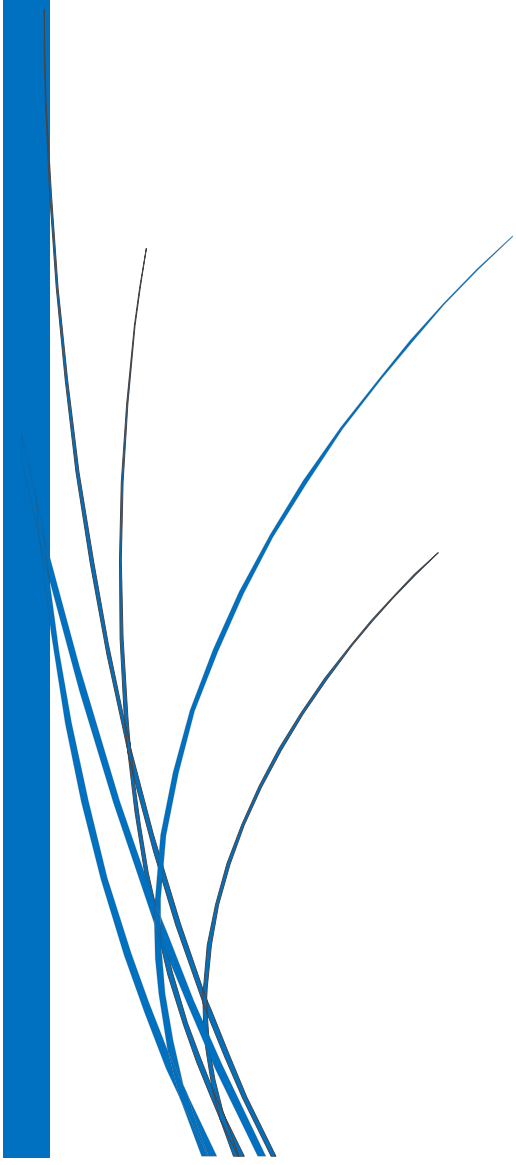
19/12/2019

Générer un discours présidentiel



DJERROUD Tekfa

Table des matières

1) Introduction

Ces jours-ci, dans mon pays, nous nous retrouvons en pleine crise politique. Nous sommes dans une période d'élections et les discours politiques, comme partout dans le monde, sont d'une médiocrité inqualifiable.

L'idée du projet et de pouvoir faire une génération de discours politiques. Pour générer ces textes, nous allons utiliser des réseaux de neurones. Ce sont les algorithmes utilisés derrière la majorité des prouesses récentes en Intelligence Artificielle : description automatique d'image, reconnaissance vocale…

Plus précisément, dans le riche panel des réseaux de neurones, nous avons utilisé des architectures récurrentes, abrégées RNN (Recurrent Neural Networks). Cette classe de réseaux de neurones travaille à partir de données séquentielles brutes, et apprend des motifs dans la succession des états présentés. Les RNN sont aux données séquentielles brutes ce que les réseaux convolutifs sont aux images brutes.

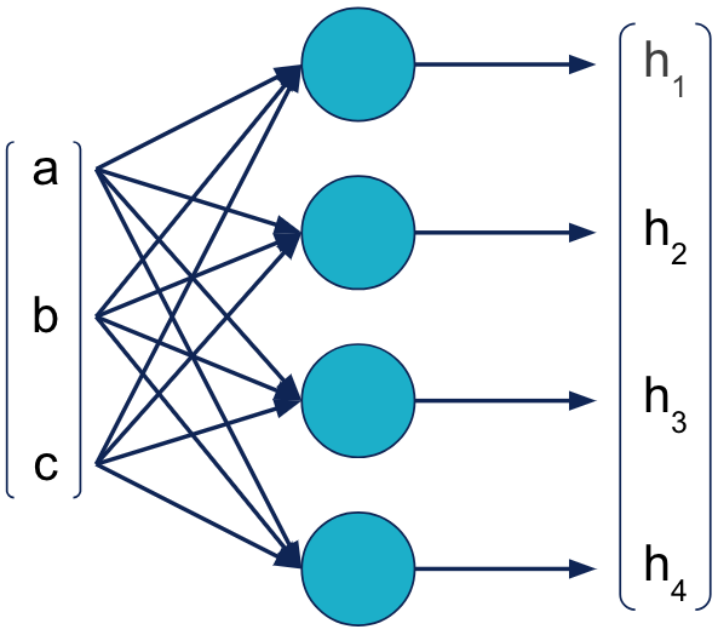
Nous utilisant donc les RNN en travaillant sur du texte. Ces modèles sont à la pointe pour capter les dynamiques dans les suites de lettres ou de mots. Ils sont par exemple utilisés en traduction automatique. Nous avons repris un cas d'usage classique : la génération de texte. Nous nous sommes fortement appuyés sur le ce célèbre article, d'Andrej Karphathy qui écrit des pièces de Shakespeare. Dans ce projet nous avons opter pour la génération des discours politiques en français.

2) Les RNN : une architecture adaptée aux séquences

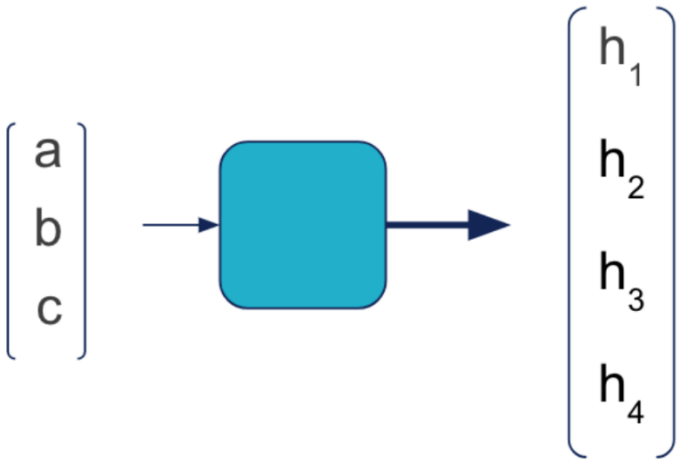
a) D'une simple couche de MLP pour des données statiques…

Les MLP sont des algorithmes de machine learning supervisé. Ils prennent la description d'un objet en entrée, et fournissent une prédiction en sortie. L'entrée est représentée par un vecteur numérique, qui décrit les caractéristiques (*features*) de l'objet. Ce vecteur traverse une succession de **couches de neurones**, où chaque neurone est une unité de calcul élémentaire. La prédiction est fournie en sortie sous la forme d'un vecteur numérique.

Dans le schéma suivant, une couche de MLP reçoit en entrée un vecteur [a, b, c], et produit en sortie un vecteur [h1, h2, h3, h4].



De façon simplifiée, nous pouvons représenter une couche entière sous la forme d'une cellule. Le schéma précédent devient alors :



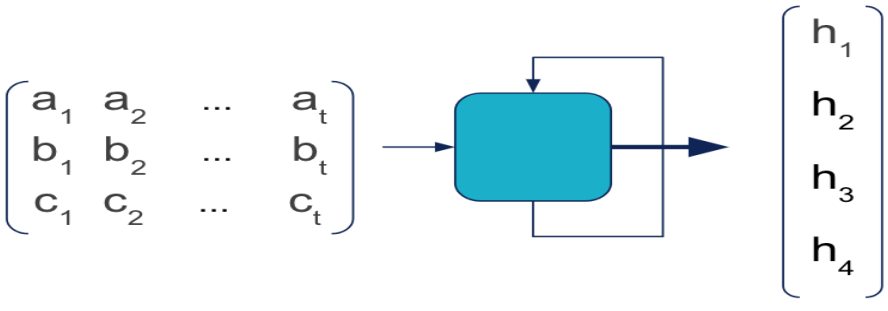
b) Vers des couches récurrentes pour les données séquentielles

Supposons maintenant que nous travaillions sur un objet dynamique. Nous souhaitions prédire le futur à partir de l'histoire récente. Les variables d'entrée [a, b, c] ne sont plus des valeurs uniques mais des séquences. Par exemple, la variable a est une suite chronologique de n valeurs (a1, a2, …, an).

Comment un réseau de neurones peut-il traiter ce genre d'entrées ?

Une façon naïve de procéder serait de mettre à plat l'ensemble des valeurs dans un unique vecteur (a1, …, an, b1, …, bn, c1, …, cn), et de les fournir en entrée de la première couche d'un MLP. Cependant les MLP ne font aucune distinction dans l'ordre des valeurs d'une couche. Cette méthode équivaut ainsi à mélanger toutes les valeurs : la dynamique temporelle serait annihilée, la relation entre les valeurs successives prises par a, b et c serait perdue. De plus le nombre de paramètres à apprendre sur la première couche augmenterait proportionnellement avec la longueur des séquences. En bref, la performance ne serait pas au rendez-vous.

C'est là que les réseaux récurrents (RNN) entrent en jeu ! Un réseau récurrent est un type particulier de réseaux de neurones, particulièrement adapté aux données séquentielles. Une couche récurrente est représentée de cette façon :



On reconnait le schéma d'une couche MLP, à 2 différences près :

L'entrée est une séquence de vecteurs. Chacun correspond à un incrément de temps, ou time step.

Une flèche en boucle apparait sur la cellule. En pratique, le RNN parcourt successivement les entrées [ai, bi, ci], et calcule des sorties intermédiaires [h1,i, h2,i, h3,i, h4,i]. Pour ce faire, il utilise non seulement l'entrée courante, mais également la sortie calculée précédemment [h1,i-1, h2,i-1, h3,i-1, h4,i-1]. C'est la signification de la boucle, qui symbolise le caractère récurrent du RNN.

Pour résumer, les RNN s'utilisent comme des MLP classiques. La différence essentielle est qu'on leur fournit une séquence de vecteurs en entrée et non un vecteur unique.

3) Nos données d'apprentissage.

Pour entraîner un modèle de machine learning, il nous faut tout d'abord des données d'apprentissage. Le site vie-publique.fr rassemble plus de 130.000 discours prononcés par des acteurs de la vie politique, depuis 1959. Ces textes sont une formidable source de travail, que l'on peut parcourir via un moteur de recherche. Ils ne font malheureusement pas partie des données téléchargeables directement sur data.gouv.fr. Notre premier travail consistait donc à parcourir automatiquement le site internet pour les télécharger.

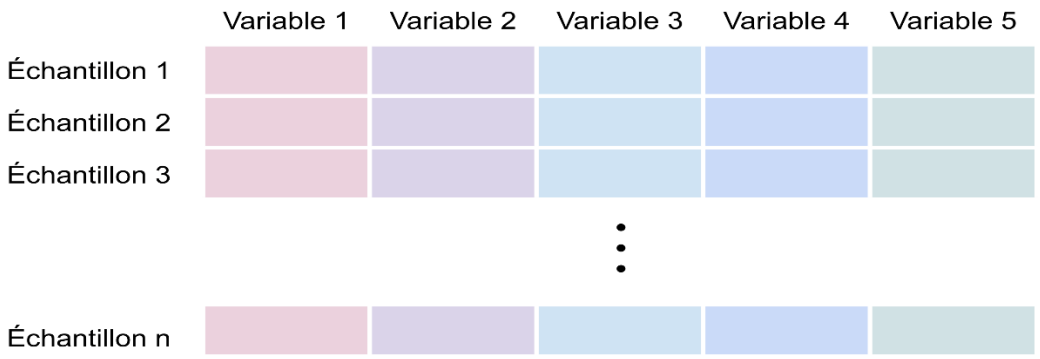
Le format de sortie de ce script est une liste de discours au format JSON, avec des informations contextuelles : titre, orateur, fonction et contexte. On obtient au final 784 millions de caractères, qui représentent 800 Mo de texte. Nous n'avons pas besoin de tant de texte pour apprendre à générer des discours politiques. Nous en extrayons donc un échantillon, en se restreignant d'abord aux discours de candidats dans le cadre d'élections, puis en sélectionnant 1,5 millions de caractères.


```
{
    "titre": "Déclaration de politique générale de M. Jacques Chirac...",
    "discours": "Le 2 avril dernier, Georges Pompidou est mort, laissant au monde l'exemple admirable de son sacrifice au service
    de la France. Pendant plus de dix ans....",
    "personne": "CHIRAC Jacques. ",
    "fonction": "FRANCE. Premier ministre",
    "contexte": "Déclaration de politique générale de M. Jacques Chirac, Premier ministre..."
}
```

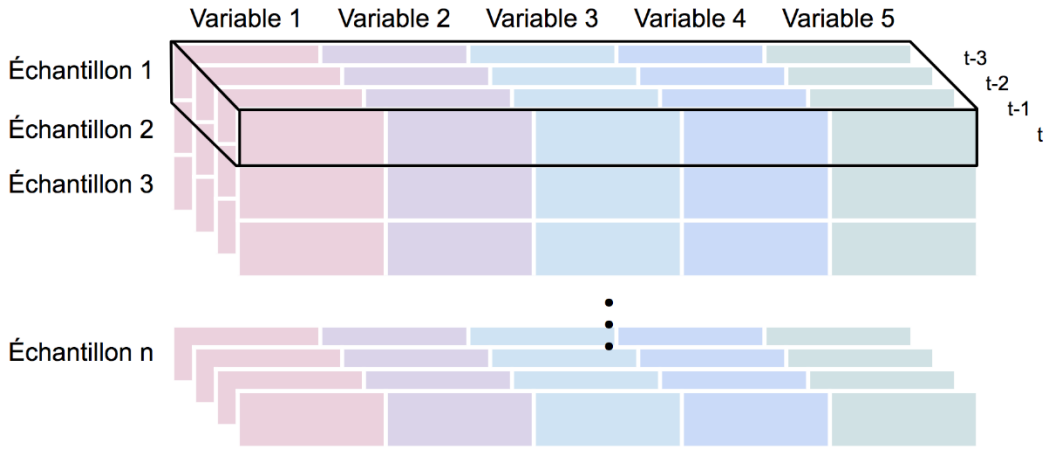
4) Préparer les données

Nous allons entraîner un RNN à prédire le prochain caractère d'un discours connaissant les n derniers. Nous pourrons alors écrire des textes caractère par caractère, en prédisant successivement le prochain caractère à partir de ceux générés auparavant. Nous pourrons ainsi tester la capacité du système à recréer des mots, puis à les assembler dans des structures grammaticales cohérentes.

En général, un algorithme de machine learning classique prend en entrée une matrice à deux dimensions : les lignes représentent les échantillons (samples) et les colonnes sont les variables (features) décrivant chaque échantillon.



En plus de ces deux dimensions, un réseau de neurones récurrent attend en entrée une troisième dimension qui est la dimension temporelle. Nos données d'entrée doivent donc avoir les dimensions (nombre d'échantillons, longueur des séquences, nombre de variables).



Nous allons procéder en 3 étapes pour transformer le texte brut vers ce format attendu :

a) Réduction de l'alphabet

Notre donnée textuelle brute contient 195 caractères différents : majuscules, minuscules, accents, caractères spéciaux en tous genres… Nous commençons par nettoyer le texte, pour se ramener à des mots en lettres minuscules, séparés par des espaces, des apostrophes, des points ou des virgules.

a A à á b B c C ç é … ÿ z Z ⟶ a b c d … z

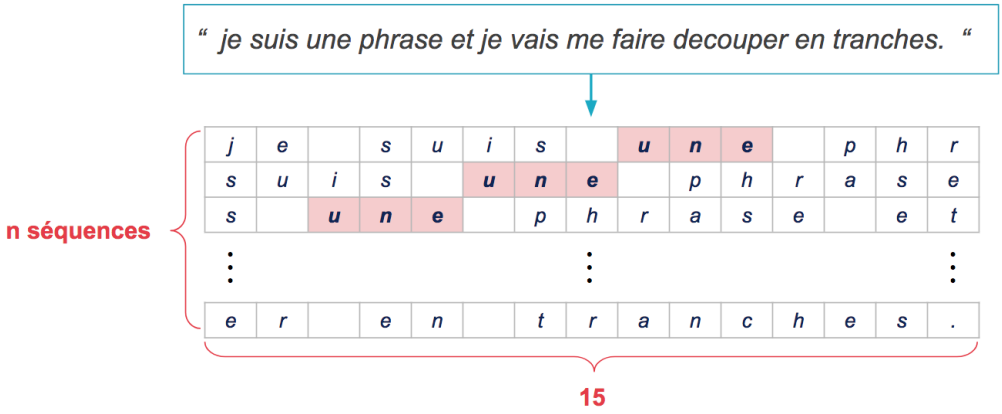
? ! - “ : ; < > … # $ \n \t ⟶ (espace) , . ’

La taille de l'alphabet est ainsi réduite à 30. Cette réduction simplifie le problème, ce qui nous permet de consommer moins de texte et de capacités de calcul pour l'entraînement

b) Découpage du texte en séquences

Pour prédire le prochain caractère, nous devons fournir au RNN une séquence des caractères précédents. Afin d'obtenir de nombreux exemples d'apprentissages, nous découpons des séquences dans les discours, en coulissant d'un certain pas entre chaque tranche.

Voici par exemple comment produire des séquences de longueur 15 en coulissant d'un pas de 3 d'une séquence à l'autre :



La longueur des séquences est libre, de même que le pas. Ce sont des paramètres qu'il faut ajuster jusqu'à obtenir un modèle satisfaisant.

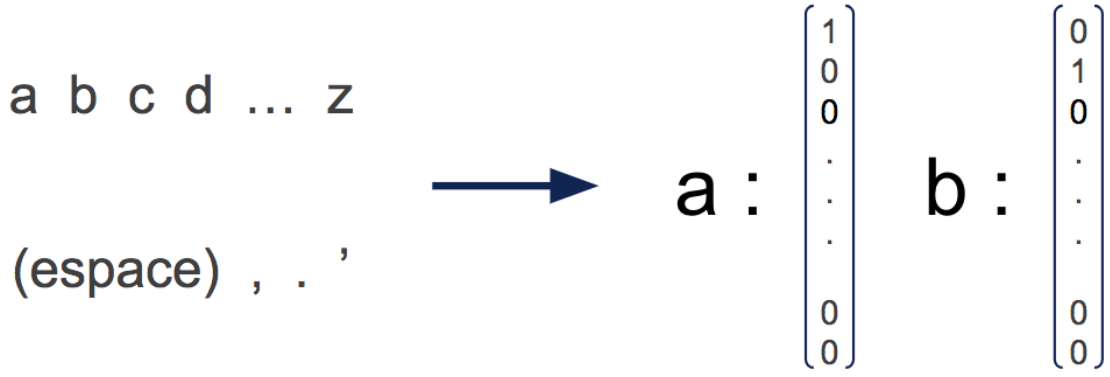
En pratique, des séquences de 15 caractères contiennent peu de contexte. Nous avons choisi des séquences de taille 50 afin que l'historique recouvre plusieurs mots. La deuxième dimension de notre matrice d'entrée est donc 50.

Le pas détermine le nombre d'échantillons, c'est à dire la première dimension de la matrice d'entrée. Il doit être choisi comme un équilibre entre le nombre d'exemples généré et la diversité des séquences produites.

À l'extrême, un pas de 1 génère toutes les séquences possibles. Cela augmentera le temps d'entraînement, mais sans forcément améliorer les performances. En effet, 2 séquences consécutives n'auront que 2 caractères différents, et le réseau verra beaucoup d'exemples redondants. Pour notre modèle nous avons choisi un pas de 3 comme représenté sur la figure.

c) Encodage « one-hot » des caractères

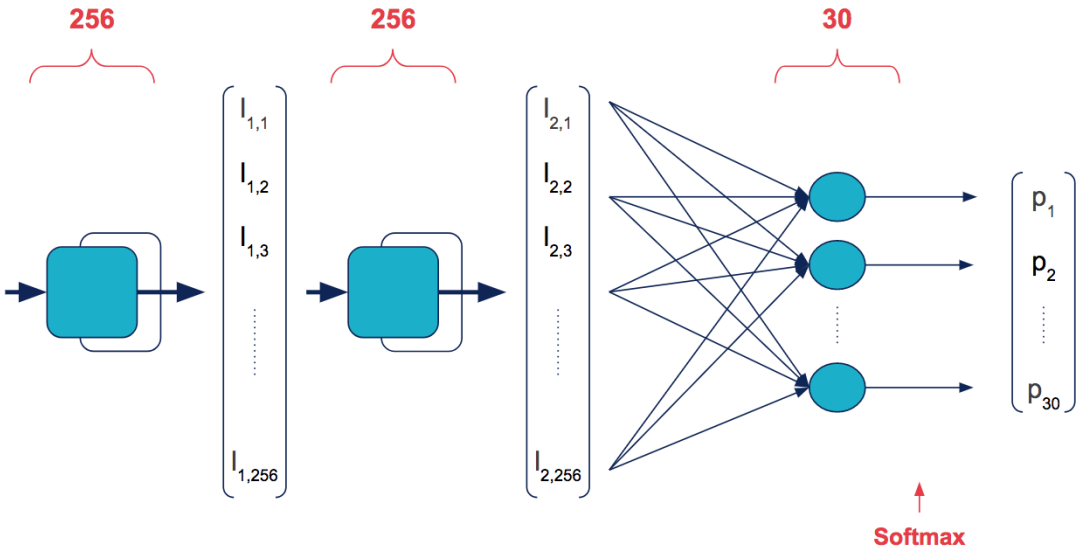
Une fois les séquences de caractères découpées, il faut les encoder dans des vecteurs numériques. On utilise pour cela l'encodage one-hot, où chaque caractère est représenté par un vecteur valant 1 sur une des composantes, et 0 sur les autres. C'est pour cela que la taille de l'alphabet correspond à la troisième dimension de notre matrice "nombre de variables".



5) Construire une architecture de réseau récurrent

Le modèle que nous avons utilisé est un réseau récurrent formé d'une succession de deux couches RNN et d'une couche MLP. La succession de plusieurs couches de RNN augmente la "profondeur" du réseau, et permet d'extraire des relations d'autant plus complexes (moins linéaires) entre les entrées. On utilise ici un type particulier de RNN, les Long Short-Term Memory (LSTM). La dernière couche MLP est de taille 30, de façon à ce que chaque neurone corresponde à un caractère à prédire. On normalise la prédiction via

une fonction softmax pour obtenir une distribution de probabilité.



Un tel modèle s'implémente en quelques lignes avec Keras. On définit un modèle séquentiel, sur lequel on ajoute une succession de couches qui correspondent directement aux étapes du schéma précédent. Les couches intermédiaires de dropout sont une technique de régularisation, pour éviter au modèle de trop coller aux données vues en entraînement et améliorer sa généralisation.

```
from keras.models import Sequential
from keras.layers import LSTM, Dense, Dropout, Activation

SEQ_LEN = 50
NB_CHARS = 30

model = Sequential()
model.add(LSTM(256, input_shape=(SEQ_LEN, NB_CHARS), return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(256))
model.add(Dropout(0.2))
model.add(Dense(NB_CHARS))
model.add(Activation('softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

6) Entraîner un RNN à prédire le prochain caractère

Notre objectif est de prédire, à partir d'une séquence de 50 caractères consécutifs, le caractère suivant. Il s'agit d'un problème d'apprentissage supervisé : à chaque itération, on fournit au modèle une séquence d'entrée encodée ainsi que le caractère encodé attendu en sortie. Le modèle effectue une prédiction, la compare à la cible attendue, et ajuste ses paramètres (aussi appelés poids du réseau) en cas d'erreur. Nous avons préparé nos données et créé notre modèle, il ne reste maintenant plus qu'à lancer l'apprentissage. Une ligne suffit pour démarrer cette étape.

```
# INPUTS : texte d'entraînement prétraité (coupé en séquences et encodé)
# TARGETS : cibles d'entraînement (prédiction attendue pour chaque séquence)
# NB_ITER : nombre d'itérations d'entraînement (passages sur l'ensemble des données)

model.fit(INPUTS, TARGETS, epochs=NB_ITER)
```

Plusieurs paramètres peuvent influencer la qualité de l'entraînement.

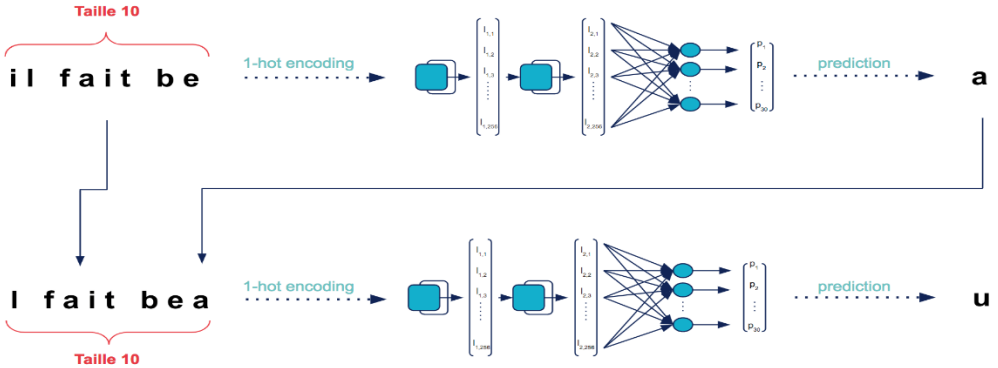
- Le nombre d'itérations définit le nombre de passes d'entraînement sur l'ensemble des données. Il a un impact direct sur la durée de l'entraînement. Avec plus d'itération, le modèle apprend mieux la relation entre la séquence d'entrée et la sortie à prédire. Avec trop d'itérations, il y a cependant le risque de sur-apprendre des spécificités des données d'entraînement qui ne se généralisent pas.

- La taille de batch définit la fréquence d'ajustement des poids du réseau. Avec un entraînement par batch, les poids ne sont pas ajustés pour chaque séquence d'entrée, mais pour chaque paquet de séquences de taille fixée. L'utilisation de batchs diminue le nombre de mise à jour et accélère l'apprentissage. Il améliore également la convergence, car on ajuste les poids à partir d'une erreur moyenne sur tout le batch et non d'erreurs individuelles plus volatiles.

7) Générer un discours caractère par caractère

Une fois le modèle entraîné, nous pouvons passer à la génération de texte proprement dite. Notre modèle a été construit pour prédire un caractère à la fois.

Pour générer un texte entier il suffit de donner au modèle une phrase de départ de la bonne taille et de lui demander de prédire un caractère. Ce caractère prédit est ensuite concaténé à la phrase initiale et la première lettre de celle-ci est supprimée. On se retrouve alors avec une nouvelle séquence de la même taille que la première, avec laquelle on peut à nouveau alimenter le modèle. En itérant autant de fois que nécessaire, on obtient finalement un texte de la longueur voulue.



Le schéma ci-dessus illustre ce processus de génération de texte, à partir d'une séquence de taille 10. En réalité, notre modèle étant entraîné sur des séquences de longueur 50, il faut respecter cette contrainte et lui donner une phrase de départ de taille 50.

Si l'on souhaite générer un paragraphe de longueur 500 à partir d'une phrase initiale donnée par l'utilisateur, il suffit d'écrire :

```
# INDEX_CHAR : dictionnaire { indice : caractère associé}

first_s = "mes chers compatriotes, nous abordons en ce moment"
generated = first_s
for i in range(500):
    preds = predict_single_input(model, first_s)
    next_index = sample(preds, temperature, do_sample)
    next_char = INDEX_CHAR[next_index]
    generated += next_char
    first_s = first_s[1:] + next_char

print(generated)
```

Une subtilité importante se cache dans la fonction sample. La sortie du modèle est une distribution de probabilité sur les 30 caractères. Pour choisir le prochain, on pourrait prendre celui de probabilité maximale. Cependant ce processus serait déterministe : un modèle et une séquence initiale fixes produiront toujours le même texte. Afin d'amener de la diversité, on réalise un tirage aléatoire dans la distribution de probabilité

8) Résultats obtenus

Notre générateur de texte n'est pas un parfait écrivain. Les phrases produites n'ont pas toujours un sens, et les textes ne suivront pas de thème clair. Cependant, si l'on se restreint à des extraits, ils sont parfois suffisamment bons pour paraître authentiques.

La génération est faite caractère par caractère, et non mot par mot ! Autrement dit, en étudiant les 50 derniers caractères et en générant le caractère suivant, notre modèle arrive à reconstruire des mots corrects sans fautes d'orthographe. Il construit des structures grammaticales propres, accorde les adjectifs, conjugue les verbes et reconstruit des parties de phrases qui ont du sens. Tout cela à partir d'un apprentissage du seul texte brut. Ce résultat montre de façon très "parlante" le potentiel des LSTM pour apprendre des dynamiques séquentielles.

Voici des exemples du texte obtenu :

> C'est une politique de constitution européenne des ressources en faveur de l'acceptation de la sécurité sociale.

> Nous abordons en ce moment le projet de la société et de l'état et de l'armée d'extrême gauche et de la conception de la société et le chômage, la croissance et la concertation actuelle que celle de l'Europe à la formation des travailleurs de Moulinex.

9) Références

http://karpathy.github.io/2015/05/21/rnn-effectiveness/
http://hyperbase.unice.fr/hyperbase/?edition=standard
https://fr.wikipedia.org/wiki/R%C3%A9seau_de_neurones_r%C3%A9currents
https://github.com/Saxamos/text-generator