MS IA
2021 - 2022

IA717 - NATURAL LANGUAGE PROCESSING

présenté par

Tancrède HAYER
Alexandre LE BRIS
Souhail OUMAMA
Florian TORROBA

Génération de poésie par LSTM et RNN

Projet effectué du 23 Novembre 2021 au 07 Fevrier 2022

Dirigé par : **Chloé CLAVEL - Cyril CHHUN**

Introduction

La génération de texte est l'une des nombreuses percées de l'apprentissage automatique dans le domaine du traitement du langage naturel (NLP). Cette avancée est très utile dans le monde des arts créatifs pour composer des chansons, des poèmes, des nouvelles et même des romans. Nous avons décidé de canaliser le Shakespeare qui est en nous en construisant et en entraînant un modèle de réseau neuronal qui génère des poèmes en prédisant le prochain ensemble de mots à partir du texte de départ à l'aide de `RNN` (Recurrent Neural Networks) et de `LSTM` (Long Short Term Memory).

Définition de la tâche

Lors de ce projet, nous nous sommes intéressés à la génération de vers, et plus précisément des ***alexandrins***, c'est à dire les vers composés de douze syllabes.

Nous pouvons citer comme exemple la célèbre réplique du Phèdre de Racine :

« Tout m'afflige et me nuit, et conspire à me nuire. »

Nous souhaitons également nous concentrer sur les rimes en paires, et notamment les rimes dites *plates*.
Par example, ce quatrain composé d'une alternance de deux vers à rime masculine et deux vers à rime féminine.

« Tout m'afflige et me nuit, et conspire à me nuire.
Comme on voit tous ses vœux l'un l'autre se détruire !
Vous-même, condamnant vos injustes desseins,
Tantôt à vous parer vous excitiez nos mains ; »

Nous aurons donc besoin de receuillir des données qui contiennent des vers en *alexandrins* qui nous serviront à entrainer notre modèle. Ces données auront besoin de prétraitement au préalable afin de pouvoir transformer notre texte brut en données encodées que le `LSTM` peut intérpréter.

Il faudra par la suite construire l'architecture de ce réseau de neurones qui serait adaptée à notre objectif. Cela veut dire qu'il puisse prendre en entrée nos données formatées, et qu'il soit capable, après son entrainement,de nous générer une suite en vers à un input donné.

Exemple :

`INPUT` :
« andromaque. »

`OUPUT` [1] :
« et que je ne puis voir que le ciel m' a fait naître
et que vous m' avez dit que je ne puis comprendre
mais je ne sais qu' un coeur qui me fait de la sorte
et que de la raison je ne suis point de peine
et je n' en ai point dit et je ne vous puis voir
et je n' ai pas besoin de vous voir en ma vie »

Une fois notre texte généré, nous évaluerons les performances de notre modèle, en évaluant son précision à générer des Alexandrins, sa capacité à incorporer de la rime, et enfin son dégré d'erreur au niveau de la syntaxe et de l'orthographe.

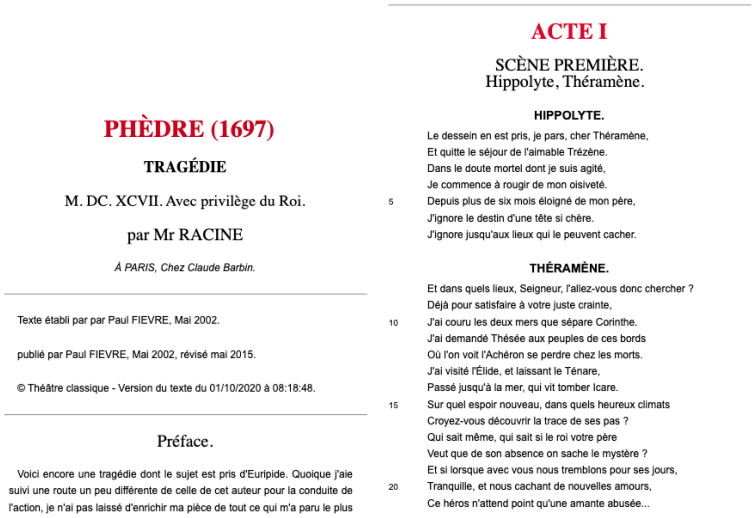
1. Séléction de rimes sur diverses générations

Chapitre 1

Acquisition des données et Preprocessing des données

1.1 Acquisition des données

Ayant besoin d'un grand nombre de textes en Alexandrins, la meilleure source disponible reste le corpus `DRAMACODE` [1], contenant 897 pièces de théatres classiques Françaises, (comédie, drame, et tragédie).



PHÈDRE (1697)

TRAGÉDIE

M. DC. XCVII. Avec privilège du Roi.

par Mr RACINE

À PARIS, Chez Claude Barbin.

Texte établi par par Paul FIEVRE, Mai 2002.

publié par Paul FIEVRE, Mai 2002, révisé mai 2015.

© Théâtre classique - Version du texte du 01/10/2020 à 08:18:48.

Préface.

Voici encore une tragédie dont le sujet est pris d'Euripide. Quoique j'aie suivi une route un peu différente de celle de cet auteur pour la conduite de l'action, je n'ai pas laissé d'enrichir ma pièce de tout ce qui m'a paru le plus

ACTE I

SCÈNE PREMIÈRE.
Hippolyte, Théramène.

HIPPOLYTE.

Le dessein en est pris, je pars, cher Théramène,
Et quitte le séjour de l'aimable Trézène.
Dans le doute mortel dont je suis agité,
Je commence à rougir de mon oisiveté.
Depuis plus de six mois éloigné de mon père,
J'ignore le destin d'une tête si chère.
J'ignore jusqu'aux lieux qui le peuvent cacher.

THÉRAMÈNE.

Et dans quels lieux, Seigneur, l'allez-vous donc chercher ?
Déjà pour satisfaire à votre juste crainte,
J'ai couru les deux mers que sépare Corinthe.
J'ai demandé Thésée aux peuples de ces bords
Où l'on voit l'Achéron se perdre chez les morts.
J'ai visité l'Élide, et laissant le Ténare,
Passé jusqu'à la mer, qui vit tomber Icare.
Sur quel espoir nouveau, dans quels heureux climats
Croyez-vous découvrir la trace de ses pas ?
Qui sait même, qui sait si le roi votre père
Veut que de son absence on sache le mystère ?
Et si lorsque avec vous nous tremblons pour ses jours,
Tranquille, et nous cachant de nouvelles amours,
Ce héros n'attend point qu'une amante abusée...

((a)) Préface. ((b)) Acte 1.

Figure 1.1 – *LE CID - Pierre Corneille / Tragédie (1637)*
[2]

1.2 Pre-Processing des données

Une fois les données récupérées, nous nous rendons compte que les textes contiennent des métadonnées qui nous sont inutiles, à savoir toute la structure de la pièce qui sert à mettre le lecteur en contexte (préface, noms des personnages, numérotation des actes, etc.). Il nous faudra dans un premier temps extraire les alexandrins, et appliquer quelques méthodes de prétraitement (Stemming, Lemmatization, Punctuation Handling) afin de les préparer à l'encodage.

Filtrage des alexandrins

Afin de réaliser ce filtrage en entrée, nous définissons une fonction qui prend un texte en entrée, et en extrait toutes les lignes qui sont des Alexandrins. Concrètement, notre fonction retire tous les caractères qui ne sont pas alphabétiques (ponctuation, chiffres, etc.), passe toutes les lettres en miniscule, et parcours le texte par ligne en comptant le nombre de syllabes et en ne gardant que celles qui sont de taille 12.

1. https://dramacode.github.io/

Stemming (*Racinisation*)

La Racinisation, ou *Stemming*, permet d'extraire la racine des mots, c'est-à-dire d'en retirer les suffixes voire préfixes, par exemple dans le cas de verbes conjugués dont on ne garderait que le radical. Les trois algorithmes de Stemming couramment utilisés sont : `Porter`, `Snowball(Porter2)`, et `Lancaster` (Paice-Husk), qui sont à divers niveaux d'agressivité en extrayant les racines. Ils sont tous proposés par le module `NLTK` (Natural Language ToolKit).

`Extrait de code (using NLTK - SnowballStemmer)|`

```
from nltk.stem import SnowballStemmer
from nltk.tokenize import word_tokenize

def stem_lines(lines):
    stemmer = SnowballStemmer("french")
    output_lem = []
    for sentence in lines:
      output_lem.append(" ".join([stemmer.stem(i) for i in sentence.split()]))
    return output_lem
```

`Exemple|`

« entre tous ces amants dont la jeune ferveur.»
`devient`
« entre tous ce amant dont la jeun ferveur. »

Lemmatization

De manière similaire, la lemmatisation permet de récupérer la racine d'un mot. Cependant, la différence est que la racine récupérée doit elle-même être un mot ayant un sens, alors que la lemmatisation renvoie parfois des troncatures de mots qui n'existent pas dans le dictionnaire.

`Extrait de code|`

```
import spacy
def lemmatize_lines(lines):
  lines_lemmmatize = []
  nlp = spacy.load('fr')
  for line in lines:
      line_lemmmatize = ''
      doc = nlp(line)
      for token in doc:
          line_lemmmatize += token.lemma_ + ' '
      lines_lemmmatize.append(line_lemmmatize)
  return lines_lemmmatize
```

`Exemple|`

« entre tous ces amants dont la jeune ferveur.»
`devient`
« entre tout ce amant dont le jeune ferveur. »

Combiner les méthodes

Ayant diverses méthodes à notre disposition, le défi se révèle être le choix des outils les plus pertinents, et qui génèrent le meilleur résultat. L'expérience nous a montré que parmis les méthodes proposées, seuls le filtrage des Alexandrins et la gestion de la ponctuation ont un impact positif sur le texte généré. Cela peut-être dû au fait qu'en réduisant l'ensemble des mots à leurs racines, nous brisons ainsi la sémantique et perdons donc le sens caché dans nos phrases. Pour la suite de ce projet, nous ne retenons donc que cette combinaison.

Répartition des données en Train, Validation et Test

Nous allons maintenant diviser notre données en trois ensembles: `Train`, `Valid` et `Test`. Le `Train` sera utilisé pour entrainer le modèle, c'est-à-dire pour mettre à jour ses paramètres. Le `Validsera` utilisé pour suivre l'évolution de la perte pendant l'entrainement, et le `Test` sera utilisé pour évaluer le modèle après l'entrainement. Nous choisissons de répartir nos données en `Train = 80% - Valid = 10% - Test = 10%`

Chapitre 2

Formatage des données et création du modèle d'entraînement

2.1 Construire le Corpus

Une fois les textes prétraités, combinés puis répartis en ensemble Train / Valid / Test, il convient dorénavant d'extraire autant d'information que possible de ces données textuelles, puis de formater l'ensemble de façon à pouvoir alimenter notre modèle de réseau de neurones : c'est la phase de création de l'objet Corpus. Celui-ci contient les trois ensembles Train / Valid / Test codés sous forme de tokens (jetons), chacun d'eux représentant un mot ou une de ses modalités - en l'occurrence, dans le cadre de notre projet, la modalité grammaticale (Part-Of-Speech ou POS) ou la modalité phonétique (phonème). Cette dernière modalité est d'autant plus importante qu'elle porte l'information sur les rimes ; la modalité grammaticale, dans les faits, apporte peu à notre modèle (l'apprentissage par celui-ci de longues séquences grammaticalement justes permet de reproduire dans les résultats une première forme de justesse grammaticale), mais permet de discriminer deux phonèmes dans certains cas ambigus (par exemple, « les poules couvent » vs « les poules du couvent »). Enfin, des dictionnaires sont créés pour passer du token (utilisé par le réseau de neurones) à l'élément associé, mot /POS / phonème selon les cas.

Nous résumons dans le tableau ci-dessous les différentes étapes de création du Corpus :

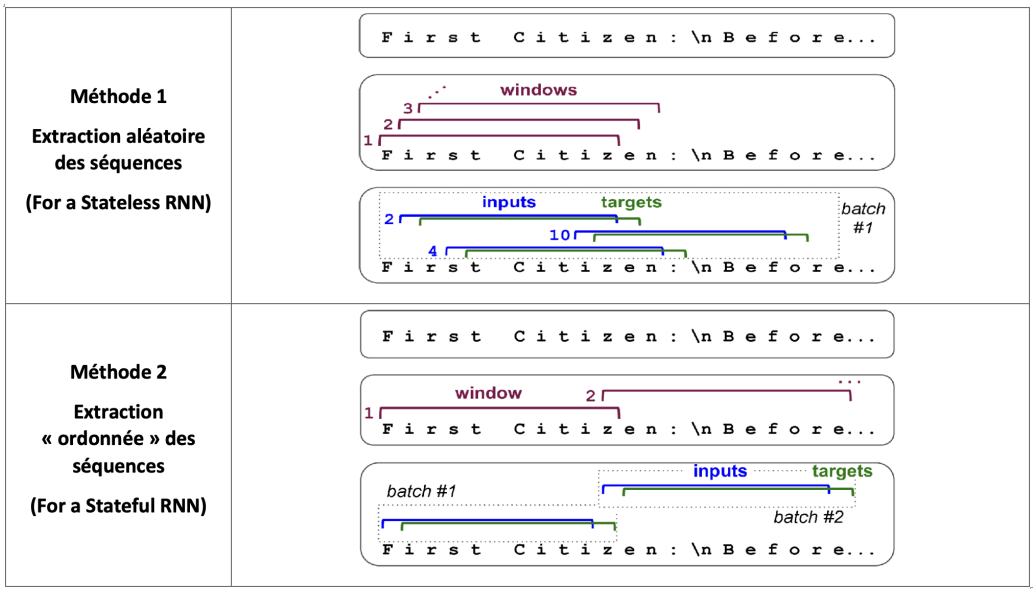
Etapes	Opérations	Outils utilisés
1	— **Tokenisation des mots** — Création du dictionnaire associé — Réordonnancement de l'index des tokens selon la fréquence d'apparition du mot (indice faible, fréquence importante - ex : je, ou, mais, etc.) [1]	
2	— **POS tagging des mots et tokenisation des POS** — Création du dictionnaire associé	**Stanza** (Stanford NLP Group)
3	— Croisement des informations lexicales et grammaticales pour extraire une transcription phonétique de chaque mot — Conservation de la partie terminale de la transcription phonétique (appelée ici phonème), pour conserver une information pour les rimes. — **Tokenisation des phonèmes** — Création du dictionnaire associé	Dictionnaire grammatico-phonétique **Morphalou** (Ortolang)
4	— Réduction de la taille du vocabulaire lexical (de 50 000 à 20 000 mots dans notre étude, afin de limiter la taille de la couche terminale de notre réseau de neurones, de loin la plus lourde en nombre de paramètres, et susceptibles d'overfitting) [2] — Plongement lexical (embeddings) des mots conservés	**SPACY**

Nous obtenons alors le résultat suivant (opération lourde qui nécessite près d'une heure pour traiter l'équivalent de 500 textes répartis entre nos 3 ensembles de données textuelles) :

Indice	Mot (info lexicale) - 20 000 tokens	POS (info grammaticale) - 17 tokens	Phonème ou rime (info phonologique) - 650 tokens
278	entre	ADP	t r
66	tous	ADJ	t u
21	les	DET	l e
578	amants	NOUN	m ã
85	dont	PRON	d ˜
9	la	DET	l a
688	jeune	ADJ	œ n
7656	ferveur	NOUN	œ r

2.2 Production des batchs

Le batch correspond à un paquet de données destiné à alimenter et entraîner notre réseau de neurones à chaque itération. Il est constitué d'un nombre fixe de séquences de mots (entrées ou inputs) accompagnées de leur séquence cible à atteindre (targets). Ces séquences seront traitées en parallèle et les scores seront agrégés au sein de la fonction perte avant rétro-propagation du gradient. La méthode la plus simple et rapide pour créer ces paires inputs-targets est de les extraire de façon aléatoire depuis les données de notre Corpus (voir méthode n°1 du graphique ci-dessous). De plus, elle permet d'exploiter au maximum la taille de nos datasets.



Cette méthode a toutefois le désavantage de perdre la cohérence globale d'un texte et d'ignorer sa structure rythmique (alexandrins) et phonétique (rimes). Afin de conserver ce type d'information, et exploiter ainsi au mieux les capacités du réseau de neurones à détecter et à reproduire les motifs rythmiques et phonétiques des textes poétiques, il convient de construire des batchs qui respectent l'ordre original des données textuelles. Pour ce faire, on adoptera une méthode séquentielle dont le principe est résumé ci-dessus (cf. méthode 2).

Dans notre contexte, pour des textes présentant des longueurs variées, le protocole suivant a été adopté :

Cette méthode présente le désavantage de mettre au rebut une partie des données. Elle présente néanmoins l'avantage de garder une continuité sémantique, rythmique et phonétique d'un batch à l'autre, comme le montre la dernière étape du schéma. On obtient ainsi des données d'entraînement qui exploitent au mieux l'état caché (hidden state) des réseaux de neurones récurrents, celui-ci portant la mémoire de la séquence de mots donnée en entrée (on parle alors de stateful RNN).

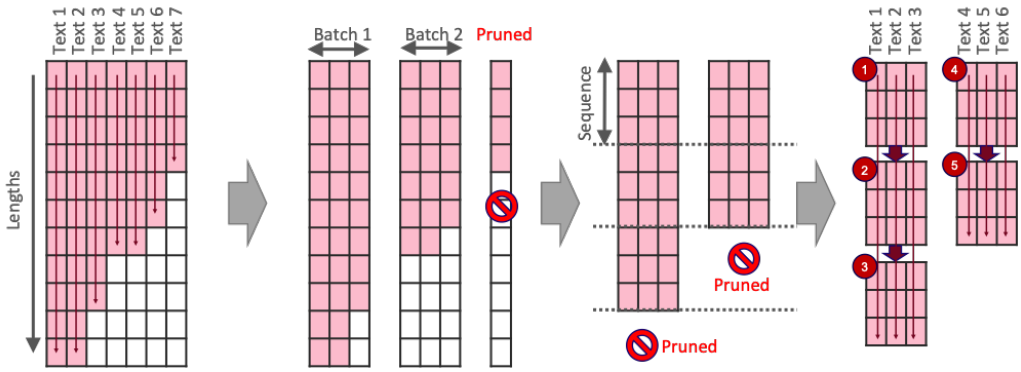


Figure 2.1 – *Etapes pour la production de batchs ordonnés.*

2.3 Architecture du réseau de neurones LSTM

Pour la génération d'un texte poétique, à savoir une séquence de mots respectant des critères d'ordre lexicographique, grammatical, rythmique et phonétique, l'attention se porte naturellement sur un réseau de neurones récurrent (RNN). Nous retiendrons la proposition initiale, consistant à utiliser un LSTM à 2 couches (les architectures de type GRU ou Transformer n'ont pas été abordées durant ce projet). 4 modèles ont été élaborés durant ce projet :

1. Le modèle « classique » (classe LSTMModel) : seuls les mots sont utilisés. Ceux-ci, sous forme de tokens, passe par une couche de type Embedding avant d'alimenter un LSTM à 2 couches et un decoder de type Linear.
2. Le modèle « Concaténation Mot/Phonème » (classe LSTMModel WordPhon) : les mots et les phonèmes, sous formes de tokens, sont donnés en entrées. Chacune des sources passe par une couche Embedding dédiée et les résultats sont concaténés avant d'alimenter le LSTM. La taille des embeddings pour les mots et phonèmes dépendent de la taille de leur vocabulaire respectif : on prendra en général un rapport (mots/phonèmes) de 1.5 à 2, correspondant au rapport du logarithme de leur taille de vocabulaire respectif (log(20 000) / log(650) 1.5) [3]
3. Le modèle « Concaténation Embedding/Phonème » (classe LSTMModel EmbPhon) : idem que ci-dessus, à l'exception que les mots en entrée sont directement donnés sous forme d'embeddings pré-entrainés (modèle fourni par Spacy).
4. Le modèle «Attention Mot/Phonème » (classe LSTMModel WordPhonAtt) : idem que le modèle 2, si ce n'est que les embeddings du mot et du phonème sont fusionnés à travers une couche d'attention (couche d'attention additive). [4]

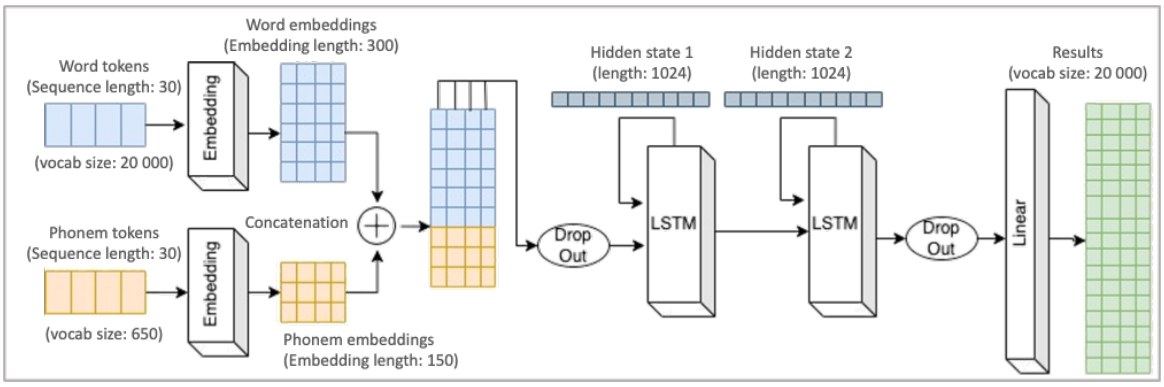


Figure 2.2 – *Modèle 2 annoté « Concaténation Mot/Phonème » (classe LSTMModel WordPhon).*

3. Le logarithme a été choisi comme mesure naturelle de la quantité d'information.
4. https://d2l.ai/chapter$_a$*ttention* − *mechanisms*/*attention* − *scoring* − *functions.html*

Pour le reste du modèle, nous avons conservé les choix initiaux :

— **Fonction de perte (Loss)** : `Cross Entropy Loss`. Une version pondérée a été testée (poids proportionnel à la fréquence du mot, donc inversement proportionnel à son rang dans le dictionnaire [5]) afin de limiter les effets du déséquilibre entre différentes classes de mot et favoriser ainsi l'apprentissage des mots moins courants pour une plus grande diversité (voir ci-dessous).

— **Optimizer** : `Stochastic Gradient Descent (SGD)`. L'optimizer Adam a été testé, sans succès. Le taux d'apprentissage a été fixé à 10 au départ, avec une décroissance d'un facteur 0.7 à chaque epoch

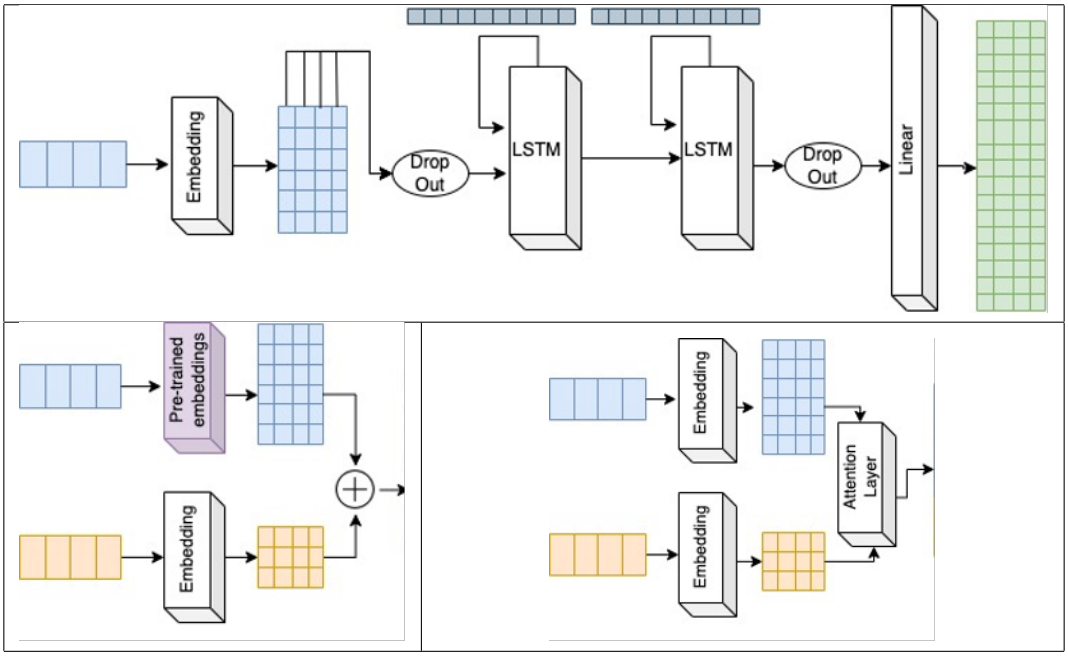


Table 2.1 – *Extraits des autres modèles de neurones testés (en haut : le modèle classique ; en bas à gauche : le modèle « Concaténation Embedding / Phonème » : en bas à droite : le modèle « Attention Mot / Phonème »*

2.4 Entrainer le modèle

Sur les schémas ci-dessous, obtenus à l'aide de Tensorboard, nous observons l'évolution de la loss sur 7 epochs pour un modèle classique (sur le set d'entraînement, puis sur celui de validation). [6]

— **La courbe bleue** : taille de batch de 64 et batchs « ordonnés »

— **La courbe orange** : taille de batch de 8 et batchs « ordonnés »

— **La courbe grise** : taille de batch de 8 et batchs « aléatoires »

Concernant l'entraînement du modèle, plusieurs observations sont à noter:

— Avec des batchs de grande taille, l'entraînement du modèle est plus rapide (grâce à la parallélisation des calculs) ; il faut toutefois plus d'itérations pour faire baisser la loss (une hypothèse probable serait que la moyenne de la loss sur un plus grand nombre de séquences « lisse » certaines spécificités à apprendre et diminue ainsi a vitesse d'apprentissage).
Pour la suite, on retiendra donc un nombre de batchs égal à 8.

— Si on observe l'évolution de la loss sur le set de validation, on observe un biais important entre batchs « larges » et batchs « étroits », mais également entre batchs « aléatoires » et batchs « ordonnés », paradoxalement en faveur de ce premier (alors que l'intuition privilégie le second set de batchs, qui conserverait mieux a priori les structures rythmiques et phonétiques des poèmes). Aucune explication à ce stade permet de justifier ce phénomène.
Nous conserverons donc pour l'entraînement des modèles des batchs de séquences générées aléatoirement

5. Poids 1 / log(rang du mot dans le dictionnaire)
6. L'abscisse représente le temps en heure.

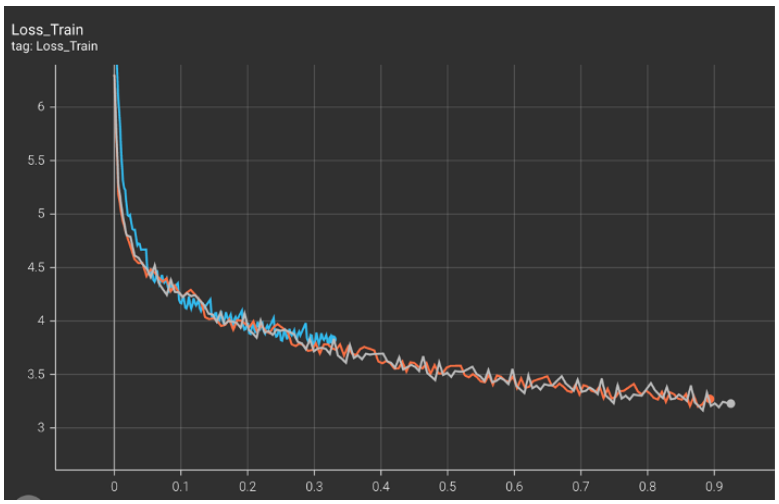


Figure 2.3 – *Évolution de la Loss pour le set d'entraînement en fonction du temps (en h) pour 7 epochs*

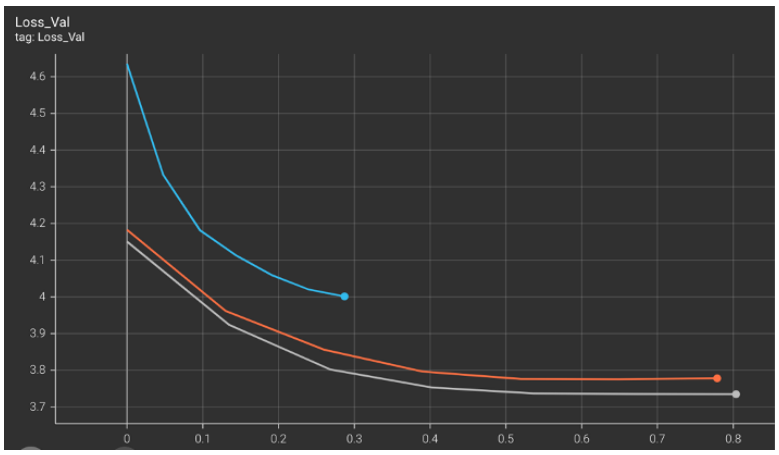


Figure 2.4 – *Évolution de la Loss pour le set de test en fonction du temps (en h) pour 7 epochs*

— La loss sur le set de validation se stabilise aux alentours de la 5e epoch, alors que la loss sur le set d'entraînement continue de baisser.
Afin de limiter tout sur- apprentissage, nous nous limiterons pour la suite à 5 epochs d'entraînement

2.5 Hyperparameter Tuning

Un modèle de Deep Learning est composé de deux différents types de paramètres : les hyperparamètres (qui sont défini de manière arbitraire par l'utilisateur) et les paramètre du modèles (qui sont appris pendant l'entrainement du model).

Les paramètres du modèle vont donc définir comment les données d'entrés vont être traitées pour obtenir la sortie souhaité. La ou, les hyperparamètres déterminent la structure de notre modèle en premier lieu.

La problématique issus de l'ensemble de nos hyperparamètres reviens à un problème d'optimisation. Nous disposons d'un ensemble de valeur et nous cherchons à trouver soit le minimum d'une fonction, avec la perte, ou bien le maximum, par la précision.

Il existe plusieurs approches pouvant résoudre ce problème, notamment on s'intéresse à:
— Recherche manuelle
— Recherche par grille (Grid Search)
— Recherche Aléatoire (Random Search)
— Réglage automatique des Hyperparameter (Bayesian Optimization, Genetic Algorithms)
— Réglages par Réseau de Neurones

2.5.1 Manual Search

Lorsque nous utilisons la recherche manuelle, nous choisissons certains hyperparamètres du modèle en fonction de notre jugement/expérience. Nous formons ensuite le modèle, évaluons sa précision et recommençons le processus. Cette boucle est répétée jusqu'à ce que l'on obtienne une précision satisfaisante.

- Seed
- Embedding Size
- Number of Layers
- DropOut rate
- Sequence Length
- Log Interval
- Learning Rate
- Optimizer

2.5.2 Grid Search

La stratégie mise en place dans une recherche par grille est la suivante, nous établissons une grille de nos hyperparamètres et testons le modèles sur chacune des combinaisons possibles, d'après la grille. Une fois les itérations faites nous pouvons choisir la combinaison ayant obtenu le meilleur score. Lors de l'utilisation de Grid Search le fait que toutes les combinaisons soit testé crée un cout de calcul très élevés.

Dans notre cas nous avons 13 paramètres. Si il 'excite que 2 cas de figures par paramètres nous devons néanmoins tester 8192 combinaison. Il existe bien sur plus de 2 cas par paramètres l'approche n'est donc pas possible compte tenu du temps qu'il faut pour faire trouver le modèle. Un autre problèmes réside dans le caractère arbitraire des points testés.

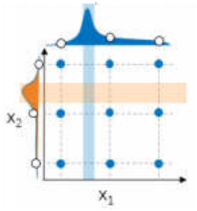


Figure 2.5 – *Recherche par grille*

Dans la figure ci-dessus, neuf combinaisons sont examinées. Néanmoins si neuf modèles ont été formés et évalués on remarque que les points optimaux ont été manqués. Une approche par random search pourrai arranger quelque peut le problème.

2.5.3 Random Search

Il s'agit d'une variante amélioré de l'instruction précédente, par un échantillonnage aléatoire das un espace de recherche au lieu de le discrétiser par une grille cartésienne. Le programme n'a pas de fin, on spécifie un budget temps (un nombre d'essais). Cet boucle souffre également de la malédiction de la dimensionnalité. L'un de ses avantage en comparaison est que si deux hyperparamètres sont peu corrélés, le fonctionnement aléatoire permet de trouver plus précisément les optima de chaque paramètre, comme le montre la figure ci-dessous.

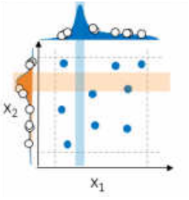


Figure 2.6 – *Recherche aléatoire*

Au final nous retrouvons le même problème du nombre de paramètre qui crée un nombre de combinaison nous empêchant de réellement procéder de cette manière d'après nos test.

2.5.4 Bayesian Optimization Search

L'optimisation bayésienne peut être facilement expliquée en examinant la régression par processus gaussiens. Les processus gaussiens sont des fonctions qui, lorsqu'elles sont échantillonnées sur k points aléatoires, suivent une distribution gaussienne multivariée.

Contrairement à la recherche aléatoire ou par grille, l'optimisation bayesienne garde une trace des résultats d'évaluation précédent. Ces résultats sont utilisé pour former un modèle probabiliste. Les hyperparamètres sont mis en correspondance avec la probabilité d'un score sur la fonction objectif et crée un modele de « substitut » de la fonction objective réelle :

$$P(score|hypermapameters)$$

L'avantage du substitut est qu'il est plus facile à optimiser, la méthode cherche à trouver le prochain ensemble d'hyperparamètre à évaluer à partir de la fonction de substitut. Le processus est donc en plusieurs étapes :

- Construire le modèle de substitution.
- Chercher les hyperparamètres les plus performant sur la fonction de substitution.
- Appliquer les paramètres à la fonction réelle.
- Mettre à jour le modèle de substitution et recommencer les étapes, jusqu'à un nombre d'itération défini.

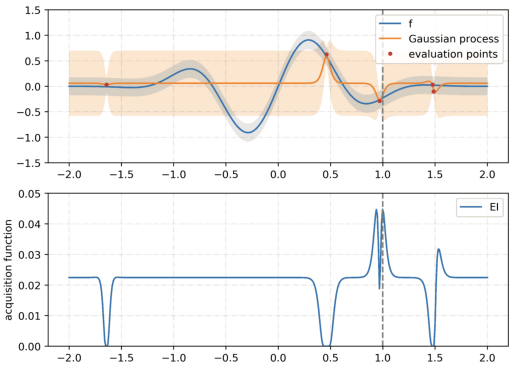


Figure 2.7 – *Fonctionnement du processus*

On retrouve en bleu la fonction objective réelle, en rouge les points évalué et en orange la fonction de «substitut» qui se m'est à jour à chaque nouvelle itération.

Le but est d'évaluer les paramètres les plus prometteur en le cherchant à partir du maximum de la fonction d'acquisition (une fonction du processus gaussien).

L'avantage de se raisonnement est qu'il permet de trouver de meilleur hyperparamètres en moins de temps car ils fonctionnent par rapport à des valeurs antérieur.

Cependant nous nous retrouvons toujours dans le même problèmes qui nous empêche de mener jusqu'au bout ces techniques. Notre puissance de calcule en accès limité nous empêche de réaliser ce processus un nombre de fois désirable pour trouver les meilleurs paramètres de notre système.

Chapitre 3

Ouput et postprocessing des données

3.1 Output - Méthodes de Sampling

Une fois le réseau de neurones récurrents entraîné sur le corpus de texte, nous pouvons procéder à la génération de poèmes à proprement parlé. Pour cela nous donnons en entrée du réseaux de neurones une phrase servant de point de départ pour le reste du poème. Chacun des mots de la phrase initiale est converti sous la forme de son plongement lexical « word embeddings » modifiant ainsi les valeurs des neurones des couches cachées.

Enfin, nous obtenons en sortie une liste de n probabilités (où n est le nombre de mots du dictionnaire) suivant une loi polynomiale ; les indexes de cette liste étant les « tokens » de ces mots et les valeurs de cette liste la probabilité de leur apparence.

Lorsque le prochain mot sélectionné, ce dernier était passé en entrée et un nouveau était généré. Nous répétons cette opération autant de fois que nécessaire afin d'obtenir notre poèmes.

Notre problématique était donc la suivante : selon quel critère le prochain mot devait être sélectionné? Probabilité la plus élevé, rajouter de l'inattendu ou encore le suivi de règles supplémentaires ? Ce problème était exacerbé par le fait que l'évaluation d'un poème ne peut se limiter à des critères aisément qualifiables par une machine. Par exemple, le poème parvient-il à conserver un sens général au fil des différents vers ?

Nous avons donc décidé de mettre en place différentes méthodes d'échantillonnage et d'observer lesquelles permettent d'obtenir des résultats probants. Ces méthodes peuvent être séparées en deux grandes catégories : méthodes déterministes et non-déterministes

3.1.1 Méthodes de Sampling Déterministe

Les méthodes déterministes font le choix des prochains mots en se basant strictement sur la plus haute probabilité ; elles ne diffèrent que selon la façon dont cette probabilité est obtenue ou calculée. Par conséquent, pour un même texte source en entrée, elles donneront toujours le même poème en sortie.

Méthode `Greedy|`

Cette méthode est la méthode la plus simple et la plus aisé à mettre en place ; elle a aussi l'avantage d'être aisément interprétable. Elle consiste simplement à choisir à chaque étape le mot ayant la plus grande probabilité.
Elle a par contre pour conséquence d'offrir des résultats répétitifs (boucles de textes) et le risque de tomber dans une solution sous-optimale.

Méthode `Beam-Search|`

Plus compliquée à mettre en place, elle conserve à chaque étape les k meilleurs « chemins » qu'elle a pu trouver. Elle permet ainsi de découvrir de meilleures solutions qui auraient été cachées derrière des mots moins probables, contrairement à la méthode « Greedy ». Elle est donc plus optimale que cette dernière.
Toutefois, elle a pour inconvénient majeur d'être en moyenne plus répétitive et de demander un temps de calcul plus important. Il est possible de modifier la valeur de l'hyperparamètre k pour améliorer la précision en sacrifiant la rapidité de calcul.

Résultats obtenus

Malheureusement, les résultats que nous obtenons en sortie avec l'une et l'autre méthode sont très peu satisfaisants. Il s'agit en effet du même alexandrin répété indéfiniment. La méthode « Beam-search » n'a pas offert de résultats plus probant avec le même poème en sortie.

```
('et que je ne puis voir que je ne puis comprendre \n'
 ' et que je ne puis voir que je ne puis comprendre \n'
 ' et que je ne puis voir que je ne puis comprendre \n'
 ' et que je ne puis voir que je ne puis comprendre \n'
 ' et que je ne puis voir que je ne puis comprendre \n'
```

Figure 3.1 – *Résultats par méthode Beam-Search*

Nous avions par conséquent décidé de nous tourner vers d'autres méthodes, cette fois-ci non- déterministes.

3.1.2 Méthodes de Sampling Non-Déterministe

Ces méthodes permettent de rajouter une composante d'inattendu dans la génération des poèmes. Un mot a désormais pour chance d'être choisi la valeur de sa probabilité (par exemple : un mot ayant 0.2 en probabilité aura 20% d'être choisi) en suivant la distribution polynomiale. Contrairement aux méthodes déterministes, celles-ci peuvent générer des poèmes différents à partir du même texte source.

Méthode Température

Il ne s'agit pas là d'une méthode de génération d'échantillonnage à proprement parlé. Elle a davantage tendance à s'incorporer aux méthodes qui suivent en modifiant les valeurs des probabilités en sortie. Elle consiste à rééquilibrer les probabilités en les divisant par une valeur (l'éponyme température) puis à utiliser une fonction de softmax sur les valeurs de sortie. Pour une température inférieure à 1, cela a pour conséquence que les mots les plus probables deviennent plus probables et les mots les moins probables encore moins probables ; et vice-versa pour une température supérieure à 1.

Méthode Random

Facile à mettre en place, elle consiste à choisir le prochain mot parmi l'ensemble des mots du corpus. Si elle est aisément interprétable et évite les répétitions, elle a par contre pour inconvénient d'être parfois trop aléatoire. Ce problème peut être mitigé par la variation de la température.

```
('joli collier rapprocher agitent establir échos gascon loi vieillesse faussa '
 'universités épuisa goguelureau céladons conseillère garderait » du '
 'déguisement clouer est produire… applaudissez fernand voeux tapissez '
 'parsonne montiez battrait eumée départir tourmentais voyla français '
 'chevalerie \n'
```

((a)) Température = 2

```
('et que je ne puis voir que le ciel m’ a fait naître \n'
 ' et je ne puis souffrir que je ne puis souffrir \n'
 ' et que vous m’ avez dit que je ne puis comprendre \n'
 ' et je ne puis souffrir que je ne puis souffrir \n'
 ' et que je ne puis voir que je ne puis comprendre \n'
```

((b)) Température = 0.1

```
('et que je ne puis voir une telle victoire \n'
 ' mais je ne puis douter de parler de ma vie \n'
 ' et que je ne puis voir que l’ amour m’ a promis \n'
 ' qui ne vous laisse point d’ un si grand sacrifice \n'
 ' il ne peut pas parler à l’ amour de mon père \n'
 ' et que l’ on m’ a dit qu’ un peu de tout à la grèce \n'
```

((c)) Température = 0.3

Figure 3.2 – *Résultats par méthode Random*

Méthodes TOP-K et TOP-P

Ces deux méthodes consistent à choisir le prochain mot parmi un groupe plus réduit de mots. Elles ont pour avantage non seulement d'éviter les répétitions et de rajouter de l'inattendu sans obtenir des résultats trop aléatoires. Elles ne diffèrent que dans la façon d'obtenir ce groupe réduit de mots.

— `TOP-K`

```
('je n’ ay point de pouvoir pour un autre que j’ aime \n'
 ' mais vous ne pouvez point le faire à la vengeance \n'
 ' mais il faut le servir de ce qu’ il a de vous \n'
 ' et que de l’ amitié je ne suis point de foi \n'
 ' je ne suis pas encor qu’ il faut que je vous vois \n'
```

Figure 3.3 – *Sélection des K mots avec la plus forte probabilité.*

— `TOP-P`

```
('je n’ ay point de pouvoir pour un autre que j’ aime \n'
 ' mais vous ne pouvez point le faire à la vengeance \n'
 ' mais il faut le servir de ce qu’ il a de vous \n'
 ' et que de l’ amitié je ne suis point de foi \n'
 ' je ne suis pas encor qu’ il faut que je vous vois \n'
```

Figure 3.4 – *Sélection de mots des + au - probables jusqu'à ce que la somme de leur probabilité soit > P*

— `TOP-P Rhyming` Cette méthode est identique au Top-P, à la différence près qu'elle détecte les mots générés en dernière position dans le vers et, parmi ces candidats, favorise le score de ceux qui engendrent des vers rimés. On utilise pour cela la fonction de sélection d'alexandrins (utilisée dans le pré-processing des données - Chapitre 1) et les fonctions de transcription phonétique / sélection des derniers phonèmes (utilisées dans la création du Corpus - Chapitre 2). Cette astuce permet un gain de 50% sur la production de rimes (voir l'évaluation des modèles - Chapitre 4).

```
' ce n’ est point la faute où mon âme s’ égare \n'
' quelle rigueur me tient à mon désespoir \n'
' mais l’ unique malheur qu’ il vous donne à m’ entendre \n'
' ou je crains que ma mort n’ ordonne d’ entreprendre \n'
' ou d’ entendre ma mort ou de se contenter \n'
' je me trompe ou mon coeur parle avec fermeté \n'
' s’ il ose se contraindre à son commandement \n'
' c’ est le seul moyen de garder son amant \n'
' non non par mon hymen il n’ a point de refuge \n'
' de qui l’ aveugle choix est un hymen funeste \n'
' sa valeur ne le peut garder que sa vertu \n'
' vous voyez ses dédains et votre qualité \n'
' il voit avec plaisir toute sa dignité \n'
```

Figure 3.5 – *Génération d'un poème avec la méthode TOP-P Rhyming*

Conclusion

A partir de ces résultats, nous pouvons observer que les meilleures méthodes semblent être TopK et TopP avec la méthode Random pouvant donner des résultats intéressants pour des valeurs de température appropriées.

Toutefois, malgré la présence d'alexandrin et de structures grammaticales à peu près correctes, nous remarquons que ces poèmes ne présentent qu'un faible degré de cohésion : les vers semblent presque indépendant les uns des autres.

3.2 Post-Processing des vers générés - Parsing syntaxique

Une autre variante étudiée pour générer du texte qui rime consiste en réalisant un traitement post génération. Cette méthode se base sur le Parsing syntaxique et est inspirée de l'article de Gil SHOTAN et Rafael FERRER [1].

L'algorithme décrit dans cet article prend en entrée une séquence de texte, qui dans notre cas sera généré par le modèle neuronal entrainé, et qui transforme deux paires de phrases pour les faire rimer, en effectuant une série de transformations syntaxiques sur l'une ou les deux phrases afin que les phrases résultantes riment, tout en conservant leur sens original. Nous aborderons l'implémentaiton de ce modèle à travers 4 étapes qui s'enchainet:

3.2.1 Génération des synonymes

La toute première étape de notre approche consiste à créer une liste de synonymes pour chaque mot des deux phrases, afin de pour produire une réserve dans laquelle nous allons choisir nos rimes.
Pour obtenir les synonymes en langue française, nous nous tournons vers le module Python `CNRTL`, créé par le Centre national de ressources textuelles et lexicales.

3.2.2 Identification des rimes

Suite à cela, nous parcourons chaque paire de mots provenant des deux phrases différentes, et pour chacune de ces paires de mots nous examinons la liste de synonymes qui leur sont associés, y compris les mot originaux eux-même. Si nous trouvons une paire de mots qui riment, nous générons une paire de phrases candidates.

Pour mieux appréhender la notion de la rime, nous utilisons une base de donnée recueillie sur le site ... qui représente plus d'un million de mots en langue Française, sous toutes leurs déclinaisons, en transcription phonétique selon le standard International Phonetic Alphabet (IPA)

3.2.3 Génération des paires de phrases séléctionnées

Une fois que nous avons trouvé une paire de mots qui riment, nous générons une paire de phrases candidates en substituant d'abord les mots originaux par les synonymes rimés, puis en réorganisant la phrase pour placer les mots rimés à la fin de la phrase, avec une perturbation minimale au reste de la phrase.

Essentiellement, l'algorithme essaie de pousser le mot cible à la fin de la phrase tout en essayant de ne pas perturber les parties non liées de la phrase, préservant ainsi les sous-arbres qui sont syntaxiquement corrects dans le processus de réarrangement. De plus, le réarrangement effectué à chaque étape correspond au type de transformations que les gens, et surtout les poètes, utilisent en pratique. Par exemple, la phrase *«Il est allé à sa rencontre.* serait transformée en: *«A sa rencontre, il est allé !»*

3.2.4 Choix de la phrase retenue

La dernière étape de notre parcours consiste à évaluer les paires de phrases produites dans à l'étape précédente, en essayant de sélectionner les candidats les plus prometteurs. Notre première préoccupation est de produire des phrases syntaxiquement correctes car l'heuristique que nous utilisons pour le réarrangement des phrases est loin d'être parfaite.

Par conséquent, nous préférons les paires de phrases candidates qui n'ont subi aucun réarrangement, c'est-à-dire les phrases candidates dans lesquelles les derniers mots, ou leurs synonymes, riment. Si une telle option n'est pas disponible, nous préférons les candidats dans lesquels une seule des phrases a été réarrangée. Enfin, nous considérons les paires de candidats dans lesquelles les deux phrases ont été réarrangées.

Conclusion

En pratique, nous avons constater que la mise en application de cet algorithme n'a pas été fructueuse, car, de par son architecture, ce modèle s'avère être très couteux lors de sa phase d'entrainement. Ceci s'explique par le fait qu'il parcourt la base de données des rimes autant de fois que de mots dans phrases générées, et cela évolue exponentiellement avec le nombre des mots générées, ce qui impacte négativement le temps d'entrainement.

1. https://nlp.stanford.edu/courses/cs224n/2013/reports/shotan.pdf

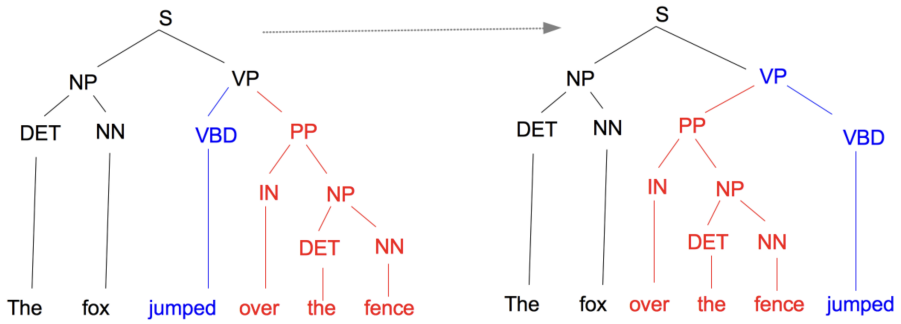


Figure 3.6 – *Transfromation syntaxique afin de retrouver une rime*

Les limites logicielles imposées par Google Colab n'ont permis d'entrainer le modèle construit ainsi que sur quelques epoches, et le résultat qui en a découlé n'était pas probant. La présence de la rime en fin des phrases candidates retenues n'était pas systématique, et nous ne pouvons attester si cela est essentiellement du à l'apprentissage incomplet du modèle, ou s'il n'a pas réellement à accomplir la tâche avec succès.

Chapitre 4

Evaluation du modèle

4.1 Métriques d'évaluation des vers générés.

Nous définissons tout d'abord quelques mesures qui nous permettront d'avoir une vision plus globale sur la performance de notre modèle et sa capacité à générer du texte que l'humain jugerait *poétique*. Nous présenterons par la suite les résultats pour chaque méthode de génération proposée avec en comparaison les performances réalisées sur un extrait du corpus d'entraînement.

4.1.1 Est-ce un Alexandrin ?

Tout d'abord, nous souhaitons évaluer le pourcentage d'alexandrins présents dans les poèmes générés. Pour cela, nous réutilisons la fonction que nous avions créée à l'origine pour préparer le corpus de texte en ne conservant que les alexandrins. Nous pouvons observer qu'à l'exception des méthodes de sampling « Random », le pourcentage d'alexandrins pour chaque méthode ne descend pas en-dessous de 90% avec le corpus originel avec, naturellement, 100% d'alexandrins ainsi que les méthodes « Max » et « Beam-search » répétant en boucle le même alexandrin.

4.1.2 La variété des mots générés.

Pour calculer la variété dans les poèmes générés, nous calculons le nombre de mots uniques sur le nombre de mots total présents dans le poème. Contrairement au pourcentage d'alexandrins, nous ne voulons pas un score de 100% ; en effet, il est attendu, et espéré, que certains mots se répètent, même un grand nombre de fois, tels que 'je' ou 'et'. Nous nous servons par conséquent de la variété du corpus originel pour avoir une intuition de la variété espérée dans les poèmes générées.

Nous observons qu'il s'agit d'une difficulté majeure dans la génération des poèmes. Seule la méthode « Random » avec une température de 1 permet d'approcher une variété similaire au corpus originel. La même méthode pour une température de 2 est beaucoup trop élevé perdant tout sens syntaxique et les autres ont une variété trop faible.

4.1.3 L'orthographe.

Pour aborder cette tâche, nous utilisons le module `pyspellchecker`, qui utilise un algorithme basé sur la distance de Levenshtein pour trouver les permutations situées à une distance de 2 par rapport au mot original. Il compare ensuite toutes les permutations (insertions, suppressions, remplacements et transpositions) à des mots connus dans une liste de fréquence de mots. Les mots qui se trouvent le plus souvent dans la liste de fréquence sont plus susceptibles d'être les résultats corrects.

4.1.4 Enfin, est-ce que ça rime ?

Nous réutilisons ici notre base de donnée phonique employée précédemment dans l'étape du Parsing Syntaxique. Nous répartissons le texte ainsi généré en quatrains, et essayons de déceler s'il y a présence de rime. Nous évaluerons uniquement la présence des rimes dites *plates*, ou *suivies* (AABB), où les vers partageant le même phonème final se succèdent.

NB Nous choisissons d'exclure les rimes `continues(AAAA)`(où tous les vers ont tous le même phonème final, car cela ne favoriserait que le texte obtenue par méthode « Beam-Search » ou par le « Max-Search », qui ont un score de présence de rime de 100% mais qui sont peu satisfantes du point de vue des autres critères considérés.

Nous vérifions notre fonction ainsi définie sur un texte de départ, afin d'avoir un référentiel de comparaison. Nous obtenons les résultats tels que présentés sur le graphe suivant :

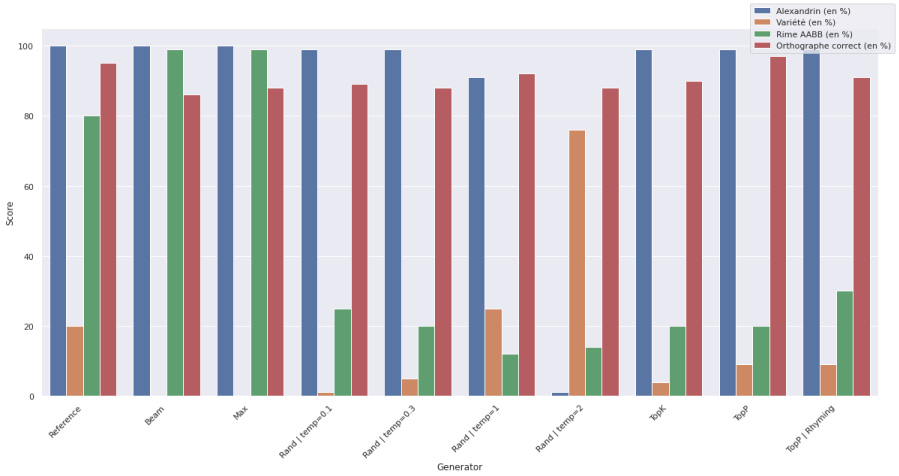


Figure 4.1 – *Présence de rime dans le texte généré, par méthode de sampling*

4.2 Evaluation des différents modèles.

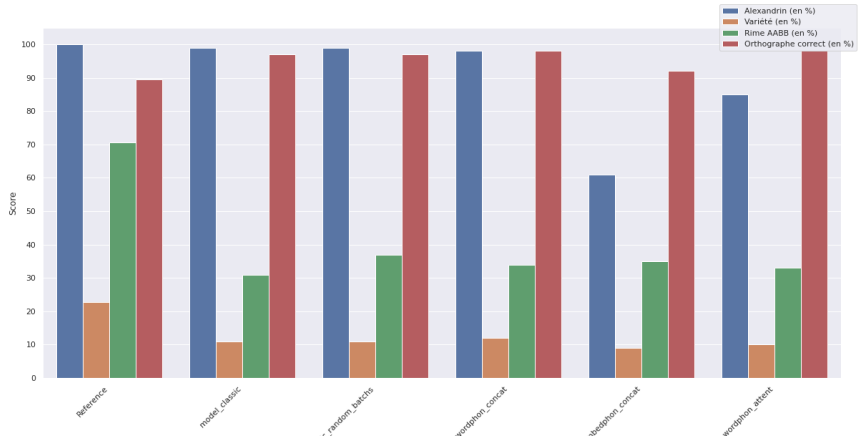


Figure 4.2 – *Comparaison des résultats entre les différents modèles.*

Au chapitre 2, nous avons défini plusieurs architectures de réseau pour notre modèle d'apprentissage. Plusieurs de ces modèles ont été entraînés et testés à l'occasion :

A partir des scores obtenus sur les différents modèles (avec le sampler TOP-P Rhyming), nous pouvons tirer les conclusions suivantes :

— Concernant les alexandrins, nous obtenons globalement de bons scores, sauf pour le modèle qui exploite des embeddings pré-entraînés. Nous pouvons l'expliquer par le fait que les embeddings créés par les autres modèles portent une information de rythme qui n'est pas présente dans les embeddings pré-entraînés (fournis par Spacy).

— Sur la fréquence des rimes, le modèle classique (avec la méthode de génération aléatoire de batches) se révèle le plus efficace (38 % des vers sont rimés). Les autres modèles se valent (30 à 35 %), mais restent loin de la référence (70%).

Nom	Modèle	Batches (aléatoires ou ordonnés)	Taille de l'embedding pour le mot	Taille de l'embedding pour le phonème
model_classic	Modèle 1 classique	ordonnés	512	0
model_classic_random_batches	Modèle 1 classique	aléatoires	512	0
model_wordphon_concat	Modèle 2 - Concaténation Mot / Phonème	aléatoires	300	150
model_embedphon_concat	Modèle 3 - Concaténation Embedding Pré-entrainé / Phonème	aléatoires	300	150
model_wordphon_attent	Modèle 4 - Attention Mot / Phonème	aléatoires	300	150

Figure 4.3 – *Comparaison des résultats entre les différents modèles.*

— La variété du vocabulaire se révèle équivalent entre les différents modèles (score de 10), loin derrière la référence (score d'environ 25). Afin d'étoffer la gamme de mots utilisés, une pondération de la fonction loss, en vue d'avantager les mots moins fréquents du corpus d'entraînement, pourrait être une piste à explorer.

— L'anomalie du score d'orthographe sur la référence est liée à l'utilisation pénalisante de mots rares ou en vieux français, non reconnus par le dictionnaire utilisé.

— **En somme, tous les modèles se valent, avec néanmoins de meilleurs scores pour les modèles classiques, sans utilisation des informations phonétiques. Cela nous donne à penser que les architectures développées et testées n'ont pas été en capacité d'exploiter efficacement la source d'information supplémentaire mise à disposition. Au final, dans le cadre de notre projet et pour ce qui est du travail de mise en rime, le post-processing s'est avéré bien plus efficace que l'élaboration de nouvelles architectures dédiées.**

Conclusion

1. Le modèle « Concaténation Mot/Phonème » (classe LSTMModel WordPhon) : les mots et les phonèmes, sous formes de tokens, sont donnés en entrées. Chacune des sources passe par une couche Embedding dédiée et les résultats sont concaténés avant d'alimenter le LSTM. La taille des embeddings pour les mots et phonèmes dépend de la taille de leur vocabulaire respectif : on prendra en général un rapport (mots/phonèmes) de 1.5 à 2, correspondant au rapport du logarithme de leur taille de vocabulaire respectif (log(20 000) / log(650) 1.5). [1]
2. Le modèle « Concaténation Embedding/Phonème » (classe LSTMModel EmbPhon) : idem que ci-dessus, à l'exception que les mots en entrée sont directement donnés sous forme d'embeddings pré-entrainés (modèle fourni par Spacy).
3. Le modèle «Attention Mot/Phonème » (classe LSTMModel WordPhonAtt) : idem que le modèle 2, si ce n'est que les embeddings du mot et du phonème sont fusionnés à travers une couche d'attention (couche d'attention additive). [2]

1. Le logarithme a été choisi comme mesure naturelle de la quantité d'information.
2. https://d2l.ai/chapter$_attention - mechanisms/attention - scoring - functions.html$

Contributions au projet

Tancrède HAYER :

— PréProcessing / Stemming
— Affinage des hyperparamètres.

Alexandre LE BRIS :

— Création des batches ordonnés.
— Création de différents modèles.
— Méthode de Sampling : Top-P Ryhyming
— Création du Dictionnaire/Corpus
— Evaluation de la loss des modèles.

Souhail OUMAMA :

— PréProcessing / Gestion de la ponctuation.
— Génération du corpus avec les rimes.
— Génération de rime par Parsing syntaxique.
— Evaluation du texte généré (Orthographe + Rimes).

Florian TORROBA :

— PréProcessing / Méthode de détection des alexandrins.
— PréProcessing / Lemmatization.
— Première méthode de batching.
— Toutes les méthodes de sampling à l'exception de TopP-Rhyming.
— Evaluation du corpus (Alexandrin + Variété).

Conclusion

Lors de ce projet, nous avons réussi à entrainer plusieurs modèles de réseaux neuronaux qui sont capables de générer du texte qui rimait, que l'on pourrait qualifier de « Poésie ».

Cependant, nous tenons à préciser que les métriques considérées dans ce rapport afin d'évaluer la qualité du texte généré restent assez limitées. Au delà du fait qu'elles représentent des outils objectifs d'évaluation, elles restent incomplètent, et ont besoin d'un oeil externe humain, qui de par sa subjectivité, sera plus à même de juger le poème sur sa qualité artistique.

Le gros défi restant à relever concerne la sémantique des phrases générées, qui peuvent satisfaire tous nos critères, mais sont dénuées de sens. Un futur travail pourrait creuser la notion des mécanismes d'attention, tels que présentés précédemment, en approfondissant dessus, afin d'essayer de capter plus de *sens* dans les vers de départ, afin d'en insuffler dans le texte généré.