Cours OpenClassrooms

Les Récurrents Neurals Networks

Contenu

[Introduction 3](#_Toc498014343)

[Fonctionnement d'un Récurrent Neural Network 3](#_Toc498014344)

[Utilisation des Récurrent Neural Networks 4](#_Toc498014345)

[Etat de l'art 5](#_Toc498014346)

[Dans les 70's 5](#_Toc498014347)

[En 1997 5](#_Toc498014348)

[En 2014 6](#_Toc498014349)

[En 2016/2017 6](#_Toc498014350)

[Les différents Cellules 7](#_Toc498014351)

[LTSM 7](#_Toc498014352)

[GRU 8](#_Toc498014353)

[QRNN 9](#_Toc498014354)

[Evaluation / Comparaison 9](#_Toc498014355)

[Présentation de l'exercice 9](#_Toc498014356)

[Code 10](#_Toc498014357)

[Imports 10](#_Toc498014358)

[Creation des fonctions utiles 11](#_Toc498014359)

[Génération du Dataset 12](#_Toc498014360)

[Creation du Model Many-to-Many 12](#_Toc498014361)

[Creation du générateur 13](#_Toc498014362)

[Résultat 13](#_Toc498014363)

[Conclusion 15](#_Toc498014364)

[Annexe 16](#_Toc498014365)

[Annexe 1 : 16](#_Toc498014366)

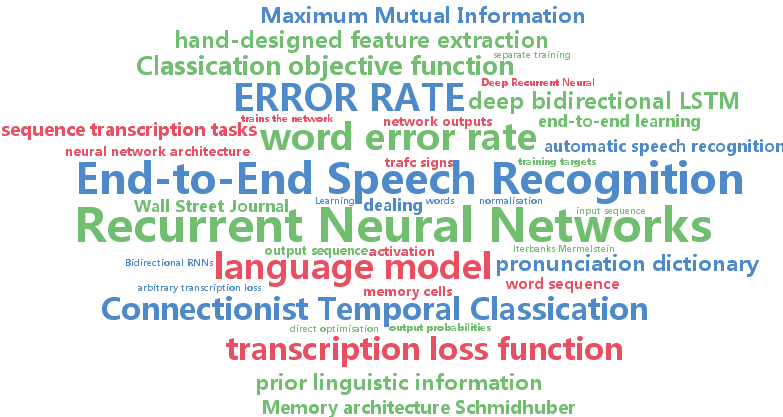
[Annexe 2 : 16](#_Toc498014367)

[Sources : 16](#_Toc498014368)

# Introduction

Lors des différentes formations sur OpenClassrooms, on a pu appréhender différentes parties de la Data Science incluant les Réseaux de Neurones Artificiels. Ceux-ci sont inspirés du fonctionnement du cerveau. On a vu notamment les *Feed Forward Neural Networks* utilisés en Régression/Classification non linéaires ou les *Convolutionnal Neural Networks* utilisés pour le traitement d'images.

Cependant ces modèles ne permettent pas de travailler sur des données temporelles. Or, il existe un type de Neural Network qui permet de traiter des données temporelles (vidéos, sons, texte, …). Ce sont les ***Recurrent Neural Networks***. Ceux si sont ont été longtemps peu utilisés pour des raisons que l'on verra par la suite mais sont depuis peu extrêmement populaires et utilisés notamment pour *la reconnaissance de paroles (Natural Langage Processing et Speech Recognition),* la traduction de texte ou encore l'analyse de sentiments comme le montre le WordCloud suivant.



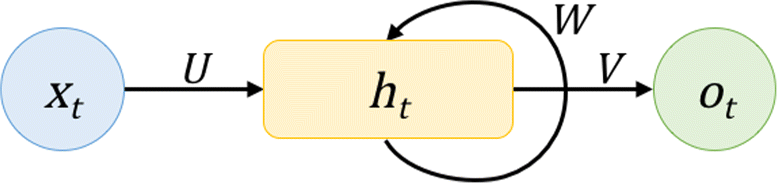
Dans ce cours nous passerons en revue l'état de l'art sur ce type de Réseaux de Neurones, les différents modèles qui existent ainsi que leurs avantages/inconvénients.

# Fonctionnement d'un Recurrent Neural Network

Comme on l'a vu, les Réseaux de Neurones Artificiels fonctionnent en 2 étapes, une phase de **front-propagation** qui va de l'entrée vers la sortie. Selon la prédiction en sortie, une erreur est calculée et propagé en arrière proportionnellement à son apport dans l'erreur, c'est la **back-propagation**. Plus une connexion participera à l'erreur, plus elle sera corrigée. Le principe est le même que la régression linéaire par batch (cf. l'annexe 1). Les différents poids de chaque neurones sont donc ajusté lors de chaque batch.

Cependant avec les Recurrent Neural Network, il y a une corrélation entre la n-ième données et toutes les précédentes. Pour ce faire, l'idée de base a été de boucler la sortie de chaque neurones sur une de ses entrées. Sur un Réseau de Neurone Artificiel, la sortie d'un layer peut se représenter comme :

Avec *f* la fonction d'activation choisie (la plus courante étant la Sigmoïde mais il en existe des dizaines, cf. Annexe 2). Pour les Récurrent Neural Network de base, la représentation est la suivante

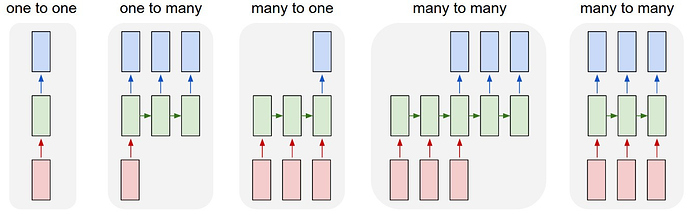


Et la sortie est donc :

Avec *f* et *h*, 2 fonctions d'activations (pouvant être identiques). On a donc un modèle qui dépend des n-précédentes étapes par récurrence. Malheureusement, ce principe simple pose divers problèmes qui ont menés à des améliorations que l'on va voir dans le prochain chapitre. Avant ce ci, voyons les différentes topologies existantes.

### Utilisation des Récurrent Neural Networks

Il existe différentes façon d'arranger un Recurrent Neural Network. On peut retrouver ci-dessous les différents types :



Le One-to-One:

C'est l'utilisation la plus basique d'un RNN. En effet, celui-ci va prédire la sortie suivante basé sur une entrée. Imaginons que l'on entraine ce type de RNN sur un jeu de morpion. On lui donne en entrée la position ou l'on pose un pion, il sera "deviner" la prochaine position à prendre. Cela est vrai car un coup n'a pas particulièrement de dépendance avec les n-précédents coups.

Le One-to-Many:

Ce modèle est utilisé comme un générateur "simple". On lui donne une entrée et il va prédire les n-steps suivantes. Par exemple ce type de modèle pourrait générer de la musique. En l'entrainant sur des partitions, la 1ère note lui permettra de déterminer la suivante, ainsi que celle qui suit etc…

Le Many-to-One :

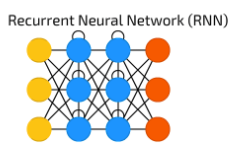
Ce modèle permet de faire des estimations sur des phénomènes "dynamiques". En effet, le RNN va prendre en compte plusieurs timesteps pour déterminer la suivante. Supposons le lancer d'une balle. Si on a en entrée la position x et y, le modèle One-to-One n'arrivera pas a prédire la prochaines position car il ne connais pas la direction et la vitesse. Un many-to-One model va prendre en compte par exemple les 3 dernières positions pour estimer la suivante. Le modèle pourra "estimer" la vitesse et la direction grâce a ces données. Ce type de modèle est aussi utilisé pour de la classification.

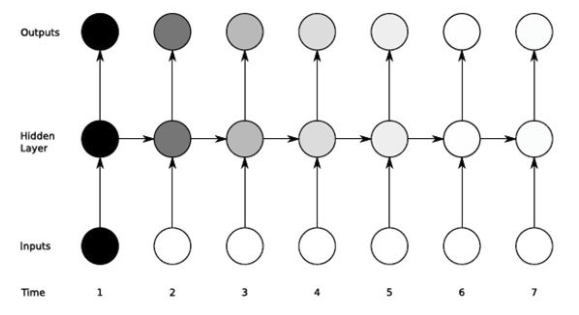
Le Many-to-Many :

Le 1er modèle de Many-to-Many est un mix entre le Many-to-One and One-to-Many. Il sert donc de générateur basé sur plusieurs timesteps. Si on reprend l'exemple du lancer de balle, le modèle retournera les n-steps suivantes. Quant au second modèle il permet la prédiction de texte. Par exemple, si la 1ere Entrée est "Je", le modèle va surement prédire "suis". En 2nd timestep, on lui donne en entrée "mange", il va surement prédire "une". En entrée, on va lui donner "une" et il va peut-être prédire "Pomme" etc..

# Etat de l'art

## Dans les 70's

 Comme nous avons vu dans le chapitre précédent, le principe de fonctionnement d'un *Recurrent Neural Network* de base est de boucler la sortie d'un layer sur son entrée pour créer une recursion. Ce modèle est celui qui a été proposé dans les années 70. Ils font partie de la famille des Fully Recurrent Neural Networks. Plutôt simples, ils ont le bénéfice d'être rapide au niveau du calcul. En contrepartie, il a un très gros défaut. Le temps d'entrainement est très lent car la backpropagation ne se fait plus uniquement sur les divers layers mais dans le temps. C'est ce que l'on appelle la *BackPropagation Throught Time* car à chaque timestep, il faut propager l'erreur sur ses divers levels mais aussi sur ses n-1 timesteps. On a donc de plus gros problèmes de Vanishing gradient.



De ce fait, à delà de 30-40 timestep, il est très complexe de l'entrainer. Par exemple il est très mauvais sur de la reconnaissance vocale ou l'on a des centaines/milliers de timesteps.

### En 1997

Il aura fallu attendre 1997 pour que 2 scientifiques, Sepp Hochreiter and Jürgen Schmidhuber, publient un article sur une nouvelle façon de penser les Récurrent Neural Network. Au lieu de boucler la sortie sur l'entrée, ils ont mis en place un système de mémoire. Cette mémoire est une cellule composé de 2 portes (Input Gate et Output Gate) ainsi que 2 états (Hidden State et Cell State). La Hidden State est une cellule qui contient les informations de l'état précédent (short term memory). Cet état va être utilisé aussi pour updater à la Cell State (mémoire à long terme) qui elle va être transférer d'une cellule à l'autre. C'est l'utilisation de ces portes sur les états qui permet de filtrer les informations importantes pour les propager à la cellule suivante (via la Cell State et Hidden State). C'est cette nouvelle architecture qui lui donne le nom de *Long Short-Term Memory*. En effet, cette cellule apprend à chaque timestep les informations importante et peut les garder pendant une longue période.

En 2000, un professeur en informatique avec son équipe a amélioré le modèle en ajoutant une 3ème porte appelé la *forget gate*. Cette modification a permis à la cellule en plus d'apprendre les informations importante, d'oublier certaines informations devenues inutiles. Cette modification a fait basculer son utilisation au stade qu'elle en est actuellement. En effet, ce modèle est extrêmement performant sur de très longue séquences (phrase, texte entier, musique). Il est donc très utilisé pour la reconnaissance vocale. C'est notamment cette algorithme utilisé par Apple au lancement de Siri.

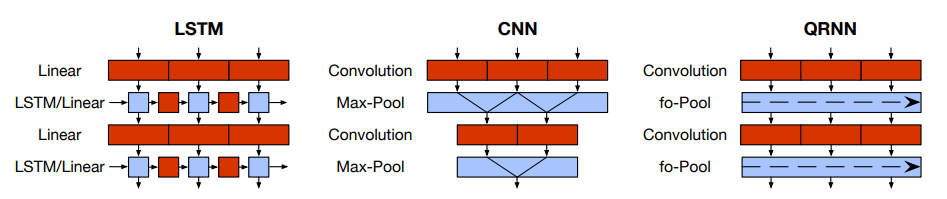
### En 2014

Depuis quelques années, un nouveau modèle fait de l'ombre au Long Short-Term Memory**.** Le principe reste le même mais on remplace la LSTM Cell par une autre cellule appelé *Gated Recurrent Unit.* L'intérêt de cette cellule est d'être plus légère en termes de calcul pour un résultat similaire. Le LSTM possède 3 gates (Input Gate, Output Gate and Forget Gate) et 2 états alors que le GRU n'a que 2 gates (Update Gate et Reset Gate).

### En 2016/2017

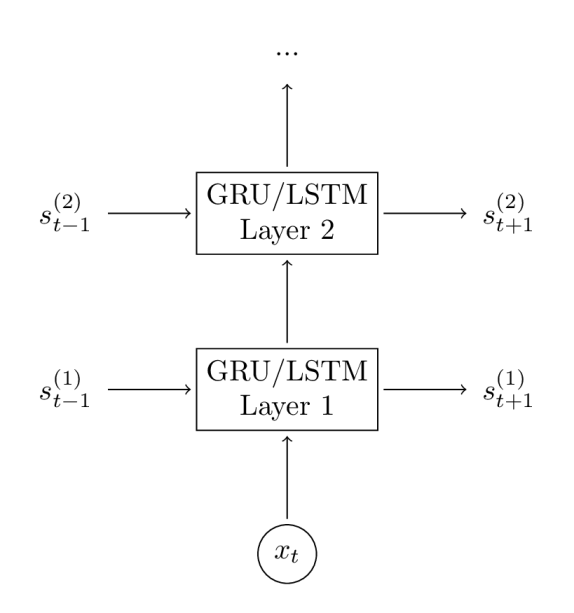
Le 5 Novembre 2016, un nouveau modèle a aussi été publié et risque de prendre la lumière. En effet, celui-ci fait table rase du système de mémoire et s'inspire des layers de convolution.

Au lieu d'entrainer, pour chaque inputs, les n-timestep, le modèle regroupe l'ensemble du batch par timestep et applique une étape de convolution.



Au lieu d'entrainer en série un modèle, chaque timestep sera effectué en parallèle. Une fonction finale viendra ensuite les regrouper. L'intérêt de ce modèle ? Un training en parallèle allant jusqu’à 16 fois plus vite et avec une précision à topology identique supérieure au LSTM.

Pour tous les modèles il est possible de stacker les niveaux si le moèle un plus complexe. Pour les Simple RNN, cela fonctionne comme pour les FeedForward Neural network en ajoutant des Hidden Layers. Pour les Cellules LSTM/GRU, elles peuvent aussi se stacker de facon similaires. La différence est que chaque cellule propagera sa mémoire dans les timesteps suivantes (de gauche à droite).

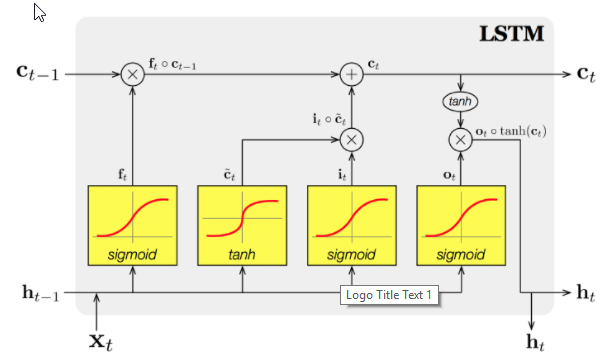


# Les différents Cellules

On a vu dans la présentation des Récurrent Neural Network, leur fonctionnement initial. Dans ce chapitre, nous allons aborder le fonctionnement des Cellule LSTM et GRU.

### LTSM

La cellule de LSTM peut se représenter linéairement comme suit :



Cette cellule est une variation du LSTM standard qui n'a pas de Input Modulation Gate. On va donc oublier cette connexion. On remarque donc que l'on a en entrée la sortie predite precedement (h\_t-1) ainsi que l'entrée actuelle (X\_t). Ces données vont être décomposées en 3 flux. L'objectif étant d'updater la Cell State pour les états futurs.

Le 1er flux va sur l'Input Gate. Cette gate, comme toutes les gates est en faite un perception. Lors du training, ses poids seront ajustés pour "comprendre quel feature est utile. La sortie de cette gate se fait avec une sigmoïde. Pour chaque input, la gate fournira en sortie une valeur. Plus la valeur sera grande, plus son interet de la garder sera élevé. La fonction est donc :

Ce résultat est ensuite multiplié (element-wise) par l'actuel Cell State tu mettre à jour ses poids.

Le 2nd flux va sur la forget gate qui est composé de 2 perceptron mais avec des poids différents.

Le 1er perceptron aura pour objectif de générer à partir des entrées un candidat pour l'update de la Cell State. Ce perceptron utilise généralement une fonction d'activation qui peut etre négative comme la tangente ou tangente Hyperbolique. Sa sortie sera donc :

Le second perceptron a le même objectif que l'input gate, selectionner les feature importantes. Son equation est donc la même hormis les matrices de poids :

Ces 2 resultat seont aussi multiplié element-wise et ajouté à la Cell State. A ce stade la Cell State vaut :

C'est cette matrice qui sera transférée à la prochaine cellule mais pour avoir la sortie de cette timestep, il faut passer par la forget gate .

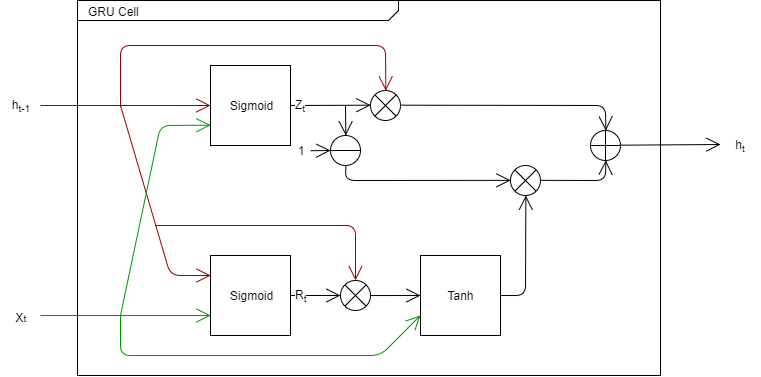
Celle-ci est aussi composée d'un perceptron basé sur les entrées :

Mais utilise aussi la Cell State pour ponderer l'importance des inputs (on utilise donc la mémoire en parralèle des entrée pour predire la sortie). L'equation de sortie est donc :

Le modèle de base d'un LSTM se compose donc de 4 "perceptions" qui ont pour abjectif de générer/updater la mémoire à long terme.Lors de l'entrainement,il faut donc faire la backpropagation uniquement sur ces matrices de poids ce qui simplifie très nettement les calculs et evite la perte de gradient.

### GRU

Le Modèle de Gated Reccurent Unit peut se représenter comme suit :



A l'opposé de la Cellule LSTM ou on a le Hidden State qui sert de mémoire courte et Cell State de mémoire long terme, le GRU ne possède qu'un seul flow qu'est le hidden State.

La cellule se compose donc de 3 "perceptrons". Sur le diagramme ci-dessus, les bias ne sont pas représentés pour simplifier le modèle. Deplus, les cellule ne sont pas exactement des perceptrons car le weight de chaque entrée n'est pas un float mais un vecteur de la taille de l'Hidden State ou de l'Input. Les 2 premiers "perceptrons" sont utilisés pour générer une Update Gate et la Reset Gate.

Les équations des 2 gates sont donc :

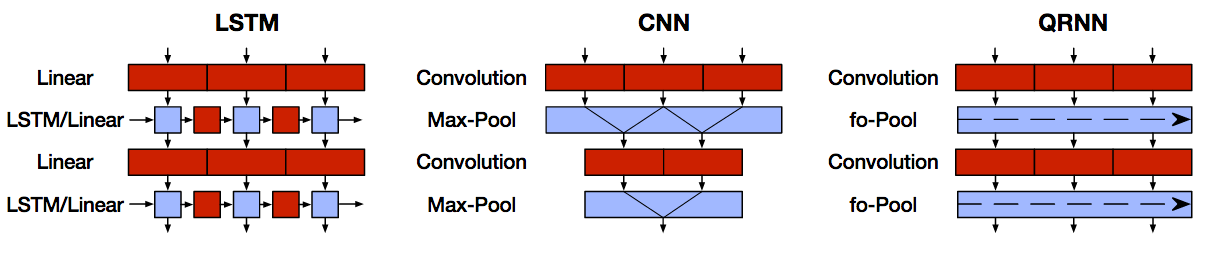
La sortie de ces 2 gates permettent de filtrer les informations a garder ou effacer. Ces valeurs sont donc multipliés element\_wise avec l'Hidden State précédent pour avoir les valeurs a garder.

La sortie de la Rest State va ensuite passer dans le 3eme perceptron afin d'extraire les information utiles sur l'input de l'état actuel. Cette sortie est ensuite multiplié à l'inverse de l'update gate. L'inversion est faite pour faire peser le pour du contre entre le reset et l'update. L'update State est ensuite additionnée à la sortie de la reset state poour fournir le prochain Hidden State qui sera propagé.

L'équation finale est donc :

### QRNN

Malheureusement le QRNN n'a pas encore beaucoup d'information concernant son fonctionnement car il est très récent.

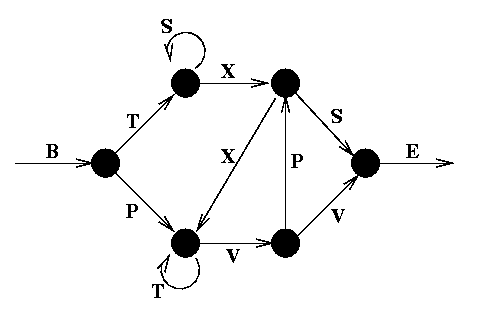


L'objectif était de ne plus être dépendant de la timestep pour pouvoir paralléliser la tâche. L'entrée linéaire pour les LSTM est donc remplacé par une convolution et la cellule LSTM/GRU est remplacé par un layer appelé fo-Pool qui équivaut le Max-Pooling Layer dans un CNN. Il existe néamoins quelques implémentations disponible que nous comparerons aux autres modèles dans la partie suivante.

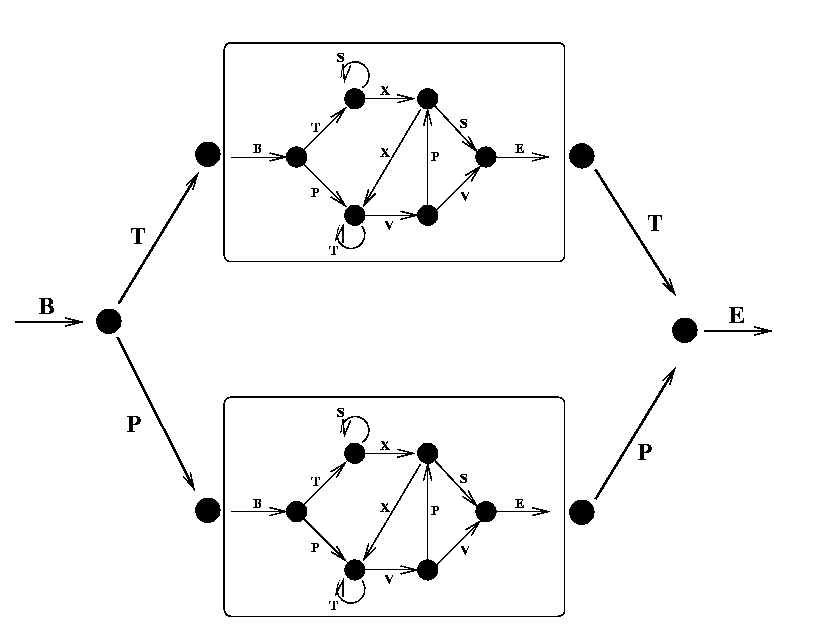
# Evaluation / Comparaison

## Présentation de l'exercice

Pour la comparaison des différents modèles, nous allons utiliser un exercice courant pour évaluer les RNN (à l'instart de MNIST pour les CNN). C'est la grammaire de Reber. Cette grammaire est basé sur le graphe suivant :



Un mot donné est dit de Reber s'il suit se graphe. Par exemple BTSSXSE est un mot valide alors que BPTSXSE n'est pas valide car il y a un saut entre T et S. Certes S peut suivre T mais que s'il est précédé de B. Il existe aussi une version Embedded qui utilise le graphe suivant :



Cependant pour cette evaluation nous allons rester sur le graphe Standard. L'objectif ici ne sera pas de classifier si un mot est de Reber ou non mais d'entrainer un modèle dessus puis d'en faire un générateur (Passage de Many-to-Many à One-to-Many). Pour des raisons de clarté, le model sera fait sur Keras et la librairie utilisée poru générer des mots de Reber sera celle présente sur [cette page](http://christianherta.de/lehre/dataScience/machineLearning/neuralNetworks/reberGrammar.php)

## Code

Passons donc au code :

### Imports

**import** numpy **as** np

**import** seaborn **as** sns

**import** time

**import** create\_dataset **as** reber

**from** keras**.**callbacks **import** ModelCheckpoint**,** EarlyStopping

**from** keras**.**models **import** Sequential**,** load\_model

**from** keras**.**layers **import** LSTM**,** SimpleRNN**,** GRU

**from** keras**.**preprocessing **import** sequence

### Creation des fonctions utiles

**def** generate**(**x0**):**

"""

Create the output vector for every input. It's only a Shift of characters

"""

end **=** np**.**array**([**0.**,** 0.**,** 0.**,** 0.**,** 0.**,** 0.**,** 1.**])**

y**=**x0**[**1**:]**

y**.**append**(**end**)**

**return** y

**def** Pick\_From\_Output**(**x**):**

"""

Pick a letter based on the output probability (except if output is below 0.1)

"""

y **=** np**.**zeros\_like**(**x**)**

x **=** np**.**where**(**x **<** 0.1**,** 0**,** x**)**

x **=** x**[**0**]/**x**[**0**].**sum**(**axis**=**1**)**

i **=** np**.**random**.**choice**(**list**(**range**(**7**)),** size**=**1**,** p**=**x**[**0**])**

y**[**0**,**0**,**i**]** **=** 1

**return** y

**def** evaluate**(**model**,** nb\_word **=** 1**,** max\_iter **=** 50**):**

"""

Make the generator create nb\_word of max length = max\_iter

Verify is the word is a Reber word of not and return the accuracy on the nb\_word created

"""

good\_pred **=** 0

**for** \_ **in** range**(**nb\_word**):**

model**.**reset\_states**()**

first\_input **=** np**.**array**([[[**1**,**0**,**0**,**0**,**0**,**0**,**0**]]])**

word **=** "B"

loop **=** 0

nextLetter **=** "B"

next\_seq **=** first\_input

**while** nextLetter **!=** "E" **and** loop **<** max\_iter**:**

y\_pred **=** model**.**predict**(**next\_seq**)**

next\_seq **=** Pick\_From\_Output**(**y\_pred**)**

nextLetter **=** reber**.**sequenceToWord**(**next\_seq**[**0**])**

loop **+=** 1

word **+=** nextLetter

**if** reber**.**in\_grammar**(**word**):**

good\_pred **+=** 1

acc **=** 100**\***good\_pred**/**nb\_word

**print(**"Good prediction : {:.2f}%"**.**format**(**acc**))**

**return** acc

### Génération du Dataset

min\_length **=** 10

X\_train**,** y\_train **=** **[],** **[]**

X\_test**,** y\_test **=** **[],** **[]**

**for** i **in** range**(**2048**):**

x**,** y **=** reber**.**get\_one\_example**(**min\_length**)**

X\_train**.**append**(**x**)**

y\_train**.**append**(**generate**(**x**))**

**for** i **in** range**(**256**):**

x**,** y **=** reber**.**get\_one\_example**(**min\_length**)**

X\_test**.**append**(**x**)**

y\_test**.**append**(**generate**(**x**))**

X\_train **=** np**.**array**(**X\_train**)**

y\_train **=** np**.**array**(**y\_train**)**

X\_test **=** np**.**array**(**X\_test**)**

y\_test **=** np**.**array**(**y\_test**)**

maxlen **=** 20

X\_train **=** sequence**.**pad\_sequences**(**X\_train**,** maxlen**=**maxlen**,** padding**=**'post'**,** truncating**=**'post'**)**

y\_train **=** sequence**.**pad\_sequences**(**y\_train**,** maxlen**=**maxlen**,** padding**=**'post'**,** truncating**=**'post'**)**

X\_test **=** sequence**.**pad\_sequences**(**X\_test**,** maxlen**=**maxlen**,** padding**=**'post'**,** truncating**=**'post'**)**

y\_test **=** sequence**.**pad\_sequences**(**y\_test**,** maxlen**=**maxlen**,** padding**=**'post'**,** truncating**=**'post'**)**

### Creation du Model Many-to-Many

nb\_unit **=** 7

inp\_shape **=** **(**maxlen**,** 7**)**

loss\_ **=** "mean\_squared\_error"

metrics\_ **=** "mean\_squared\_error"

optimizer\_ **=** "Nadam"

nb\_epoch **=** 50

batch\_size **=** 64

model **=** Sequential**()**

model**.**add**(**LSTM**(** units**=**nb\_unit**,**

input\_shape**=**inp\_shape**,**

return\_sequences**=True))**

model**.**compile**(** loss**=**loss\_**,**

optimizer**=**optimizer\_**,**

metrics**=[**metrics\_**])**

checkpoint **=** ModelCheckpoint**(**"LSTM.h5"**,**

monitor**=**loss\_**,**

verbose**=**1**,**

save\_best\_only**=True,**

save\_weights\_only**=False,**

mode**=**'auto'**,**

period**=**1**)**

early **=** EarlyStopping**(** monitor**=**'loss'**,**

min\_delta**=**0**,**

patience**=**10**,**

verbose**=**1**,**

mode**=**'auto'**)**

history **=** model**.**fit**(**X\_train**,** y\_train**,**

validation\_data**=(**X\_test**,** y\_test**),**

epochs**=**nb\_epoch**,**

batch\_size**=**batch\_size**,**

verbose**=**1**,**

callbacks **=** **[**checkpoint**,** early**])**

### Creation du générateur

Model **=** load\_model**(**"LSTM.h5"**)** # lstm\_simple / srnn\_simple / gru\_simple

newModel **=** Sequential**()**

newModel**.**add**(**LSTM**(**units**=**7**,** stateful**=True,** batch\_input\_shape**=(**1**,**1**,**7**),** return\_sequences**=True))**

newModel**.**set\_weights**(**Model**.**get\_weights**())** # On recupère les données du training pour le générateur

nb\_samples **=** 20

result\_LSTM **=** **[]**

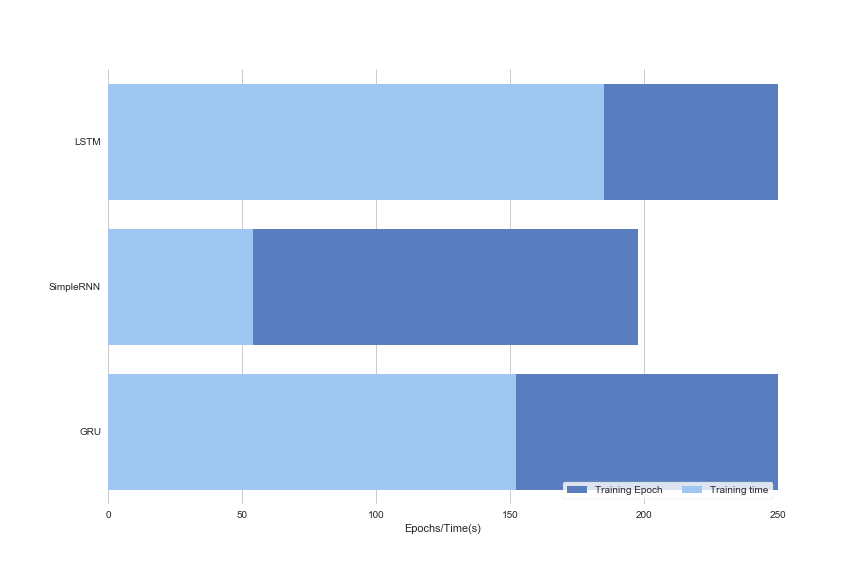
**for** \_ **in** range**(**nb\_samples**):**

result\_LSTM**.**append**(**evaluate**(**newModel**,** 100**,** 50**))**

## Résultat

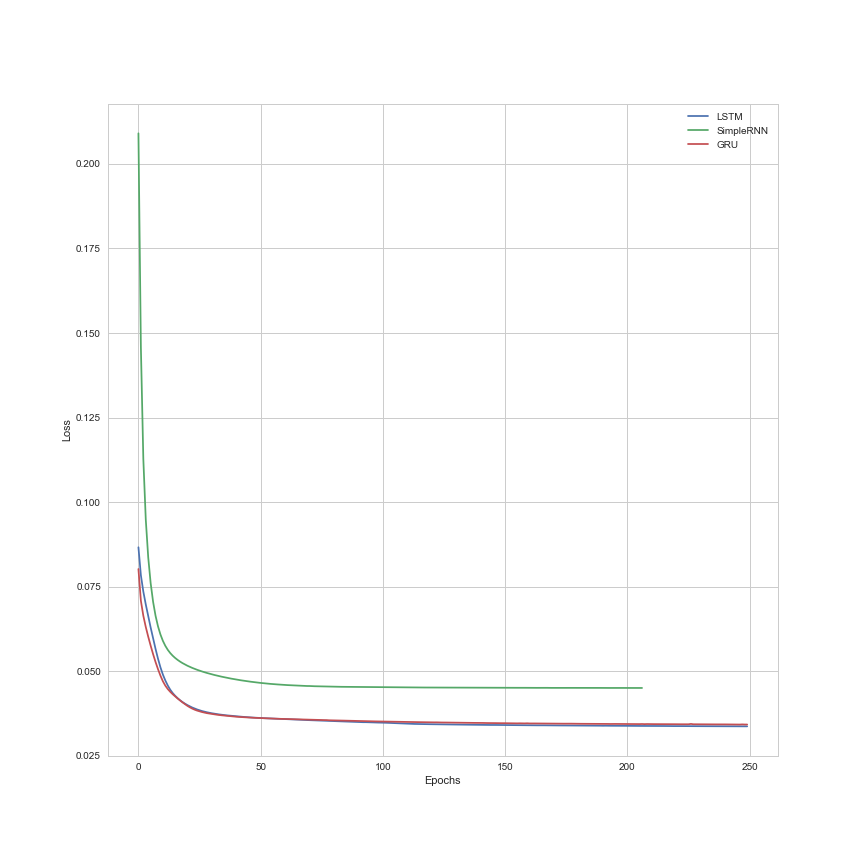
Sur le modèle précédent on peut donc essayer tous les modèles en remplacant juste la cellule LSTM par soit le GRU ou le SimpleRNN.

Dans un premier temps, on peut analyser la vitesse d'entrainement en nombre d'Epochs et de temps en secondes à paramètres fixe et on trouve :



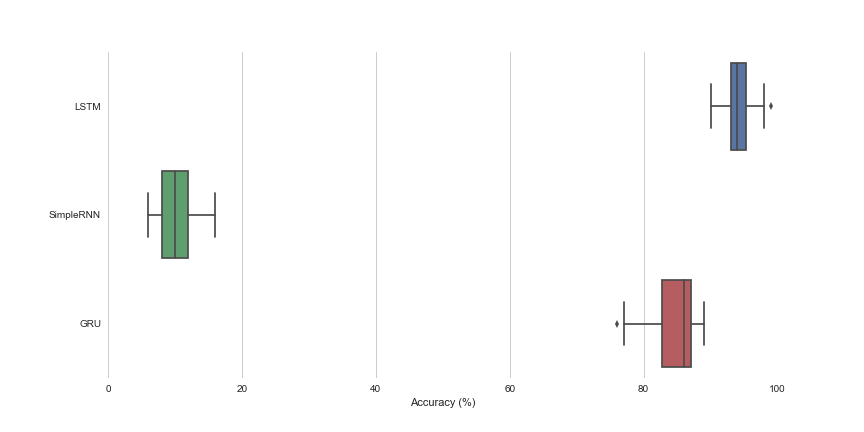
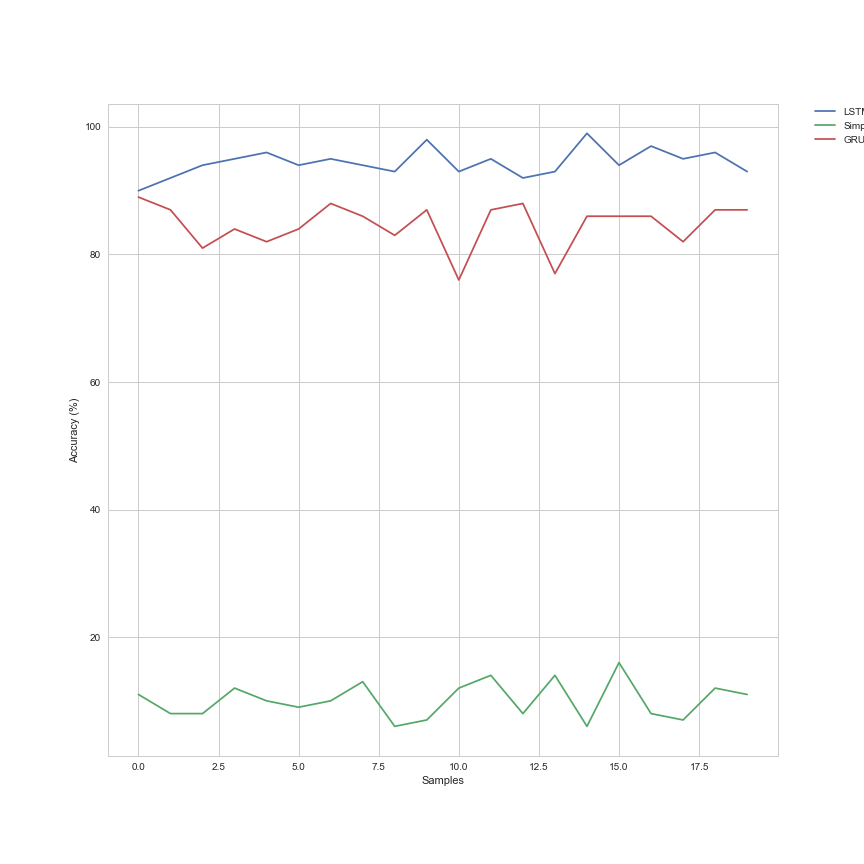
Que ce soit le LSTM ou le GRU, les 2 n'ont pas atteint le Early Stopping apres 250 Epochs. Quant au SimpleRNN, il a atteint le Eraly Stop apreès 198 Eopchs. SI on compare en terme de temps, on remarque que le SimpleRNN est aussi beaucoup plus rapide (53.97s pour 198 Epochs). Le LSTM et le GRU ayant tourné 250 Epoch on peut aussi les comparer temporellement et on remarque que le GRU est 18% plus rapide (LSTM = 185.01s et GRU = 152.41s). Le temps c'est bien, mais qu'en est-il de l'apprentissage ?

Pour cela on peut tracer le Loss via l'historique stocké durant le fit. On a donc :



On remarque déjà que le SimpleRNN a convergé à un Loss beaucoup plus élevé que le LSTM et GRU. Cela s'explique par le fait qu'il n'a pas de mémoire donc c'est plus difficile pour lui de "mémoriser" l'architecture des mots. Quand Au GRU, il apprend plus vite au depart que le LSTM mais converve avec légèrement plus de Loss (GRU : 0.0343 vs LSTM : 0.0337). Le training étant plus rapide pour le GRU, c'est juste une question de choix (mémoire vs rapidité).

On a maintenant nos 3 modèles entrainé, on peut donc les comparer au niveau qualité en tant que générateur. Cela peut se faire sur l'accuracy sur plusieurs essais de génération de 100 mots. On a :



On remarque déjà que le SimpleRNN est très mauvais sur cette tache (< 10% des mots valide) Quand au LSTM il dépasse très nettement le GRU malgré le peu de différence de Loss (95% pour le LSTM en moyenne vs 85% pour le GRU). Cela s'explique notamment par sa mémoire à long terme qui l'empêche de sauter d'un maillon a un autre du graphe comme expliqué au tout début. L'avantage du GRU est qu'il est plus rapide donc on peut avoir de topology plus profonde que des LSTM pour avoir un même temps de calcul. Dans ce cas, le résultat peut être identique (ou supérieur).

# Conclusion

Plus de 27 ans après les premiers modèles de Recurrent Neural Networks, La cellule LSTM a donné un regain de vigeur à ce type de modèle qui lui-même a permis une amélioration signification dans le monde du Machine Learning. Il a permis notament l'analyse de texte, la traduction ou même la reconnaissance vocale. Ce type de Reseaux de Neuronne évoluent très fréquemment parfois de manière infime (un changement de fonction d'activation dans la Cellule, une connexion etc et parfois de manière plus importante comme recemment avec le Quasi Recurrent Neural Network qui promet des entrainement beaucoup plus rapide donc aussi beaucoup plus complexe.

# Annexe

### Annexe 1 :

Principe de la backpropagation :

* <https://fr.wikipedia.org/wiki/R%C3%A9tropropagation_du_gradient>
* <https://fr.wikipedia.org/wiki/Gradient>
* <https://www.youtube.com/watch?v=q555kfIFUCM> (Chaine de Siraj Raval)

### Annexe 2 :

Il existe diverses fonctions d'activations en fonction de l'application. Certaines permette la convergence plus rapide du modèle, d'éviter le problème du *Vanishing/Exploding Gradient.*

Parmi les plus connues on peut lister :

|  |  |
| --- | --- |
| L'identité |  |
| La Marche |  |
| La sigmoïde |  |
| La tangente ou tangente Hyperbolique |  |
| Le ReLU (Rectified Linear Unit) |  |
| Le PReLU (Parametric Rectified Linear Unit) |  |
| Le ELU (Exponential Linear Unit) |  |
| Le swish activation (Nouveau – 10/2017) |  |

# Sources :

<https://en.wikipedia.org/wiki/Recurrent_neural_network>

<https://en.wikipedia.org/wiki/Long_short-term_memory>

<http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/>

<http://karpathy.github.io/2015/05/21/rnn-effectiveness/>

<http://www.asimovinstitute.org/neural-network-zoo/>

<https://deeplearning4j.org/lstm.html>

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

<https://arxiv.org/pdf/1412.3555v1.pdf>

<https://arxiv.org/abs/1611.01576>

<https://github.com/icoxfog417/tensorflow_qrnn>

https://theneuralperspective.com/2016/12/16/quasi-recurrent-neural-networks/

print(accuracy\_score(y\_eval, y\_pred\_floor))

print(confusion\_matrix(y\_eval, y\_pred\_floor))

print(precision\_score(y\_eval, y\_pred\_floor))

print(recall\_score(y\_eval, y\_pred\_floor))

print(f1\_score(y\_eval, y\_pred\_floor))

print(log\_loss(y\_eval, y\_pred))

0.96875

[[122 6]

[ 2 126]]

0.954545454545

0.984375

0.969230769231

0.0964980389554

0.7578125

[[100 28]

[ 34 94]]

0.770491803279

0.734375

0.752

0.48050892388

0.984375

[[125 3]

[ 1 127]]

0.976923076923

0.9921875

0.984496124031

0.0586241011113