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 *Artificial Neural Networks* utilisés en Régression/Classification non linéaires ou les *Convolutionnal Neural Networks* utilisés notamment pour le traitement d'images.

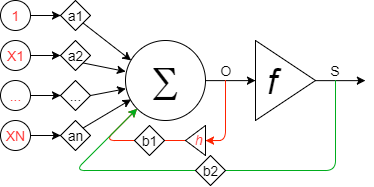
Cependant ces modèles ne permettent pas de travailler sur des données temporelles. Or, il existe un type de Neural Network qui permet de traiter des données temporelles (vidéos, sons, texte, …). Ce sont les ***Recurrent Neural Networks***. Ceux si sont ont été longtemps peu utilisés pour des raisons que l'on verra par la suite mais sont devenus très populaires récemment. Ils sont maintenant utilisés pour des *prédictions temporelles* (anticipation du cours d'une action, de retards), *la reconnaissance de paroles (Natural Langage Processing et Speech Recognition),* la traduction de texte ou encore l'analyse de sentiments.

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 phase de **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 initiale a été de boucler la sortie de chaque neurones sur une de ses entrées. La représentation sur un perceptron serait donc :



Selon les cas le bouclage a lieu avant ou apres la fonction d'activation.

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, le bouclage permet de passer à cette formule récurrente :

Cas de la récurence représenté en rouge :

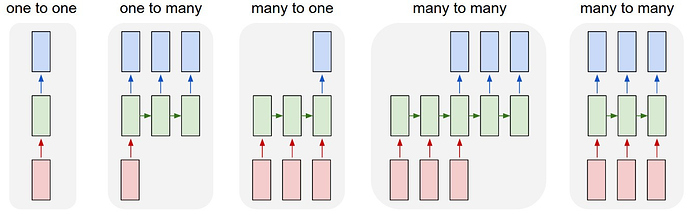
Cas de la récurrence représenté en vert :

Le cas de la récurrence en rouge permet d'avoir 2 fonctions d'activations (f et h) différentes et ainsi de pouvoir mieux ajuster le modèle.

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 Récurrent 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 :

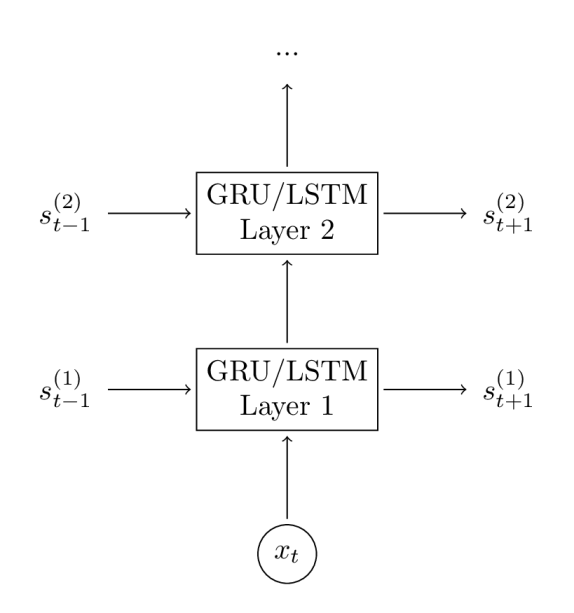
Ce modèle permet de faire des estimations sur des phénomènes "dynamiques". En effet, le RNN va prendre en compte plusieurs timesteps pour déterminer la suivante. Supposons le lancer d'une balle. Si on a en entrée la position x et y, le modèle One-to-One n'arrivera pas a prédire la prochaines position 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..

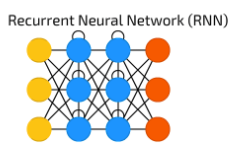
Stacking de Layers

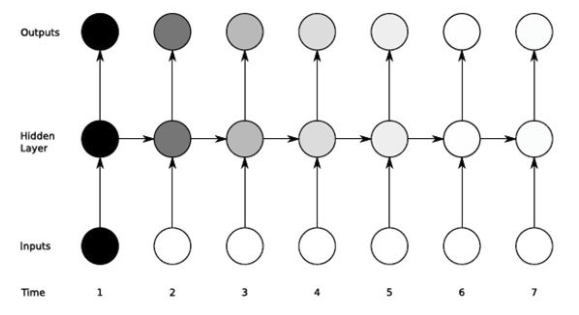
Comme tous réseaux de Neuronnes, on peut Stacker les cellules aussi en profondeur pour un apprentissage de règles plus complexes.



# 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 récursion. Ce modèle simple 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 aussi dans le temps. C'est ce que l'on appelle la *BackPropagation Throught Time*. Cette façon d'entrainer le réseau pose des plus gros problèmes de Vanishing gradient.



De ce fait, à delà de 30-40 timestep, il devient compliquer de l'entrainer correctement.

### En 1997

Il aura fallu attendre près de 30 ans pour que 2 scientifiques, Sepp Hochreiter and Jürgen Schmidhuber, publient un article sur une nouvelle façon de penser les Récurrent Neural Network. En plus de boucler la sortie sur l'entrée, ils ont mis en place un système de mémoire. La mémoire fonctionne à partir de 2 états (Cell State et Hidden State) ainsi que 2 portes (Input Gate et Output Gate).

La Cell State est la mémoire principale de de la cellule. C'est sa mémoire à long terme. Quant à la Hidden State, c'est la mémoire à court-terme. Elle ne mémorise l'information qu'entre 2 timesteps. Pour ce qui est des portes, la Input Gate a pour objectif de mettre à jour la Cell State en fonction de l'Input de la Hidden State. Quant à la Output Gate, elle va utiliser l'Input et la Cell State pour générer la prochaine Hidden State.

Lors du training, les portes sont donc entrainées de façon à filtrer correctement les informations utiles. C'est cette nouvelle architecture qui lui donne le nom de *Long Short-Term Memory*.

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 longue séquences, mais malheureusement encore limité (il n'est pas possible de l'utiliser pour générer de la musique). C'est cependant, très utilisé sur des texte pour faire de l'analyse de sentiments, faire des résumés, etc.

Il existe de nombreuses variantes au LSTM classique comme le peephole LSTM ou le Fast-Slow LSTM.

### En 2014

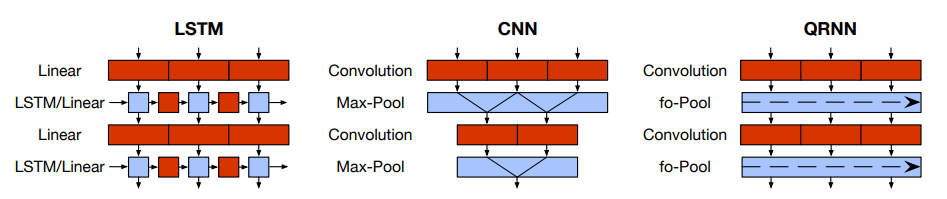
Depuis quelques années, un nouveau modèle a été mis en place**.** Le principe reste le même que le LSTM mais son architecture est différente. C'est la *Gated Recurrent Unit.* L'intérêt de cette cellule est d'être plus légère en termes de calcul pour une performance similaire. Le LSTM possède 3 gates (Input Gate, Output Gate and Forget Gate) et 2 états alors que le GRU n'a que 2 gates (Update Gate et Reset Gate). Comme leurs noms l'indique, la Update Gate met à jour les états et la Reset Gate permet d'oublier certains paramètres. Comme il n'y a pas de States, l'équivalent du Hidden State est la sortie du GRU.

### En 2016/2017

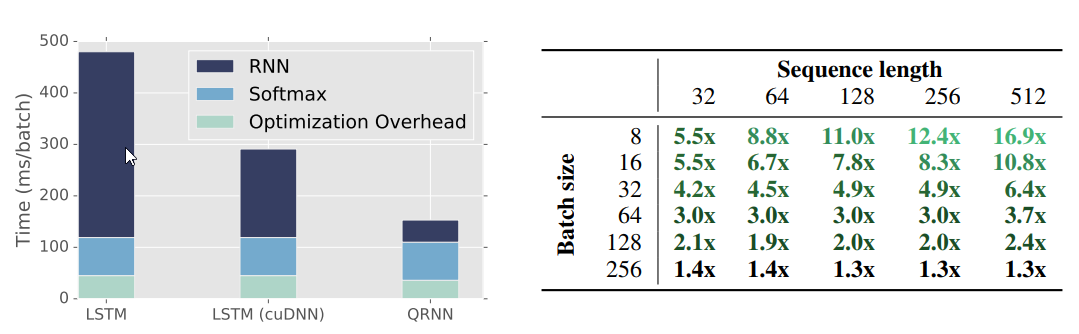
Le 5 Novembre 2016, un nouveau modèle a été publié. Il fait table rase du système de mémoire et s'inspire des layers de convolution. Son intérêt est de permettre un entrainement beaucoup plus rapide car il permet désormais de travailler en parallèle.

L'entrainement des RNN actuelles sont entrainés sur toutes les timesteps de chaque batch. De ce fait, il faut attendre la fin de chaque timestep pour passer au suivant. Une fois toutes les timesteps parcourues, un nouveau batch est utilisé et ce pendant n-epochs. Dans le cas des QRNN, l'entrainement est fait sur tout le dataset pour chaque timesteps. De ce fait, le calcul sur chaque timestep peut être parallélisé. Chaque timestep est entrainé sur un Layer de Convolution et entre chaque timestep, un layer appelé fo-Pool fait un travail similaire au Max-pooling Layer des CNNs.

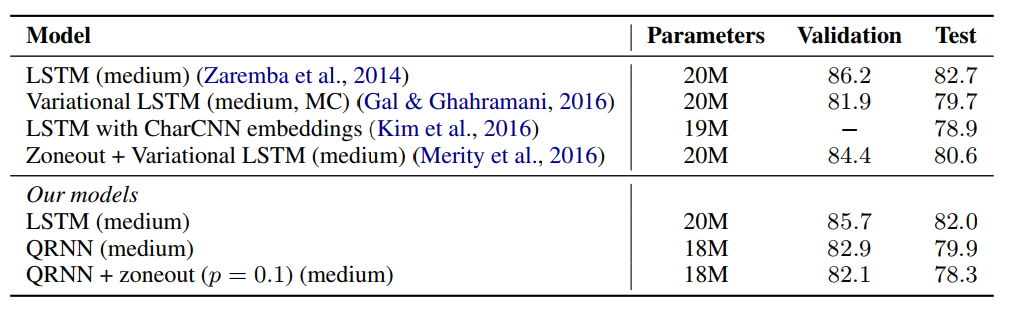
Dans le graphe ci-dessous, on peut voir un diagramme représentant les diagrammes pour les LSTM/GRU, CNN et QRNN.



Si l'on regarde au niveau de ces diagrammes, seul le LSTM à une partie séquentielle (blocs discontinus). C'est cette phase qui limite le parallélisme. Ce modèle étant récent, il n'a pas encore été implémenté et bien étudié mais les performances en terme de durée d'entrainement est très important pour les longue séquences (jusqu’à 17 fois).



Si l'on regarde au niveau des performances d'apprentissage il est légèrement en dessous des LSTM mais celui-ci va surement évoluer dans les mois/années à venir comme ce fût le cas avec le LSTM.

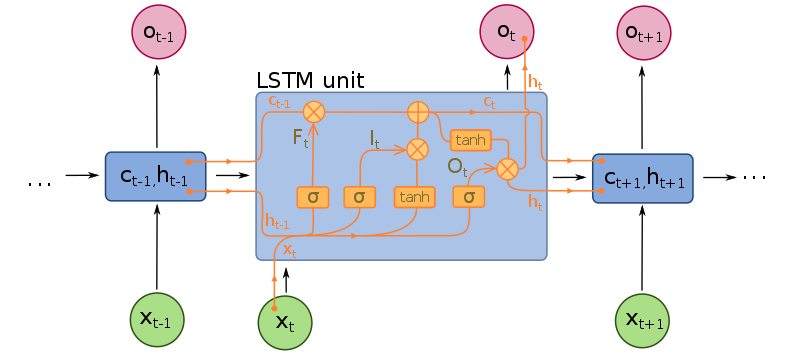


# Les différents Cellules

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

### LTSM

La cellule de LSTM peut se représenter linéairement comme suit :



On remarque dans un 1er temps que l'on a en entrée la sortie prédite précédemment (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. La Cell State est aussi transmise par la précédente cellule.

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 "apprendre" quels features sont 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.

Le 2nd flux va sur la forget gate qui est composé de 2 perceptrons 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 être 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 résultat seront aussi multiplié element-wise et ajouté à la Cell State. A ce stade la Cell State vaut :

De ce fait, si la sortie est négative, la Cell State va "perdre" de l'information.

Après cette étage, la Celle State est prête à être transférée à la prochaine cellule mais pour avoir la sortie de cette timestep et le Hidden State pour l'état suivant, il faut passer par la forget gate .

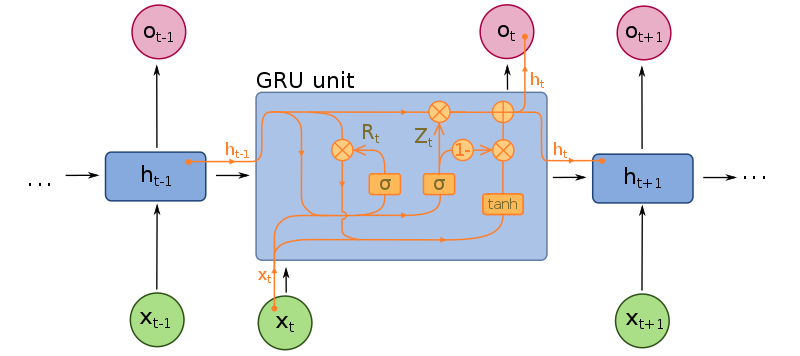
Celle-ci est aussi composée d'un perceptron basé sur les entrées :

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 objectif 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 où le Hidden State sert de mémoire à court terme, le GRU l'utilise comme mémoire à long terme aussi.

La cellule se compose donc de 3 "perceptrons". Les 2 premiers "perceptrons" sont utilisés pour générer une Update Gate (Zt) et la Reset Gate (Rt).

Les équations des 2 gates sont donc :

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

La sortie de la Reset State va ensuite passer dans le 3eme perceptron afin d'extraire les informations utiles sur l'input de l'état actuel. Cette sortie est ensuite multipliée 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 :

# Evaluation / Comparaison

Comme nous l'avions vu dans l'introduction, les Recurrent Neural Networks sont très utilisés dans de l'analyse de texte. Un dataset bien connu pour évaluer ces modèles est le dataset IMDB (aussi appelé Large Movie Review Dataset). Celui-ci regroupe 25000 commentaires avec un label (sentiment positif ou négatif) préparés en amont. Si vous desirez avoir plus d'info sur la préparation des données, vous pouvez vous référez au rapport publié par l'université de Stanford en 2011 qui a préparé le dataset (http://ai.stanford.edu/~amaas/papers/wvSent\_acl2011.pdf)

Dans cette partie, nous allons comparer les performances de ces 4 modèles sur ce dataset. Attention, l'objectif n'est pas d'atteindre la meilleur performance en classification mais bel et bien comparer à topologie identique, les performances des modèles. Pour ce faire la topologie sera la même pour tous les modèles avec uniquement la variation de la cellule du RNN (SimpleRNN/LSTM/GRU ou QRNN).

Le modèle a été mis en place en suivant cette très bonne présentation (<https://machinelearningmastery.com/predict-sentiment-movie-reviews-using-deep-learning/>). La différence est que la classification va se faire uniquement avec une cellule RNN et non un Feed Forward Neural Neztwork ou Convolutionnal Neural Network.

Le code partiel est donc :

max\_features **=** 20000 # On ne garde que les 20 000 mots les plus courants

maxlen **=** 256 # On ne garde que 256 mots (=256 timesteps)

batch\_size **=** 32

epochs **=** 25

padding\_mode **=** "pre" # Si le commentaire est < 256 mots, on complete en amont de 0

truncating\_mode **=** "post" # Si le commentaire est > 256 mots, on tronque en aval

# Chargement du dataset et padding

**(**X\_train**,** y\_train**),** **(**X\_test**,** y\_test**)** **=** imdb**.**load\_data**(**num\_words**=**max\_features**)**

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

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

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

model**.**add**(**Embedding**(**max\_features**,** 128**))** # On transpose les 20000 mots dans 128 dimensions

model**.**add**(**SimpleRNN**(**128**))** # Seulement cette cellule change

model**.**add**(**Dense**(**1**))** # Pour la classification

model**.**add**(**Activation**(**'sigmoid'**))**

model**.**compile**(**loss**=**'binary\_crossentropy'**,** # On est sur de la classification

optimizer**=**'adam'**,**

metrics**=[**'accuracy'**])**

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

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

verbose**=**0**,** callbacks**=[**early**,** time\_callback**])**

Après avoir fait tourné toutes les cellules dans ce modèle, on peut regarder l'évolution du loss en fonction des Epochs.

On peut aussi regarder pour chaque modèle la précision en training et test en fonction des Epochs

Pour finir, on peut regarder la précision en fonction du temps

Maintenant, si on regarde le temps par epoch, on remarque que

D'après le rapport sur les QRNN, notre modèle utilise un batch de 32 et 256 timesteps, le QRNN devrait donc être 4.9x plus rapide que le LSTM. Si on regarde, on est seulement … fois plus rapide.

# 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 notamment 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>

https://wiseodd.github.io/techblog/2016/08/12/lstm-backprop/

<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/>

<http://ai.stanford.edu/~amaas/papers/wvSent_acl2011.pdf>

https://machinelearningmastery.com/predict-sentiment-movie-reviews-using-deep-learning/

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