manières très différentes d'y parvenir.
Ces résultats semblent proposer une méthode d'inférence de langages réguliers : on pourrait classifier les états de la couche cachée et de là, construire un automate.
Cependant la structure de cet automate est telle qu'il suffit de connaître les deux dernières lettres de la séquence pour déterminer l'état. Cette tâche requiert donc très peu de mémoire à court terme.
On considère maintenant la grammaire de Reber embarquée :

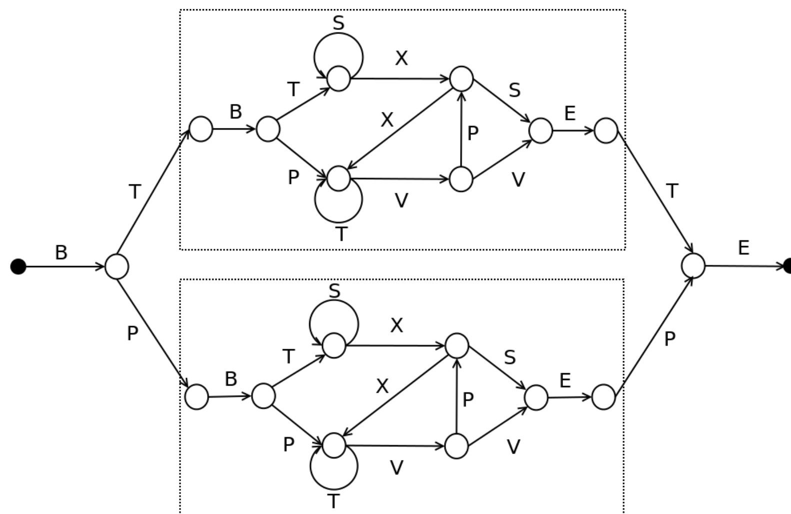


Figure 9 : Grammaire de Reber embarquée

Elle fournit des mots de la forme :
— BT - mot de Reber - TE
— BP - mot de Reber - PE
Ainsi il est nécessaire de prendre en compte des dépendances temporelles à beaucoup plus long terme : il faut retenir le deuxième caractère pour produire un mot correct.

Le RNN simple – 18 neurones cachés – échoue :

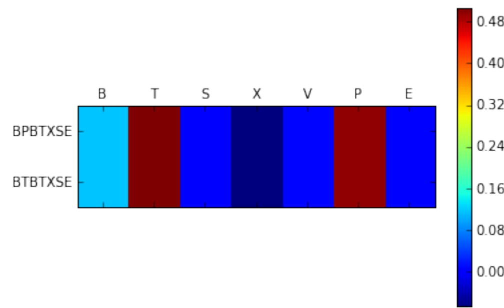


Figure 10 : Prédictions fausses d'un RNN face au problème de la grammaire embarquée

En effet celui-ci prédit indifféremment le P et le T quand on attend un P dans le premier cas et un T dans le deuxième.
Un LSTM avec un nombre de paramètres équivalent maîtrise bien cette tâche, après entraînement :

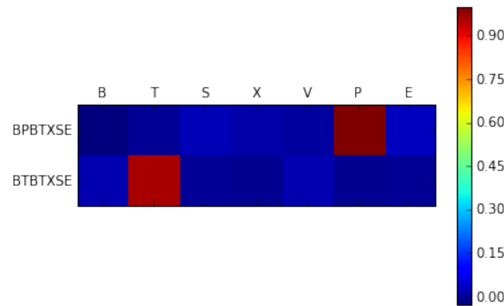
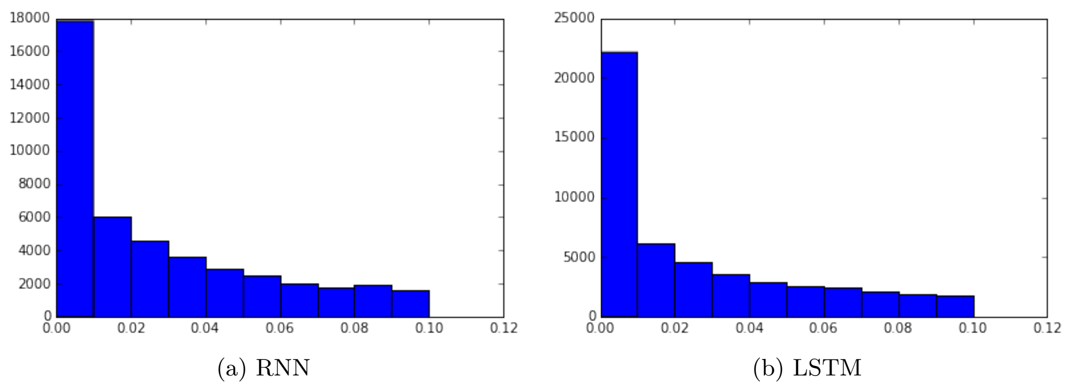


Figure 11 : Prédictions correctes d'un LSTM face au problème de la grammaire embarquée

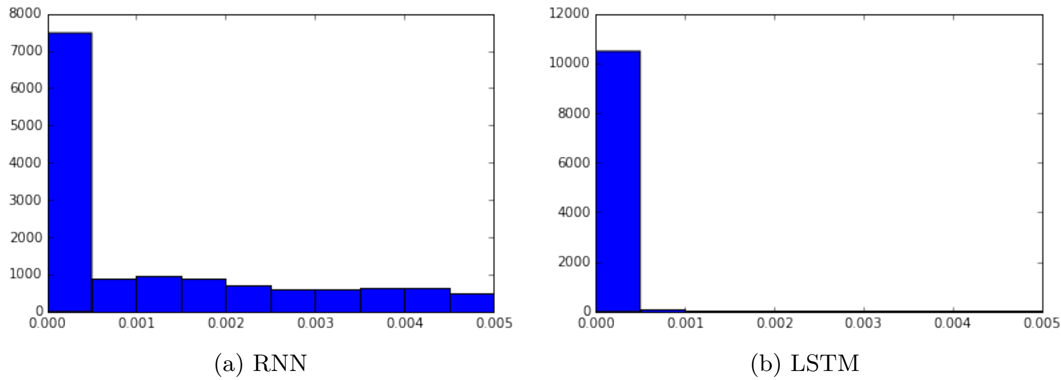
Dans la littérature la supériorité des LSTM face aux *Vanilla* est souvent expliquée par la résistance aux problèmes de gradients qui explosent ou qui s'écrasent. Donc par des phénomènes qui ont lieu pendant l'apprentissage et non intrinsèquement liés à la nature du modèle. Cependant ici si on regarde les répartitions des gradients des différents paramètres – en valeur absolue – pour les deux modèles RNN et LSTM elles sont équivalentes :



(a) RNN (b) LSTM

Figure 12 : Répartition des gradients pour les deux modèles

Ces profils ne manifestent pas d'explosion et, si on se concentre autour de zéro ils présentent le même taux d'écrasement :



(a) RNN (b) LSTM

Figure 13 : Répartition des gradients autour de 0 pour les deux modèles

Ici on peut donc difficilement expliquer le succès du modèle LSTM simplement par des arguments liés à l'apprentissage et au calcul du gradient.

2.2 Séquences indistinguables

Dans la suite on va s'intéresser aux RNNs en tant que systèmes dynamiques. Rappelons les équations :

$$\boldsymbol{h}_t = \sigma(W\boldsymbol{x}_t + W_h\boldsymbol{h}_{t-1}) \tag{1}$$
$$\boldsymbol{y}_t = O\boldsymbol{h}_t \tag{2}$$

Si on reconsidère l'exemple de la figure 10 on se rend compte qu'après passage dans le RNN, les deux séquences BPBTXSE et BTBTXSE produisent des $\boldsymbol{y}_t$ indistinguables. Numériquement en moyennant sur toutes les composantes on trouve une différence absolue de 0.0009. On pourrait penser que la couche de sortie est responsable il convient donc de calculer cette différence pour la couche cachée. On trouve une différence de 0.0002.
En clair le système dynamique ainsi paramétré est très peu sensible aux conditions initiales (ici seul le deuxième caractère diffère).
En répétant l'expérience avec des paramètres aléatoires on retrouve le même phénomène.
Or il paraît impossible de stocker de l'information à long terme si numériquement les différences sont gommées à ce point. Pour quantifier la capacité d'un RNN à exploiter une information antérieure de l pas on propose donc la mesure expérimentale suivante :

Définition 2.1 ((l,k)-distinguabilité)**.** On considère :
- $l, k \in \mathbb{N}$
- Un alphabet binaire **0**, **1**
- Deux mots u, v tels que :

$$\begin{aligned} &|u| = |v| = l \\ \forall i \neq 0 \quad &u_i = v_i = \mathbf{0} \\ &u_0 = \mathbf{1} \\ &v_0 = \mathbf{0} \end{aligned}$$

Pour un RNN paramétré par W et W_h (on ne consièdre plus les couches de sorties) on note $\boldsymbol{T}(u, W, W_h)$ l'état de la couche cachée après lecture du mot u, c'est-à-dire $\boldsymbol{h}_{|u|}$ avec $\boldsymbol{x}_t = u_t$ dans l'équation (1).

Alors la (l,k)-distinguabilité $D(l,k)$ est calculée selon le procédé suivant :
- Un grand nombre de fois (2000 dans les tests qui suivent), calculer $\boldsymbol{T}(u, W, W_h)$ et $\boldsymbol{T}(v, W, W_h)$ pour des choix aléatoires différents de W et W_h
- À chaque étape calculer la norme de la différence des deux vecteurs
- Prendre la moyenne de cette série, c'est $D(l,k)$

Donc plus $D(l,k)$ est élevée plus le réseau sait distinguer deux séquences au passé différent à $t-l$. On assimile **par hypothèse** la capacité de distinguabilité à la capacité de mémoire.

Voici le profil de $D(l,k)$ dans l'espace pour un RNN simple :

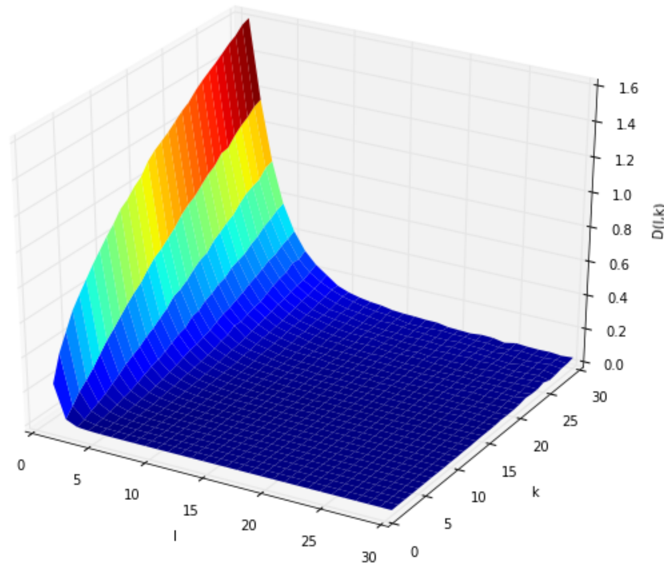


Figure 14 : D(l,k) pour un RNN simple

On constate d'abord, comme espéré, une augmentation de la capacité de mémoire lorsqu'on augmente le nombre de neurones cachés (k).
On constate ensuite l'énorme prévalence des informations très proches (distance ≤ 5) par rapport aux informations distantes. Le RNN simple tire donc très peu profit du passé non immédiat. Cela explique donc bien son succès avec la grammaire de Reber et son échec avec la grammaire embarquée.

On peut calculer $D(l,k)$ analoguement en étendant la définition précédente pour un modèle LSTM. On trouve le profil suivant :

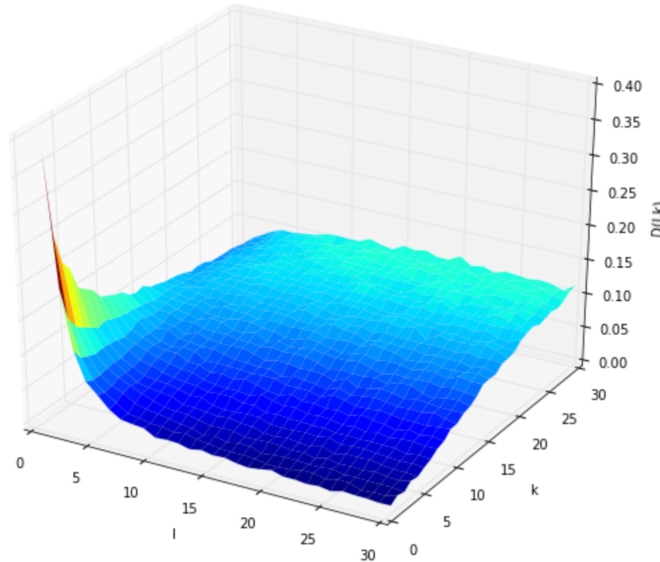


Figure 15 : D(l,k) pour un LSTM (GRU)

Ici, à partir de $k = 20$ on comprend qu'un LSTM compose de manière très homogène avec toutes les échelles de distances et est donc capable de traiter à égalité des dépendances court, moyen et long terme. En effet pour 30 neurones cachés et $l = 2000$ on trouve $D(l,k) = 0.8$ ce qui montre une très faible décroissance en l comparé au RNN simple.

Les résultats présentés rencontrent quelques limites :

- La mesure $D(l,k)$ est purement expérimentale
- Elle est liée à un choix d'alphabet $\mathbf{0}, \mathbf{1}$ et de séquences tests ($\mathbf{0}, \dots, \mathbf{0}$ et $\mathbf{1}, \dots, \mathbf{0}$)
- En pratique elle présente une variance non négligeable : en fonction des tests les valeurs peuvent osciller sensiblement

Dans les années 2000 est présentée la théorie des *Echo State Networks*, [15], qui propose un cadre théorique à ces considérations de mémoire pour les RNNs. Notamment elle montre rigoureusement les effets d'indistinguabilité et propose une mesure de capacité de mémoire théorique.

2.3 L'approche Echo State Networks

L'approche *Echo State Networks* – ESN dans la suite – est à la croisée de l'étude des systèmes dynamiques et du traitement du signal. Elle propose notamment une manière de concevoir des RNNs sans entraînement de la couche cachée. Cette méthode s'inscrit dans le cadre du *reservoir learning* qui considère que selon certaines conditions un RNN non entraîné fournit un réservoir de dynamiques suffisamment riche pour être exploité tel quel.
Le système dynamique étudié est toujours celui de l'équation (1).
On note :

- $\boldsymbol{T}$ la fonction telle que $\boldsymbol{h}_t = \boldsymbol{T}(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t)$
- $\bar{\boldsymbol{x}}^{\boldsymbol{l}} = (\boldsymbol{x}_0, \dots, \boldsymbol{x}_{l-1})$ un mot de taille l
- $\boldsymbol{h}_{t+l} = \boldsymbol{T}(\boldsymbol{h}_t, \bar{\boldsymbol{x}}^{\boldsymbol{l}})$ l'état après traitement du mot $\bar{\boldsymbol{x}}^{\boldsymbol{l}}$
- $\bar{\boldsymbol{x}}^{-\infty}$ un mot infini à gauche

Définition 2.2 (Propriété *echo states*)**.** Un réseau à la propriété *echo states* si une séquence infinie à gauche en entrée $\bar{\boldsymbol{x}}^{-\infty}$ détermine un unique $\boldsymbol{h}_t$. C'est à dire si pour toute séquence d'entrée infinie à gauche $\ldots, \boldsymbol{x}_{t-1}, \boldsymbol{x}_t$, pour toutes suites d'états cachés $\ldots, \boldsymbol{h}_{t-1}, \boldsymbol{h}_t, \ \ldots, \boldsymbol{h}'_{t-1}, \boldsymbol{h}'_t$ telles que :

$$\forall i \leq t \quad \boldsymbol{h}_i = \boldsymbol{T}(\boldsymbol{h}_{i-1}, \boldsymbol{x}_i)$$
$$\forall i \leq t \quad \boldsymbol{h}'_i = \boldsymbol{T}(\boldsymbol{h}'_{i-1}, \boldsymbol{x}_i)$$

On a $\boldsymbol{h}_t = \boldsymbol{h}'_t$.
C'est à dire qu'il existe une fonction $\boldsymbol{E}$ dite *echo* telle que :

$$\boldsymbol{h}_t = \boldsymbol{E}(\ldots, \boldsymbol{x}_{t-1}, \boldsymbol{x}_t)$$

L'état mémoire au temps t est uniquement fonction de l'historique des entrées.

Pour se convaincre de l'intérêt de cette définition on peut considérer un autre système dynamique comme :

$$\boldsymbol{h}_t, \boldsymbol{x}_t \in \mathbb{R} \quad \boldsymbol{h}_t = \boldsymbol{h}_{t-1} + \boldsymbol{x}_t + 1$$

Si on prend une séquence d'entrée nulle, conviennent en états mémoires $2\mathbb{Z}$ et $2\mathbb{Z}+1$: il n'y a pas unicité, le système n'est pas *echo states.*

L'auteur de [15] propose une condition suffisante pour qu'un RNN soit ESN :

Théorème 2.3 (Condition suffisante pour être ESN)**.**
Pour être ESN il suffit que la plus grande valeur singulière de la matrice W_h soit strictement inférieure à 1.

Cette condition n'est pas très contraignante et a en pratique été vérifiée dans tous les tests présentés jusqu'ici.
La propriété suivante, comme démontré dans [15], est équivalente à ESN et assure la prévalence des informations court terme pour un RNN.

Définition 2.4 (Propriété *input forgetting*)**.** Un RNN a la propriété *input forgetting* si pour toute séquence d'entrée infinie à gauche $\bar{\boldsymbol{x}}^{-\infty}$ il existe une suite tendant vers 0 $(\delta_l)_{l\geq 0}$ telle que pour tous suffixes $\bar{\boldsymbol{x}}^{\boldsymbol{l}} = \boldsymbol{x}_{t-l}, \ldots, \boldsymbol{x}_t$, pour toutes suites infinies à gauche de la forme $\bar{\boldsymbol{w}}^{-\infty}\bar{\boldsymbol{x}}^{\boldsymbol{l}}$ et $\bar{\boldsymbol{v}}^{-\infty}\bar{\boldsymbol{x}}^{\boldsymbol{l}}$ et pour tous états $\boldsymbol{h}, \boldsymbol{h}'$ respectivement compatibles avec $\bar{\boldsymbol{w}}^{-\infty}\bar{\boldsymbol{x}}^{\boldsymbol{l}}$ et $\bar{\boldsymbol{v}}^{-\infty}\bar{\boldsymbol{x}}^{\boldsymbol{l}}$ on a :

$$||\boldsymbol{h} - \boldsymbol{h}'|| \leq \delta_l$$

On entend par compatible un état tel qu'il existe une séquence d'états qui, couplée avec l'entrée spécifiée aboutit à cet état.

Il s'agit exactement du comportement observé figure 14 : plus l est grand plus les états auxquels le système aboutit sont proches.
Ainsi, intrinsèquement, un RNN qui a la propriété ESN − non contraignante − n'est pas capable de tirer également partie de tous les événements antérieurs.
Dans [16] l'auteur poursuit une étude approfondi de la capacité de mémoire des RNNs vus avec l'approche ESN. Son approche est la suivante : il s'agit de quantifier la capacité d'un ESN à recomposer un signal d'entrée délayé. Plus précisément on se donne un signal scalaire aléatoire stationnaire *i.i.d* $\nu(t)$ et on fait apprendre à chaque neurone de sortie y_i le signal $\nu(t-i)$. Comme mentionné plus haut il n'y a pas d'apprentissage au niveau de la couche cachée. Le réseau est simplement entraîné par régression linéaire sur sa couche de sortie. Ainsi le ième neurone de sortie doit correspondre au signal ν délayé de i.

Pour juger de la capacité du réseau à recomposer un signal l-délayé l'auteur utilise la mesure *Memory Capacity* :

$$MC_l = \frac{cov^2(\nu(t-l), y_l(t))}{\sigma^2(\nu(t))\sigma^2(y_l(t))}$$

Par construction $MC_l \in [0,1]$ et plus MC_l est proche de 1 plus la tâche de reconstruction est maîtrisée. La capacité totale du réseau, MC, est définie par :

$$MC = \sum_{l=1}^{\infty} MC_l$$

Les principaux résultats de l'auteur sont les suivants :

— MC_l est monotone décroissante en l
— Dans un réseau à k neurones cachés on a :

$$MC \leq k$$

— La borne k est *tight* avec une caractérisation algébrique des RNNs qui l'atteignent

Par analogie avec $D(l,k)$ on note $MC(l,k)$ MC_l pour un réseau à k neurones cachés. Le profil spatial de $MC(l,k)$ est le suivant :

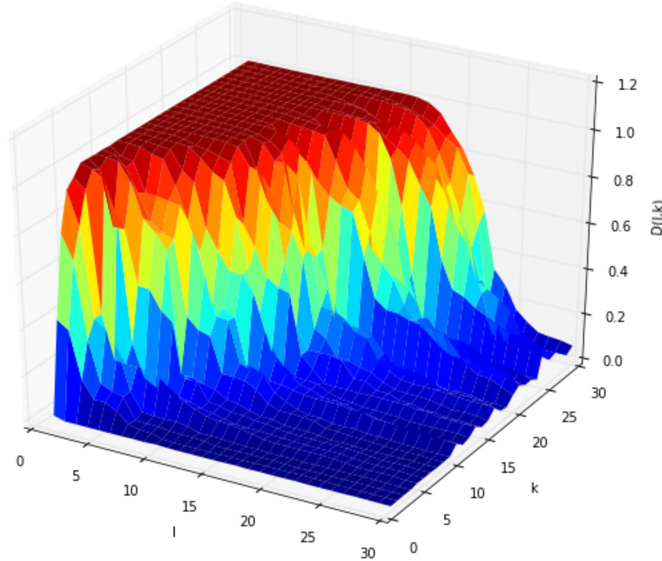


Figure 16 : MC(l,k) pour un ESN

Cette mesure est plus optimiste que $D(l,k)$ mais est issue d'une autre méthode d'entraînement. D'autre part par son fondement théorique, elle est très liée aux RNNs *Vanilla* et n'est, tel quel, pas utilisable dans le contexte des LSTMs. Les tentatives théoriques et expérimentales que nous avons effectuées se sont avérées infructueuses.

3 Application à la musique

3.1 Les chorals de Bach

Le choral est un élément important de la tradition lithurgique luthérienne. C'est une pièce courte, polyphonique, que l'on retrouve dans les oeuvres sacrées. Par leur simplicité les chorals pouvaient être chantés par tous les fidèles. La structure du choral est attirante car très canonique. En effet elle est découpée en différentes *périodes* cadentielles finissant chacune par un point d'orgue.

Aus meines Herzens Grunde BWV 269

Figure 17 : Choral issu de la cantate BWV 269 de Bach

Les chorals que nous considérons ici sont ceux de Johan-Sebastian Bach, à quatre voix. On en compte 382, la plupart issus des cantates – oeuvre plutôt courte pour choeur, instruments et solistes. En effet lorsque Bach était maître de chapelle à Leipzig il se devait, contractuellement, de composer une cantate par semaine pendant 5 ans pour la messe dominicale. Beaucoup ont été perdues mais elles fournissent tout de même la majorité des chorals connus.
Ces chorals sont assez utilisés en *machine learning* et servent souvent de benchmark ou de moyen de comparaison de modèles (cf. [14]). Il existe par exemple un dataset répandu, les regroupant sous forme de *piano-roll*, nommé *JSB Chorales* introduit par [17].

3.2 Méthode

La génération de polyphonie n'est a priori pas évidente, les modèles que nous avons proposés jusque là ne se généralise pas bien. En effet on peut voir une partition polyphonique comme une séquence à deux dimensions avec la représentation du *piano-roll* :

Figure 18 : Un *piano-roll*

On pourrait vouloir considérer cette matrice en la traitant vecteur colonne par vecteur colonne. Cependant il n'y a pas de modèle génératif en sortie puisqu'on ne prédit plus une distribution de probabilité sur l'élément suivant mais une autre colonne de *piano-roll.* Ainsi cette approche ne s'inscrit pas dans le contexte étudié.
Pour rester très proche du cadre décrit en partie 1 on se contente de générer des lignes de sopranos de chorals. C'est à dire la voix la plus aigües des quatre. Cela a du sens musicalement puisque historiquement c'est le travail du musicien que d'inférer les trois voix inférieurs étant donné le soprano.

Figure 19 : Soprano du choral de la figure 18

On extrait donc les soprani de tous les chorals de Bach, on a utilisé la transcription *lilypond* accessible ici : `http://superbonus.project.free.fr/spip.php?article48`.
On transpose ensuite tous les chorals dans la même tonalité de référence – do majeur/mineur – et on les convertit dans un format très simplifié.

Le soprano présenté en figure 19 devient :

```
c' |
c'2 g'4 |
e'4. d'8 c'4 |
c'4. d'8 e'4 |
d'2 \ e'4 |
g'2 f'4 |
e' d'2 |
c' \
e'4 |
e' f' g' |
g'4. f'8 e'4 |
d'2 \ c'4 |
e'2 f'4 |
g'2 f'4 |
e'2. |
c'2 \ e'4 |
g'2 f'4 |
e'2 d'4 |
c'4. d'8 e'4 |
d'2 \ e'4 |
g'2 f'4 |
e' d'2 |
c' \
```

Figure 20 : Représentation finale du soprano en figure 19

C'est avec ce format qu'on réalise l'apprentissage de séquences.
On entraîne Vanilla et LSTM avec un nombre de paramètres équivalent – environ 30.000 – à l'aide des codes fournis en annexe.

3.3 Résultats

Les résultats montrent la capacité des deux modèles à appréhender la structure globale des chorals – comme la découpe en période. Cependant la musique proposée n'est pas toujours très cohérente cela tient très certainement à une boucle d'apprentissage – minimisation de la fonction de coût – trop simpliste pour cette tâche. On arrive en fin d'optimisation à une erreur moyenne de 660 pour *Vanilla* contre 134 pour LSTM. Cependant les implémentations les plus performantes disponibles sur internet comme *torch-rnn*[3] tombent à 0.1.

(a) RNN (b) LSTM

Figure 21 : Exemple de partition générée par un Vanilla et un LSTM

On remarque cependant que le LSTM propose une forme d'ensemble plus cohérente qui respecte certaines règles d'harmonie : la première et la dernière note font parties de l'accord de tonique et les fins de périodes (les points d'orgues) pourraient se justifier harmoniquement. Les générations du Vanilla sont plus obscures comme en témoigne l'exemple donné.

3. https://github.com/jcjohnson/torch-rnn

Ces résultats sont encourageants. En mettant en oeuvre une technique d'apprentissage plus performante nous aurions très bon espoir de les améliorer et de pouvoir considérer les problématiques de mémoire chez les RNNs au travers des règles d'harmonie qui sont très dépendantes des éléments de contexte court, moyen et long terme.

Conclusion et perspectives

Avec l'exemple de la grammaire de Reber embarquée on a mis en évidence que la supériorité des LSTMs ne semble pas être majoritairement du fait des raisons de gradient invoquées. La mesure de distinguabilité propose une explication qui semble pertinente et qui est en lien avec la théorie des *ESNs*. Enfin le caractère intrinsèquement séquentiel de la musique et la structure très canonique des chorals de Bach proposent un moyen très attirant d'étudier ces modèles. En effet, au moyen d'une expertise humaine en harmonie, il serait possible de développer une intuition de la manière dont les RNNs exploitent leur mémoire à différentes échelles de temps.
Dans le futur il serait intéressant de pouvoir formaliser $D(l,k)$ en s'inspirant de l'approche *ESN* mais en la rendant adaptables aux LSTMs. Par exemple pouvoir donner des vitesses de convergence de la suite δ_h introduite à la définition *input forgetting* et une explication théorique des profils $D(l,k)$ observés serait satisfaisant. Cela permettrait de comprendre fondamentalement d'où vient, dans les équations, la supériorité des LSTMs et sans doute de pouvoir simplifier ce modèle qui est encore lourd en calcul comparé aux RNNs *Vanilla*. Enfin que ce soit pour valider une théorie ou pour faire naître des intuitions il conviendrait d'améliorer la méthode d'apprentissage de notre modèle musical et pourquoi pas d'y intégrer la polyphonie. À l'image des discussions qui ont suivi l'étude de la grammaire de Reber nous sommes convaincus que la musique peut nous aider à mieux comprendre ces modèles.

Références

[1] Rosenblatt F., ‘The perceptron : a probabilistic model for information storage and organization in the brain”, *Psychological Review*, **65**, 6, (1958).

[2] Hornik K., Stinchcombe M., White H., “Multilayer feedforward networks are universal approximators”, *Neural Networks*, **2**, p. 359-366 (1989).

[3] reyfus G., Martinez J.-M, Samuelides M., Gordon M. B., Badran F., Thiria S., *Apprentissage statistique*, p. 82, Eyrolles (2008).

[4] Barron A., “Universal Approximation Bounds for Superpositions of a Sigmoidal Function”, *IEEE transactions on information theory,* , **39**, p. 930 (1993).

[5] LeCun Y., Bottou L., Orr G., Müller K., **Efficient BackProp**, in Orr, G. and Muller K. (Eds), *Neural Networks : Tricks of the trade*, Springer, (1998).

[6] T. Siegelmann H., D. Sontag E., “On the Computational Power of Neural Nets”, *Journal of computer and system sciences*, **50**, p. 132-150, (1995)

[7] Bengio Y., Sinard P., Frasconi P., “Learning Long-Term Dependencies with Gradient Descent is Difficult”, *IEEE transactions on neural networks*, **5**, 2, (1994)

[8] Werbos P., “Backpropagation Trhough Time : What It Does and How to Do It”, *Proceedings of the IEEE*, **78**, 10, (1990)

[9] Hochreiter S., Schmidhuber J., “LONG SHORT-TERM MEMORY”, *Neural Computation*, **9**, p. 1735-1780, (1997)

[10] Graves A., Wayne G., Danihelka I., at **DeepMind**, “Neural Turing Machines”, *arXiv*, (2014)

[11] Reed S., Freitas N., at **DeepMind**, “Neural Programmer-Interpreters”, *ICLR 2016*, (2016)

[12] Karpathy A., “The Unreasonable Effectiveness of Recurrent Neural Networks”, http ://karpathy.github.io/2015/05/21/rnn-effectiveness/, (2015)

[13] Cho K., Merriënboer B., Bahdanau D., Bengio Y., “On the Properties of Neural Machine Translation : Encoder-Decoder Approaches”, arXiv, (2014)

[14] Chung J., Gulcehre C., Cho K., Bengio Y., “Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling”, arXiv, (2014)

[15] Jaeger H., “The "echo state" approach to analysing ang training recurrent neural networks”, TechReport, http ://minds.jacobs-university.de/sites/default/files/uploads/papers/EchoStatesTechRep.pdf, (2001)

[16] Jaeger H., “Short term memory in echo states networks”, TechReport, http ://minds.jacobs-university.de/sites/default/files/uploads/papers/STMEchoStatesTechRep.pdf, (2002)

[17] Boulanger-Lewandowski N., Bengio Y., Vincent P., “Modeling Temporal Dependencies in High-Dimensional Sequencer : Application to Polyphonic Music Gneration and Transcription”, arXiv, (2014)

Annexe A Principaux éléments de code

Pendant ce stage j'ai principalement travaillé en python avec la librairie **theano** qui est orientée *machine learning*. J'ai aussi découvert l'interface python **jupyter** avec laquelle j'ai réalisé l'intégralité de mes tests.

A.1 Implémentation des RNNs Vanilla

Voici le code qui implémente les RNNs *Vanilla* :

```
import numpy as np
import random
import theano
import theano.tensor as T
import collections as c
import copy

class RNN:

   def __init__(self, n_i, n_h, n_o):
      self.n_i = n_i
      self.n_h = n_h
      self.n_o = n_o

      self.init_W_x = np.random.randn(self.n_h,self.n_i)
      self.init_W_h = np.random.randn(self.n_h,self.n_h)
      self.init_W_y = np.random.randn(self.n_o,self.n_h)

      self.n_parameters = self.n_h*self.n_i
      +self.n_h*self.n_h+self.n_h + self.n_o*self.n_h

      self.W_x = theano.shared(copy.deepcopy(self.init_W_x), name='W_x')
      self.W_h = theano.shared(copy.deepcopy(self.init_W_h), name='W_h')
      self.W_y = theano.shared(copy.deepcopy(self.init_W_y), name='W_y')
      self.b1 = theano.shared(np.zeros(self.n_h), name='b')
      self.b2 = theano.shared(np.zeros(self.n_o), name='b')

      self.params = [self.W_x,self.W_h,self.W_y,self.b1, self.b2]

      self.__theano_build__()

   def __theano_build__(self):
      W_x, W_h, W_y, b1, b2 = self.W_x, self.W_h, self.W_y,
      self.b1, self.b2

      #First dim is time
      x = T.matrix()
      #target
      t = T.matrix()
      #initial hidden state
      h0 = T.vector()
```

```
    def step(x_t, h_tm1, W_x, W_h, W_y, b1, b2):
        h_t = T.nnet.sigmoid(T.dot(W_x, x_t)+T.dot(W_h, h_tm1)+b1)
        y_t = T.dot(W_y, h_t)+b2
        return y_t, h_t

    [y, h], _ = theano.scan(step,
                            sequences=x,
                            non_sequences=[W_x,W_h,W_y, b1, b2],
                            outputs_info=[None, h0])


    error = ((y - t) ** 2).sum()
    gW_x, gW_h, gW_y, gW_b1,gW_b2 = T.grad(error, [W_x, W_h, W_y, b1, b2])



    self.model = theano.function([x, h0], (y,h))
    self.get_error = theano.function([x, t, h0], error)
    self.bptt = theano.function([x, t, h0], [gW_x, gW_h, gW_y, gW_b1, gW_b2]


    lr = T.scalar()
    self.train_step = theano.function([h0, x, t, lr],
                                      (y, h, error),
                updates=c.OrderedDict({W_x: W_x - lr * gW_x,
                                       W_h: W_h - lr * gW_h,
                                       W_y: W_y - lr * gW_y,
                                       b1: b1 - lr*gW_b1,
                                       b2: b2 - lr*gW_b2}))

def reset(self, random_init=True):
    if random_init:
        self.init_W_x = np.random.randn(self.n_h,self.n_i)
        self.init_W_h = np.random.randn(self.n_h,self.n_h)
        self.init_W_y = np.random.randn(self.n_o,self.n_h)
    self.W_x.set_value(self.init_W_x)
    self.W_h.set_value(self.init_W_h)
    self.W_y.set_value(self.init_W_y)
    self.b1.set_value(np.zeros(self.n_h))
    self.b2.set_value(np.zeros(self.n_o))
```

A.2 Implémentation des LSTMs (GRU)

Voici le code qui implémente les RNNs LSTM (GRU) :

```
import numpy as np
import random
import theano
import theano.tensor as T
import collections as c
import copy

class GRU:

   def __init__(self, n_i, n_h, n_o):
      self.n_i = n_i
      self.n_h = n_h
      self.n_o = n_o

      self.init_W_z = np.random.randn(self.n_h,self.n_i)
      self.init_U_z = np.random.randn(self.n_h,self.n_h)

      self.init_W_r = np.random.randn(self.n_h,self.n_i)
      self.init_U_r = np.random.randn(self.n_h,self.n_h)

      self.init_W_h = np.random.randn(self.n_h,self.n_i)
      self.init_U_h = np.random.randn(self.n_h,self.n_h)

      self.init_W_y = np.random.randn(self.n_o,self.n_h)

      self.n_parameters = 3*self.n_h*self.n_i+
      3*self.n_h*self.n_h + self.n_o*self.n_h

      self.W_z = theano.shared(copy.deepcopy(self.init_W_z), name='W_z')
      self.U_z = theano.shared(copy.deepcopy(self.init_U_z), name='U_z')

      self.W_r = theano.shared(copy.deepcopy(self.init_W_r), name='W_r')
      self.U_r = theano.shared(copy.deepcopy(self.init_U_r), name='U_r')

      self.W_h = theano.shared(copy.deepcopy(self.init_W_h), name='W_h')
      self.U_h = theano.shared(copy.deepcopy(self.init_U_h), name='U_h')

      self.W_y = theano.shared(copy.deepcopy(self.init_W_y), name='W_y')

      self.b1 = theano.shared(np.zeros(self.n_h), name='b1')
      self.b2 = theano.shared(np.zeros(self.n_h), name='b2')
      self.b3 = theano.shared(np.zeros(self.n_h), name='b3')

      self.params = [self.W_z,self.U_z,self.W_r,self.U_r, self.W_h, self.U_h,
      self.W_y, self.b1, self.b2, self.b3]

      self.__theano_build__()
```

```
def __theano_build__(self):
    params = self.params

    #First dim is time
    x = T.matrix()
    #target
    t = T.matrix()
    #initial hidden state
    s0 = T.vector()


    def step(x_t, s_tm1, W_z, U_z, W_r, U_r, W_h, U_h, W_y, b1, b2, b3):
        z = T.nnet.sigmoid(W_z.dot(x_t)+U_z.dot(s_tm1)+b1)
        r = T.nnet.sigmoid(W_r.dot(x_t)+U_r.dot(s_tm1)+b2)
        h = T.tanh(W_h.dot(x_t)+U_h.dot(s_tm1*r)+b3)
        s_t = (1-z)*h + z*s_tm1
        y_t = T.nnet.sigmoid(W_y.dot(s_t))
        return y_t, s_t,z,r

    [y,s,z,r], _ = theano.scan(step,
                              sequences=x,
                              non_sequences=params,
                              outputs_info=[None, s0, None, None])



    error = ((y - t) ** 2).sum()
    grads = T.grad(error, params)



    self.model = theano.function([x, s0], (y,s,z,r))
    self.get_error = theano.function([x, t, s0], error)
    self.bptt = theano.function([x, t, s0], grads)


    lr = T.scalar()

    chgt = {}
    for i in range(len(params)):
        chgt[params[i]] = params[i]-lr*grads[i]

    self.train_step = theano.function([s0, x, t, lr],
                                      (y, s, error),
                                      updates=c.OrderedDict(chgt))

def reset(self, random_init=True):
    if random_init:
        self.init_W_z = np.random.randn(self.n_h,self.n_i)
        self.init_U_z = np.random.randn(self.n_h,self.n_h)

        self.init_W_r = np.random.randn(self.n_h,self.n_i)
```

```
        self.init_U_r = np.random.randn(self.n_h,self.n_h)

        self.init_W_h = np.random.randn(self.n_h,self.n_i)
        self.init_U_h = np.random.randn(self.n_h,self.n_h)

        self.init_W_y = np.random.randn(self.n_o,self.n_h)

    self.W_z.set_value(self.init_W_z)
    self.U_z.set_value(self.init_U_z)

    self.W_r.set_value(self.init_W_r)
    self.U_r.set_value(self.init_U_r)

    self.W_h.set_value(self.init_W_h)
    self.U_h.set_value(self.init_U_h)

    self.W_y.set_value(self.init_W_y)

    self.b1 = theano.shared(np.zeros(self.n_h), name='b1')
    self.b2 = theano.shared(np.zeros(self.n_h), name='b2')
    self.b3 = theano.shared(np.zeros(self.n_h), name='b3')
```

A.3 Boucle d'apprentissage

Enfin voici la boucle d'apprentissage – qu'il conviendrait d'améliorer – pour GRU mais celle des RNNs simples est tout à fait similaire :

```
gru.reset()

n_epochs = 1000
lr = 0.01
epsilon = 1e-5
hm1= [0]*n_hidden

last_err = 0.0
for i in range(n_epochs):
    err_moy = 0.0
    for d in dataset:
        err_moy += gru.train_step(hm1,d[0],d[1],lr)[2]
    err_moy /= float(len(dataset))

    if i != 0 and last_err - err_moy < 0:
        lr /= 1.5

    print "Iteration %d: error of %f, lr of %f" % (i, err_moy, lr)

    if abs(last_err-err_moy) < epsilon:
        print "Early stopping, iteration %d: error of %f, lr of %f"
        % (i, err_moy, lr)
        break

    last_err = err_moy
    i += 1
```

Annexe B Contexte institutionnel

J'ai été accueilli par l'équipe Neucod du département délectronique de Télécom Bretagne qui travaille à la frontière de plusieurs domaines, tels que linformatique, lélectronique, la psychologie et les neurosciences, pour établir des modèles à la fois efficaces et biologiquement plausibles. Le laboratoire a fondé le projet NEUCOD (pour Neural Coding) qui vise à identifier et exploiter les analogies observées entre les propriétés du cortex cérébral, et celles des décodeurs correcteurs derreurs modernes. Les sujets les plus répandus en informatique dans l'équipe sont : les réseaux à clique – dits Gripon-Berrou –, le traitement de signal sur graphe, l'analyse des EEG et les modèles neuronaux en général. Avec les autres stagiaires nous avons été totalement intégrés dans l'équipe, participant aux différentes réunions, présentations, conférences et activités.
Ce fut une très belle et enrichissante première expérience du monde de la recherche, aussi bien sur le plan scientifique qu'humain.