Cours OpenClassrooms

Les Récurrents Neurals Networks

Contenu

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*** *(appelés FFNN par la suite)* utilisés majoritairement dans des problèmes en Régression/Classification non linéaires ou les ***Convolutionnal Neural Networks*** *(appelés CNN par la suite)* utilisés notamment pour le traitement d'images.

Cependant ces modèles ne permettent pas de travailler sur des données temporelles (vidéos, sons, texte, …). Dans ce type de problème, la donnée à un instant T va dépendre des sorties précédentes. Il existe un type de Neural Network qui permet de traiter des données temporelles : ce sont les ***Recurrent Neural Networks*** *(appelés RNN par la suite)*. 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, prédiction 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 et 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. Lors de cette phase, une prédiction va être faite en fonction des entrées. En fonction de cette prédiction, 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. De plus amples information sur la Back Propagation est disponible en Annexe 1.

Cependant avec les Récurrent Neural Network, la backpropagation ne se fait uniquement au travers des layers mais aussi dans le temps. C'est la Back Propagation Through Time. Cela pose un souci beaucoup plus important de Vanishing Gradient. En cas de problème assez complexe, on se retrouve à stacker en profondeur et dans le temps les cellules rendant l'entrainement presque impossible.

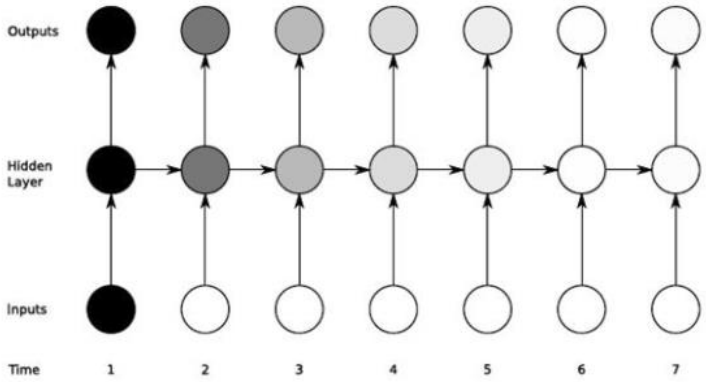


Figure 1 – Représentation dépliée d'un RNN - Problème de Vanishing Gradient en BPTT

Utilisation des Récurrent Neural Networks

Il existe différentes façon d'arranger un RNN en fonction du problème que l'on veut résoudre. On peut retrouver ci-dessous les différents types :

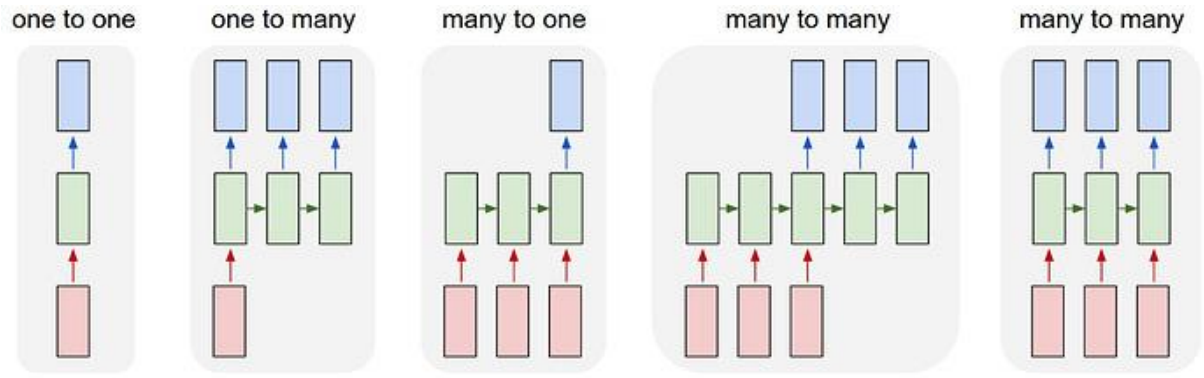


Figure 2 - Les différentes architectures de RNN

Le rectangle vert représente la cellule du RNN, le rouge représente les Entrées (l'entrée peut être un scalaire ou un vecteur) et en bleu les Sorties (scalaire ou vecteur). La représentation verticale représente les timesteps (c'est la représentation dite "déroulée ou "unfolded")

Le One-to-One:

C'est l'utilisation la plus basique d'un RNN. En effet, celui-ci va prédire la sortie suivante basée sur une entrée sans considération des timesteps (représentés horizontalement). Cela peut servir à classer par exemple des images. Son utilisation est donc très rare car on perd l'intérêt de la récurrence.

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 une phrase en fonction d'un input qui serait par exemple le 1er mot.

Le Many-to-One :

Cette architecture est utilisée en Classification. Notamment pour l'analyse de sentiments. En fonction d'une phrase de longueur N (ce sont les timesteps), le modèle prédit si elle est positive ou négative.

Le Many-to-Many :

Représentation 1:

Cette topologie est très utilisée pour la traduction. En entrée, une phrase est fournie, lorsque celle-ci est terminée, le modèle donne la traduction.

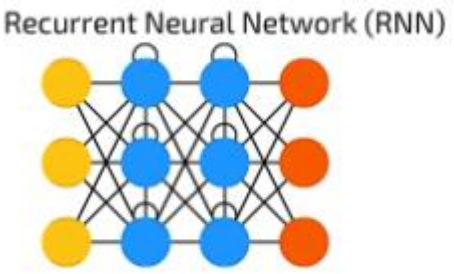
Représentation 2:

Ce type de modèle peut être utilisé pour faire par exemple de la classification de vidéos. A chaque image (et en fonction des images précédentes), le modèle fournit des classes pour chaque frame.

Etat de l'art

Dans les années 1970 – Simple RNN

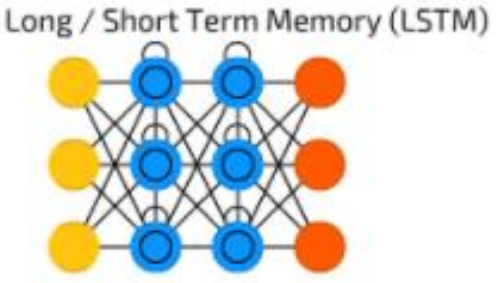
L'idée initiale pour mettre en place un RNN était de boucler la sortie d'un layer sur son entrée pour créer une récursion. Ce modèle simple est celui qui a été proposé dans les années 70. Ils font partie de la famille des Fully Recurrent Neural Networks. Plutôt simples, ils ont le bénéfice d'être rapides au niveau du calcul. En contrepartie, ils ont le très gros défaut d'être difficiles à entrainer sur des données complexes ou avec beaucoup de timesteps. (Une présentation plus approfondie est présente en Annexe 4)



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

En 1997 - LSTM

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).



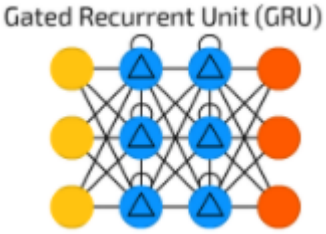
La Cell State est la mémoire principale de de la cellule. C'est sa mémoire à long terme. Celle-ci va évoluer durant les diverses timesteps et sera transmises tout au long des timesteps. Quant à la Hidden State, c'est la mémoire à court-terme (le bouclage). Elle ne fait que transmettre l'information pour la timestep suivante. 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 avec une mémoire à courts et longs termes qui lui donne le nom de ***Long Short-Term Memory*** *(appelés LSTM par la suite)*.

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 d'oublier certaines informations devenues inutiles et a permis à ce type de modèle de se démocratiser. En effet, ce modèle est très performant sur de plus 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 textes pour faire de l'analyse de sentiments, faire des résumés, de la traduction, ...

Il existe de nombreuses variantes au LSTM classique comme le peephole LSTM, le multi-array LSTM ou le Fast-Slow LSTM. Une explication plus approfondie du LSTM de base est fournie en Annexe 5.

En 2014 - GRU



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*** *(appelés GRU par la suite)*. L'intérêt de cette cellule est d'être plus légère en termes de calculs 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) et un seul état (Hidden State). Comme leurs noms l'indiquent, la Update Gate met à jour la Hidden State et la Reset Gate permet d'oublier certaines informations. Une présentation plus en détail est présente en Annexe 6.

En 2016/2017 -QRNN

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 RNNs actuelles est effectué en série sur toutes les timesteps de chaque batch avant de passer au batch suivant. De ce fait, l'entrainement est fait en Série. 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 fait en parallèle. 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.

On peut représenter les 2 processus d'entrainement comme suit :

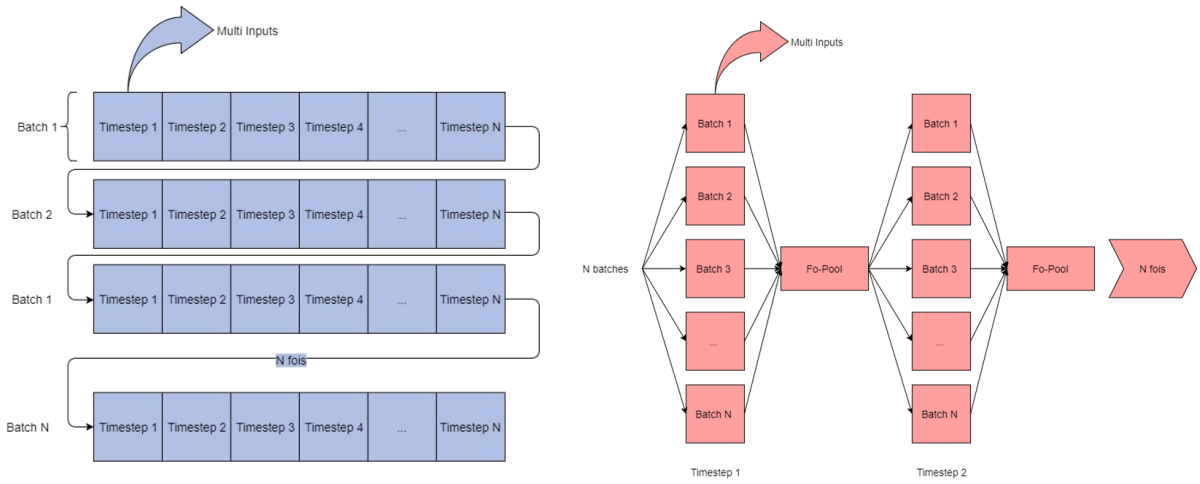
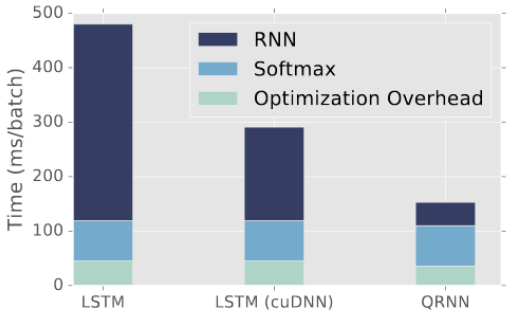


Figure 3 - Processus d'entrainement d'un RNN Standard à gauche et QRNN à droite

Ce modèle étant récent, il n'a pas encore été implémenté et bien étudié mais le gain en terme de durée d'entrainement est important pour les longue séquences (jusqu'à 17 fois) comme le montre le benchmark suivant :



		Sequence length				
		32	64	128	256	512
Batch size	8	**5.5x**	**8.8x**	**11.0x**	**12.4x**	**16.9x**
	16	**5.5x**	**6.7x**	**7.8x**	**8.3x**	**10.8x**
	32	**4.2x**	**4.5x**	**4.9x**	**4.9x**	**6.4x**
	64	**3.0x**	**3.0x**	**3.0x**	**3.0x**	**3.7x**
	128	**2.1x**	**1.9x**	**2.0x**	**2.0x**	**2.4x**
	256	**1.4x**	**1.4x**	**1.3x**	**1.3x**	**1.3x**

Figure 4 - Comparaison du temps d'entrainement entre LSTM et QRNN

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 ou années à venir comme ce fût le cas avec le LSTM.

Model	Parameters	Validation	Test
LSTM (medium) (Zaremba et al., 2014)	20M	86.2	82.7
Variational LSTM (medium, MC) (Gal & Ghahramani, 2016)	20M	81.9	79.7
LSTM with CharCNN embeddings (Kim et al., 2016)	19M	–	78.9
Zoneout + Variational LSTM (medium) (Merity et al., 2016)	20M	84.4	80.6
Our models			
LSTM (medium)	20M	85.7	82.0
QRNN (medium)	18M	82.9	79.9
QRNN + zoneout ($p = 0.1$) (medium)	18M	82.1	78.3

Figure 5 - Performance des modèles en Classification

Evaluation / Comparaison

Comme nous l'avions vu dans l'introduction, les RNN 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 sentiment attaché (positif ou négatif) préparés en amont. La présentation du dataset est fournie en Annexe 3.

Dans cette partie, nous allons comparer les performances de ces 4 modèles sur ce dataset. L'objectif n'est pas d'atteindre la meilleure 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).

Etant dans un problème de classification, on va utiliser une structure Many-to-One. 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 FFNN ou CNN.

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 = 1
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'])

model.fit(X_train, y_train, batch_size=batch_size, epochs=epochs,
                    validation_data=(X_test, y_test),
                    verbose=0, callbacks=[time_callback])

# time_callback est un Callback fait pour récupérer les info durant le training (temps,
loss, accuracy)
```

L'entrainement ne s'est fait que sur 1 Epoch car selon les modèle, l'overfitting était très rapide (cf. Annexe 7).

Après l'entrainement de tous les modèles, on peut donc regarder l'évolution du Loss et de la Précision en fonction des batchs :

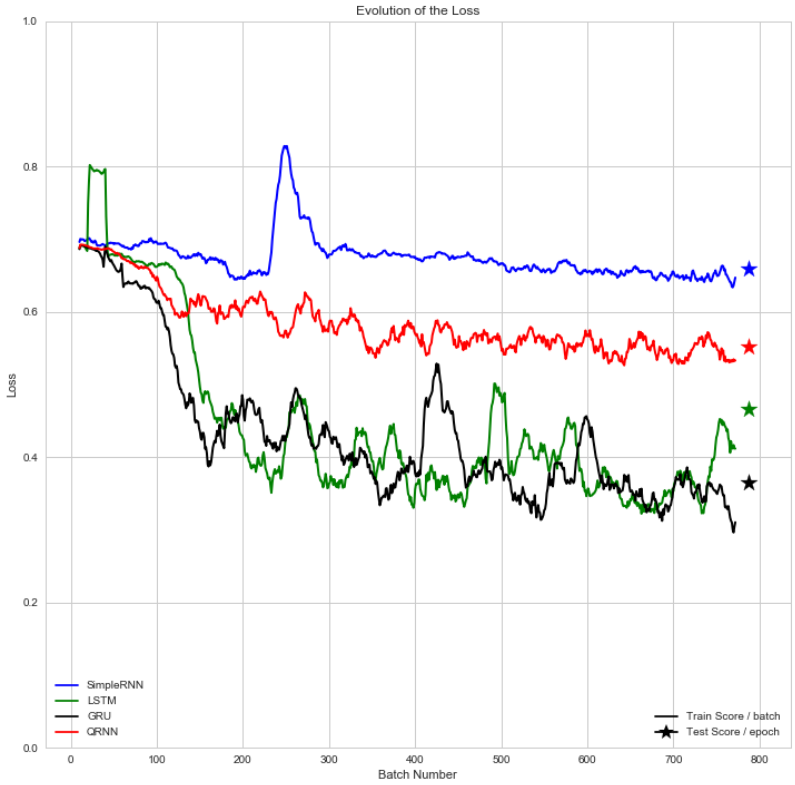


Figure 6 - Evolution du Loss pendant la 1ère Epoch

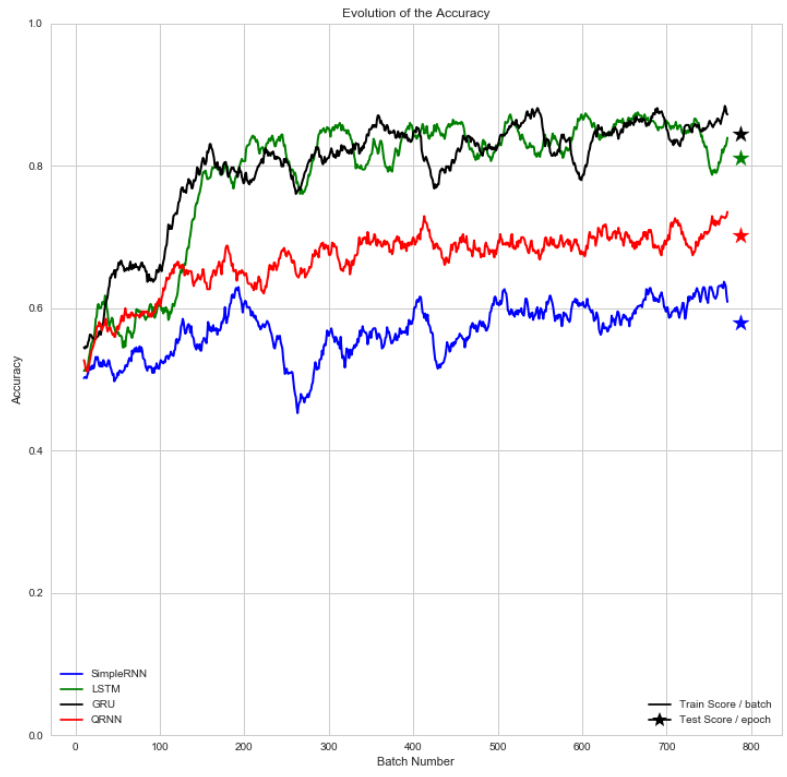


Figure 7 - Evolution de la précision pendant la 1ère Epoch

On remarque que l'apprentissage du QRNN n'est pas aussi bon que le LSTM/GRU actuellement. Ce modèle n'est qu'à son balbutiement et va surement évoluer comme ce fût le cas avec le LSTM. De plus, il est possible d'ajouter des régularisations sur celui-ci (non pris en compte pour cet entrainement afin de ne pas le ralentir) comme une L^2 régularisation, un gradient Clipping ou du dropout. Ce que l'on remarque aussi c'est qu'au début du training, le QRNN apprend plus vite que le LSTM mais légèrement moins vite que le GRU. Par contre son apprentissage s'arrête aussi plus tôt.

Au vu du loss, il était prévisible d'avoir un entrainement moins bon que les LSTM/GRU. Cependant, l'intérêt de ce modèle réside dans sa vitesse d'entrainement. Si l'on regarde maintenant l'évolution de la précision en fonction du temps d'entrainement on trouve :

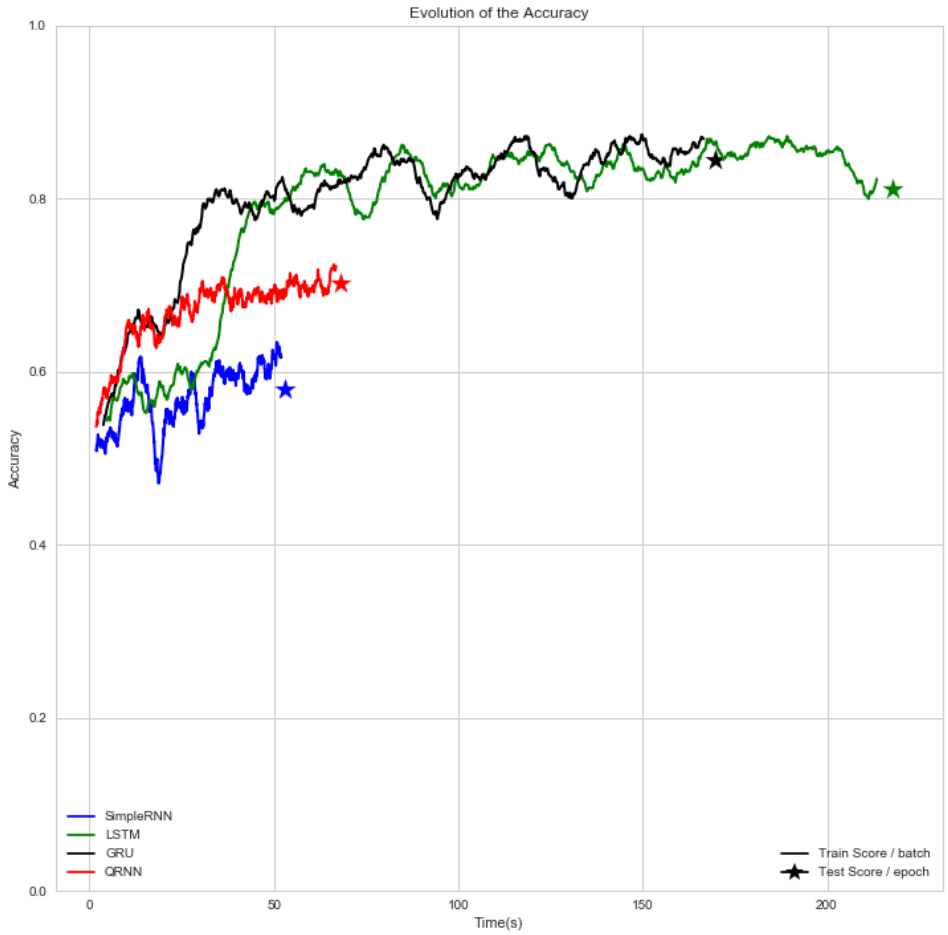


Figure 8 - Evolution de la précision en fonction du temps d'entrainement

On remarque une très forte amélioration du temps d'entrainement. En effet, le QRNN est presque aussi rapide que le Simple RNN mais son apprentissage est meilleur. La partie d'apprentissage est aussi rapide que le GRU au départ. On remarque aussi que le LSTM met toujours un peu de temps à commencer l'apprentissage car il a plus de paramètres à ajuster.

Pour finir, on peut essayer d'évaluer nous aussi les performances du QRNN en fonction du Batch Size et Sequence Length comme annoncé dans le document publié en Novembre 2016. Le heatmap de gauche est le gain de performance annoncé et celui de gauche, celui mesuré.

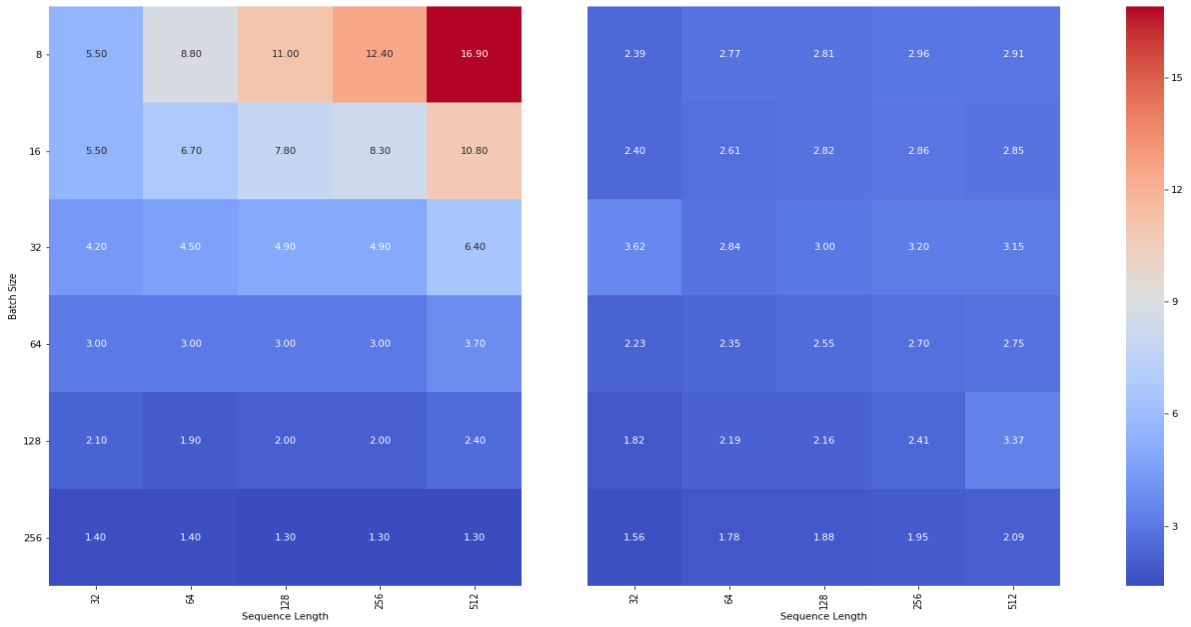


Figure 9 - Evaluation du gain de temps d'entrainement entre LSTM et QRNN. Comparaison avec les performances annoncées

On peut remarquer que l'on a un gain en terme de temps d'entrainement de l'ordre de 2 à 3 mais pas le gain annoncé (jusqu'à 17 fois). Dans le papier, les essais étaient faits avec plus de calculs (régularisation L^2, dropout1D, gradient Clipping) et pour autant les performances sont meilleures. Cela s'explique car les performances calculées sont faites sans prendre en compte le temps du Softmax et d'optimisation (cf. figure 5). Sur ce même graphe, si on prend le temps total, on a un gain d'environ 3 au lieu de 17. De plus, le modèle de QRNN n'est pas officiel. Il a été publié sur Github et n'est peut-être pas complètement optimisé.

Conclusion

Plus de 27 ans après les premiers modèles de Recurrent Neural Networks, la cellule LSTM a donné un regain de vigueur à ce type de modèle qui lui-même a permis une amélioration significative dans le monde du Machine Learning. Il a permis notamment l'analyse de texte, la traduction ou même la reconnaissance vocale. Ce type de Réseaux de Neurones évolue assez fréquemment. Parfois l'évolution est assez mineure comme avec des variations du LSTM et parfois de manière plus importantes comme lors de la sortie du LSTM, du passage de 2 à 3 gates sur le LSTM, de la sortie du GRU et depuis peu avec l'idée émergente du Quasi Recurrent Neural Network qui promet des entrainements beaucoup plus rapide.

Durant ce projet, nous avons pu tester ces modèles et ainsi voir le bénéfice que ceux-ci apportent. Le QRNN va surement encore évoluer dans les années à venir notamment sur sa capacité à apprendre, et qui sait, peut-être détrôner le LSTM.

Quant au problème du Vanishing gradient, l'implémentation du Synthetic gradient (arxiv 07/2017) pourrait être aussi une alternative prometteuse.

Annexe

Annexe 1 - Backpropagation:

Vous trouverez ci-dessous quelques liens parlant du 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 Youtube de Siraj Raval)

Annexe 2 – Différentes fonctions d'activations :

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

Parmi les plus connues, on peut lister :

L'identité	$y = x$
La Marche / Fonction de Heaviside	$y = \begin{cases} 0 \; pour \; x \leq 0 \\ 1 \; pour \; x > 0 \end{cases}$
La sigmoïde	$y = \frac{1}{1 + e^{-x}}$
La tangente ou tangente Hyperbolique	$y = \tan(x) \;\; ou \; y = \tanh(x)$
Le ReLU (Rectified Linear Unit)	$y = \begin{cases} 0 \; pour \; x \leq 0 \\ x \; pour \; x > 0 \end{cases}$
Le PReLU (Parametric Rectified Linear Unit)	$y = \begin{cases} \alpha x \; pour \; x \leq 0 \\ x \quad pour \; x > 0 \end{cases} \quad \alpha < 1$
Le ELU (Exponential Linear Unit)	$y = \begin{cases} \alpha(e^x - 1) \; pour \; x \leq 0 \\ x \qquad\qquad pour \; x > 0 \end{cases} \quad \alpha < 1$
Le swish activation (Nouveau – 10/2017)	$y = x * sigmoid(x) = \frac{x}{1 + e^{-x}}$

Annexe 3 – Préparation du dataset

La préparation du dataset a été faite en 2011 par l'université de Stanford dont vous trouverez le papier original à cette adresse : http://ai.stanford.edu/~amaas/papers/wvSent_acl2011.pdf.

Annexe 4 – Equation du Simple RNN

Le principe du Simple RNN est de boucler la sortie sur l'entrée. De ce fait, on peut représenter la cellule comme suit :

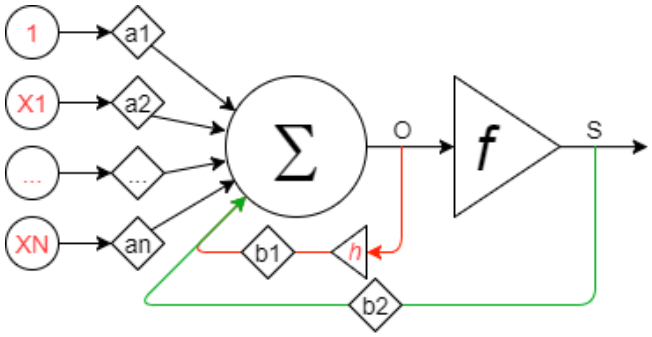


Figure 10 - Représentation du SimpleRNN

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

Sur un FFNN, la sortie d'un layer peut se représenter comme :

$$S = f(\sum_{inputs} a_i * x_i)$$

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 ces formules récurrentes :

Cas de la récurrence représentée en rouge :

$$S(t) = f\left(\sum_{inputs} a_i * x_i(t) + h\left(\sum_{inputs} a_i * x_i(t-1)\right)\right)$$

Cas de la récurrence représentée en vert :

$$S(t) = f\left(\sum_{inputs} a_i * x_i + b2 * (S(t-1))\right)$$

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.

Annexe 5 – Equation du LSTM

La cellule de LSTM peut se représenter linéairement comme suit :

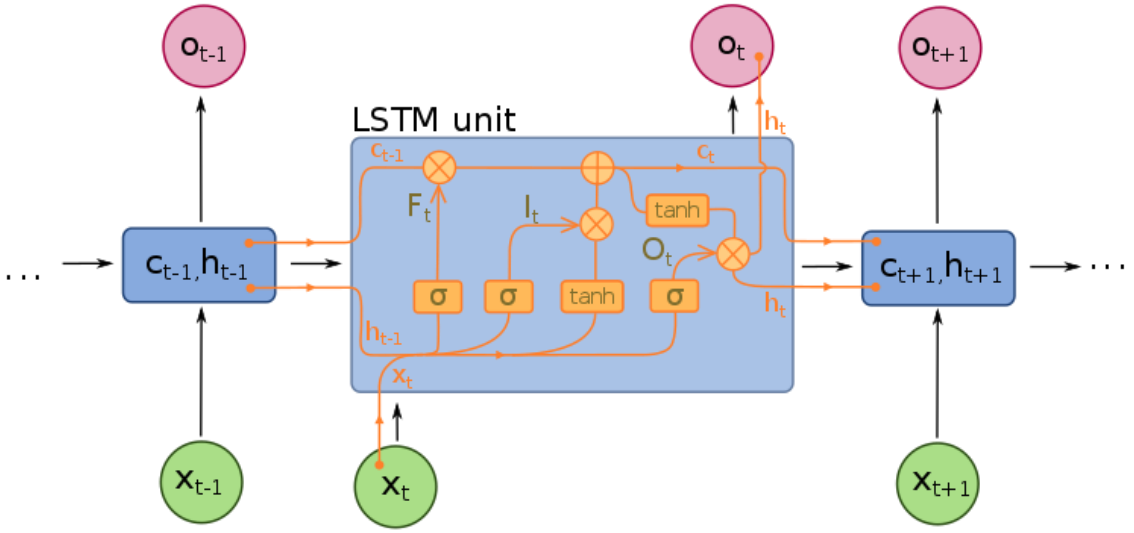


Figure 11 - Représentation du LSTM

Dans un premier temps, 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 de C_{t-1} à C_t pour la transmettre à la prochaine timestep.

Le premier flux va sur l'Input Gate. Cette porte, comme toutes les portes est en fait un perceptron. Lors du training, ses poids seront ajustés pour "apprendre" quels features sont utiles. La fonction d'activation est la sigmoïde. Pour chaque input, la gate fournira en sortie une valeur. Plus la valeur sera grande, plus son intérêt de la garder sera élevé. La fonction est donc :

$$F_t = \sigma(W_t[h_{t-1}, X_t] + b_t)$$

Ce résultat est ensuite multiplié (element-wise) par l'actuel Cell State.

$$C_t = f_t \times C_{t-1}$$

Le second flux va sur la Forget Gate qui est composé de 2 perceptrons mais avec des poids différents. Le premier 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 :

$$f_c = tanh(W_c[h_{t-1}, X_t] + b_c)$$

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

$$I_t = \sigma(W_i[h_{t-1}, X_t] + b_i)$$

Ces 2 résultats seront aussi multipliés element-wise et ajouté à la Cell State. A ce stade la Cell State vaut :

$$C_t = f_t \times C_{t-1} + f_c \times i_t$$

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

Après cette étape, la Celle State est prête à être transférée à la prochaine cellule mais pour avoir la sortie de cette timestep (le Hidden State), il faut passer par la Output Gate.

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

$$o_t = \sigma(W_o[h_{t-1}, X_t] + b_o)$$

et aussi la Cell State pour pondérer l'importance des inputs (on utilise donc la mémoire en parallèle des entrée pour prédire la sortie). L'équation de sortie est donc :

$$h_t = o_t \times \tanh(C_t)$$

Le modèle de base d'un LSTM se compose donc de 4 "perceptrons" 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 limite très nettement les calculs et évite la perte de gradient.

Annexe 6 – Equation du GRU

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

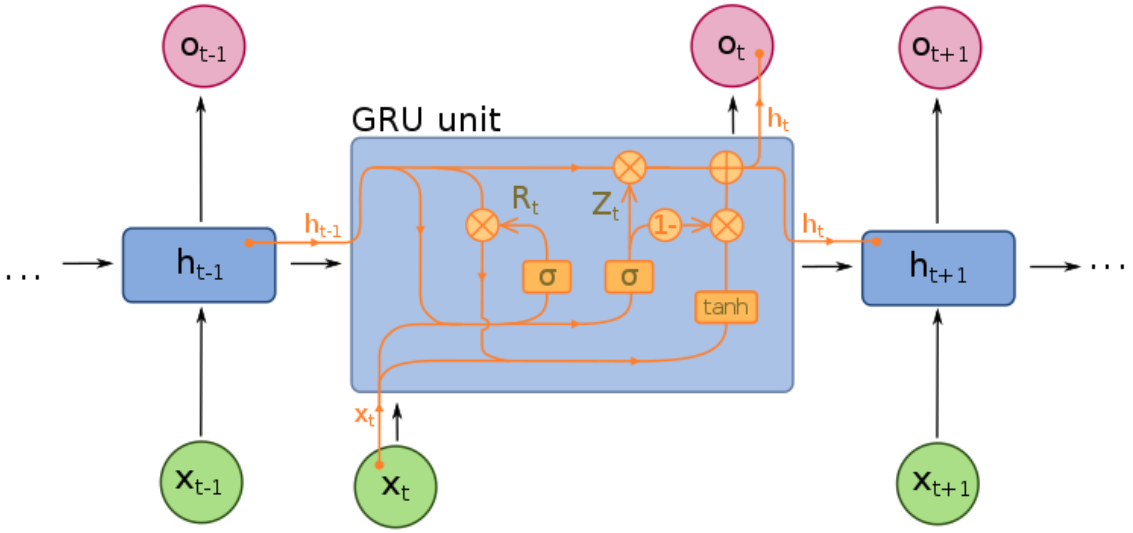


Figure 12 - Représentation du GRU

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 (Z_t) et la Reset Gate (R_t).

Les équations des 2 gates sont donc :

$$Z_t = \sigma(W_z * h_{t-1} + U_z * X_t + b_z)$$

$$R_t = \sigma(W_r * h_{t-1} + U_r * X_t + b_r)$$

Les sorties 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 pour fournir le prochain Hidden State qui sera propagé.

L'équation finale est donc :

$$h_t = z_t * h_{t-1} + (1 - z_t) * tanh(W_h * x_t + U_z * (r_t * h_{t-1}) + b_h)$$

Annexe 7 – Overfitting du training

Initialement, le training a été fait en enregistrant le Loss du Training Set et Test Set. L'entrainement était lent (1h 30 pour le LSTM). Cependant, l'overtiffiting a tendance à apparaitre entre la première et deuxième Epoch comme le montre le graphe suivant.

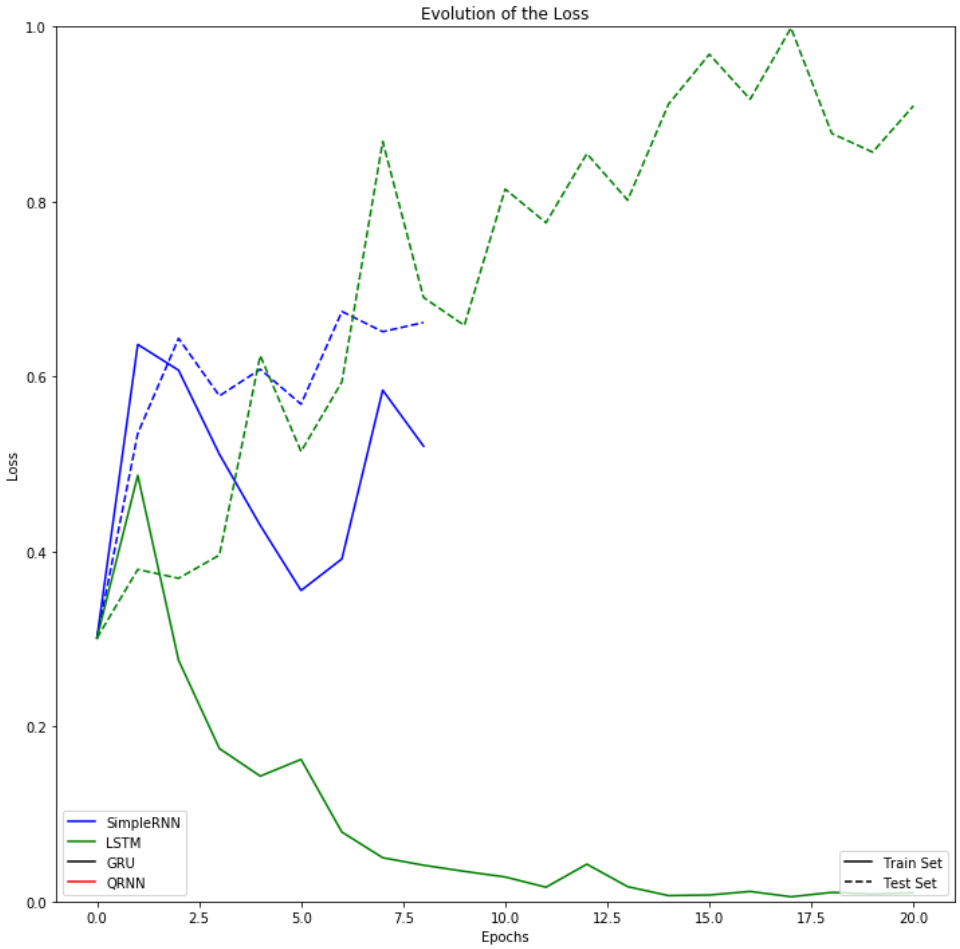


Figure 13 - Evolution du Loss par Epoch - Visualisation de l'overfitting

Si l'on regarde au niveau du LSTM (courbe verte), dès l'Epoch 2, le loss du Train Set (ligne continue) descend alors que le loss en Test augmente fortement. De ce fait, pour la partie d'évaluation, la limitation à 1 Epoch a été ajoutée.

Sources :

Recurrent Neural Networks

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

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

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://slazebni.cs.illinois.edu/spring17/lec03_rnn.pdf

https://commons.wikimedia.org/wiki/Artificial_neural_network

LSTM

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

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

https://deeplearning4j.org/lstm.html

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

GRU

https://en.wikipedia.org/wiki/Gated_recurrent_unit

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

http://www.vincent-gripon.com/files/conf/2017-Cognitive-3.pdf

QRNN

https://arxiv.org/abs/1611.01576

https://github.com/icoxfog417/tensorflow_qrnn

https://github.com/DingKe/nn_playground/tree/master/qrnn

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

DATASET

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

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

Table des illustrations