Rapport de projet PRIM-TSIA

Génération d'Alexandrins Classiques par des Méthodes de Deep Learning

Auteur :
Mayer Paul
paul.mayer@telecom-paris.fr

Référents :
Amarilli Antoine
antoine.amarilli@telecom-paris.fr
Beaudouin Valérie
valerie.beaudouin@telecom-paris.fr
Labeau Matthieu
matthieu.labeau@telecom-paris.fr

Institut Polytechnique de Paris

23 février 2021

Résumé

Métrique et rime en main, nous allons générer
De beaux alexandrins, plus beaux que ce dernier.

Même si de tels vers n'ont pas le style des plus grands poètes, créer un tel distique n'est pas bien compliqué pour un être humain. Pour un système informatique, générer un court poème avec du sens, du rythme, de la rime et du style, est une tâche très complexe. Depuis quelques années, les modèles de langage utilisant des méthodes de Deep Learning ont été avancées pour générer de la poésie, et avec un certain succès : en effet ces méthodes peuvent apprendre à partir d'un corpus le style littéraire et les règles qui le composent de manière rapide et assez étonnante. Dans ce rapport nous nous pencherons sur ces méthodes, puis nous présenterons notre solution entrainée sur un corpus de pièces de théatre pour la génération d'alexandrins classiques, une structure poétique très codifiée qui présente des défis supplémentaires pour les systèmes informatiques. Notre projet peut être retrouvé sur cette adresse : `https://github.com/paulmayer-tsp/AlexandRNN`.

Table des matières

Introduction

Dans ce rapport, nous présenterons notre système capable de générer des alexandrins classiques à partir d'un corpus de pièces de théatres. Nous essayerons de contraindre le moins possible un réseau de neurones LSTM simple et essayerons plusieurs méthodes de génération, en les comparant grâce à plusieurs approches d'évaluation.

ACTE I : État de l'Art

Scène I : Création Mécanique

Avant les ordinateurs, la question de la génération automatique de poésie en français a déjà été abordée par des personnalités littéraires et scientifiques fascinées par la créativité sous des contraintes ou l'expérimentation, la recherche d'un processus scientifique dans la création poétique. Parmi eux on retrouve Raymond Queneau, et en particulier son livre *Cent Mille Milliards de Poèmes* qui permet de combiner les quatorze vers d'un sonnet avec dix choix pour chaque vers. On peut donc générer cent mille milliards de poèmes à partir de cent quarante vers, tous gardant la rime, une structure et une thématique solide [1]. Ces travaux seront poursuivis par le collectif dont il est co-fondateur, l'Oulipo.

Scène II : Règles Formelles

La littérature combinatoire proposée par Queneau est encore loin de notre but : générer de la poésie avec un système informatique. Dans les années 90 le groupe ALAMO propose ses *Rimbaudelaires*[1], des sonnets de Rimbaud où les noms, verbes et adjectifs sont remplacés par ceux utilisés par Beaudelaire, suivant des règles rythmiques et syntaxiques. Une des premières méthodes informatiques pour la génération poétique vient de Hisar Manurung (2000) [2], et elle consiste à prendre en entrée une sémantique qui fonctionne comme un bac à idée pour le programme, qui pourra choisir de l'augmenter ou bien de s'y fier complètement, et une structure phonétique rigide souhaitée, composée de la suite d'accents toniques ainsi que le plan de rime voulus, pour produire un poème à partir de ces deux entrées. Un des problèmes rencontré par Manurung était la taille de la base de données qu'il devait utiliser pour pouvoir produire une bonne sortie : en effet le texte poétique se distingue de textes plus simples par un large vocabulaire et des structures syntaxiques complexes et peu rencontrées dans le langage courant. Il faut donc beaucoup d'information en entrée et beaucoup de savoir au système pour pouvoir créer un poème à partir de règles rigides sur une entrée avec l'aide d'une base de données. Pour parcourir ce gigantesque ensemble, Manurung a utilisé la recherche stochastique en escalade, qui consiste à partir d'une configuration initiale, comme un grimpeur partirait d'une prise de départ, et de chercher les configurations voisines dans l'ensemble qui maximisent un score (bonne phonétique, syntaxe, sémantique), comme si le grimpeur cherchait les meilleurs

1. `http://www.alamo.free.fr/pmwiki.php?n=Programmes.Rimbaudelaires`

creux dans le roc qu'il tente de grimper. Après une phase d'initialisation où un candidat respectant la sémantique d'entrée est généré, il va subir des modifications qui vont donner plusieurs candidats possibles, qui vont être évalués par un score. Le programme est fait pour qu'un changement puisse se faire à n'importe quel niveau de la représentation du poème, et ces changements peuvent affecter les autres niveau, ce qui doit être pris en compte dans le score donné à un changement : par exemple, le programme peut décider de changer la structure phonétique du poème candidat en modifiant un mot, ce qui va entraîner un changement syntaxique et sémantique. Après une évaluation de la base de donnée avec cette technique, le programme rentre dans une phase évolutive où les configurations les plus robustes sont choisies et deviennent les nouvelles configurations initiales, jusqu'à arriver à un résultat satisfaisant, étant le fruit d'une sorte de sélection naturelle sur des candidats qui ont muté jusqu'à être adaptés à la fois à la sémantique, mais aussi à la structure phonétique voulue. La solution de Manurung nécessite une entrée précise et a besoin de beaucoup de données annexes au texte. Ce travail s'inscrit dans la lignée du NLG (Natural Language Generation) qui s'appuyait sur des données en entrée pour générer du texte. C'est également dans cette lignée que Pablo Gervás (2001) a commencé à s'intéresser à la génération de poésie avec WASP, un système qui générait des poèmes à l'aide de contraintes strictes sur la métrique ou le thème abordé, et remplissait le poème avec des mots aléatoires satisfaisant les contraintes. Les résultats ayant souvent peu de sens, il a créé ASPERA [3], un système basé sur des règles formelles mais aussi sur du raisonnement à partir de cas. Gervás utilise les solutions à des problèmes rencontrés dans des textes et qui ont été résolus ; Ainsi, si un problème similaire se présente lors de la génération d'un poème, la solution est présente dans le système. Son approche nécessitait un texte en prose en entrée et quelques données supplémentaires de la part de l'utilisateur, comme le style souhaité, et s'appuyait sur un retour de l'utilisateur par rapport à la qualité de la sortie, qu'il intégrait au système. Il s'est aussi heurté au problème de taille de sa base de données complexe et l'a contourné en donnant à chaque mot un ordre de priorité ; Ainsi les calculs se faisaient majoritairement sur les mots les plus prioritaires. Toutefois, la puissance de calcul des ordinateurs lors de ces travaux en génération ne permet pas d'explorer toutes les possibilités théoriques des modèles : on ne peut par exemple pas excéder une certaine taille de vocabulaire, faute de performances.

Scène III : Intelligence Artificielle

Plus tard, en utilisant les mêmes règles et des méthodes d'optimisation plus performantes, ainsi qu'une architecture modulaire, Gonçalo Oliveira propose PoeTryMe qui génère des poèmes en portugais (2012) [4]. Toutes ces méthodes donnent des résultats assez rigides et dépendent beaucoup des données très précises dont elles ont besoin au delà du texte d'entrée. Plus récemment, grâce notamment à l'apport considérable des GPU à la réduction du temps de calcul sur ordinateur, il est possible de travailler avec beaucoup de données, et c'est justement ce que fait l'Intelligence Artificielle. Les Réseaux Récurrents Neuronaux (RNN en Anglais) ont joué un grand rôle dans le développement du domaine de la génération de texte : ces réseaux de neurones sont particulièrement adaptés pour s'entraîner sur des données séquentielles, dont le langage naturel fait partie. Dans le cadre du texte, les RNN prennent en entrée une séquence de mots et essayent de

prédire le mot suivant à chaque étape de la séquence d'entrée. La prédiction ne dépend pas uniquement du mot d'entrée courant, comme dans un réseau de type MLP (Multi-Layer Perceptron), mais aussi de l'état caché du réseau lors de sa prédiction du mot précédent. On a donc une dépendance temporelle dans le réseau, les mots sont appris dans leur contexte. Ainsi, si l'on donne à un RNN entrainé sur un corpus une séquence de mots, il peut prédire le mot suivant de cette séquence. Si l'on rajoute ce mot prédit à l'entrée, on a créé un générateur qui se nourrit lui-même des prédictions qu'il fait pour prédire la suite de la séquence.

Les RNN dans leur forme la plus simple ont étés utilisés par Zhang et Lapata (2014) [5] pour générer des poèmes Chinois. Dans leur solution, deux RNN se succèdent pour générer un court poème à partir de mots clés donnés par l'utilisateur. Ces mots clés sont utilisés pour former un premier vers, qui va servir d'entrée au premier RNN. Celui-ci va avoir un rôle d'encodeur : il va lire les vers d'entrée mot par mot et transformer le vers en une représentation vectorielle. Cette représentation va ensuite servir d'entrée au décodeur, le deuxième RNN, qui va l'analyser et générer le prochain vers grâce à elle. Cette approche encodeur-décodeur est rapidement devenue le standard dans la génération de poésie par des méthodes de Deep Learning. Elle a été raffinée par Rui Yan (2016) [6] : iPoet utilise un mécanisme de raffinement dans son RNN décodeur pour que le réseau s'améliore lorsqu'il crée le poème. Les mots clés de l'utilisateur et le dernier état caché du RNN sont compactés en un nouveau vecteur d'entrée pour le deuxième RNN, offrant ainsi au décodeur une représentation abstraite du thème saisi par l'utilisateur - l'état caché, censé encapsuler la sémantique de l'entrée, mais aussi un rappel précis des mots clés. Yan affirme que cette approche est plus humaine : le poète choisi un thème, écrit un premier jet, rature et revient à son thème initial en gardant des éléments de son premier poème pour le raffiner.

Wang (2016) [7] change la donne en utilisant un réseau LSTM (Long / Short Term Memory units) avec un mécanisme d'attention. Utiliser un LSTM permet d'éviter les problèmes de disparition du gradient qui apparaîssent pendant l'entraînement d'un RNN, et qui causent le modèle à ne pas apprendre la dépendance de la prédiction par rapport au début de la séquence d'entrée. Si l'on veut travailler sur des séquences longues, il devient vite impossible d'utiliser des RNN simples, et c'est pourquoi Wang utilise un LSTM. Le mécanisme d'attention permet de garder des dépendances sur toute l'entrée et peut en accentuer à tout moment une partie pour que le réseau garde bien un thème précis en tête. Malheureusement, un LSTM peine à conserver une cohérence thématique forte tout au long du poème, et la cohérence sémantique (le "sens" des vers) n'est pas optimale. Ghazvininejad (2016) [8] veut également générer des poèmes longs, mais également s'assurer de leur formalité, autant rythmique qu'au niveau de la rime. L'utilisateur donne un mot-clé, à partir duquel un thème, consistant de mots ou d'expressions proches du mot d'entrée est extrait. Des mots-rimes sont alors choisis aléatoirement dans le thème et fixent la partie droite des vers du poème. Pour compléter le poème, un Automate à États Finis (FSA en Anglais) est créé. Les vers des poèmes générés suivent la contrainte rythmique stricte des sonnets de Shakespeare, le FSA est donc constitué d'états de type L_i, S_j où L_i est le $i^{ème}$ vers du poème et S_j la $j^{ème}$ syllabe du vers. Le FSA est peuplé d'arcs correspondant aux mots respectant la contrainte rythmique p our passer d'un état à un autre. Le dernier arc d'un vers est associé au mot rime fixé au préalable par

échantillonnage sur les mots proche du mot de contexte donné par l'utilisateur, et les lignes du poème sont reliés par deux arcs, un arc correspondant à une virgule, et un arc correspondant à un point. Un parcours aléatoire de ce FSA consiste donc à trouver son chemin en choisissant un mot possible parmi tous les arcs de sortie d'un état à chaque étape, en partant de l'état initial. Un tel parcours donne à tous les coups un sonnet formellement correct, mais a de grandes chances de n'avoir aucun sens, le FSA comportant énormément d'arcs empruntables, les seules contraintes étant les mots-rimes et la rythmique du poème. Ainsi, Ghazvininejad utilise un LSTM entraîné sur de nombreuses chansons Anglaises. Lors du parcours du FSA, on emprunte uniquement les arcs associés aux mots les plus probables selon le LSTM, par une méthode de beam-search (voir ACTE II). Pour éviter un problème de répétition et une utilisation de mots fréquents comme des déterminants à outrance, un score faible est donné aux mots qui se répètent et à ceux qui sont peu associé au thème choisi par l'utilisateur. Les résultats sont très satisfaisants, le modèle réussit à conserver une cohérence sémantique et syntaxique bien que le cadre de la génération soit très formel.

Hopkins et Kiela (2017) [9] reprennent l'idée de Ghazvininejad en entraînant un LSTM sur un corpus poétique assez large, dont ils contraignent la forme en utilisant un automate fini. Néanmoins, leur modèle se différencie de tous les autres en n'étant pas basé sur des mots, mais sur des caractères. L'avantage d'entraîner un LSTM sur des caractères est la taille très réduite du vocabulaire, surtout pour la poésie où le nombre de mots dans le vocabulaire d'entraînement atteint souvent les dixaines de milliers. On a généralement moins d'une centaine de caractères dans un corpus, ce qui permet un entraînement plus efficace. De plus les modèles basés sur les caractère peuvent générer des mots qui ne sont pas présents dans le corpus, mais appris suivant les constructions grammaticales qui le composent : le modèle peut générer le mot "inefficace" même s'il n'est pas présent dans le corpus mais que "efficace" et des mots comme "incomplet" ou "inachevé" sont présents. Cela permet de travailler sur un corpus relativement petit tout en assurant la qualité de la génération, qui garde une bonne cohérence sémantique. En mélangeant ce modèle avec un automate à états finis qui assure la rigueur rythmique et de rime du poème généré, le programme peut générer des poèmes de plusieurs formes, dépendemment des règles de l'automate et pas du corpus d'entraînement du réseau de neurones. Hopkins et Kiela ont également tenté d'entraîner un LSTM sur la représentation phonétique des mots, pour apprendre sans contrainte dure la rythmique et la rime. Cette approche nécessite un corpus spécialisé pour que la sortie soit de la forme désirée : si l'on veut générer des sonnets, il faut trouver un corpus de sonnets relativement large. De plus il faut traduire le corpus en sa représentation phonétique, générer des phonèmes avec le modèle, et enfin traduire ces phonèmes en mots pour obtenir un poème, la dernière étape n'étant pas triviale dans le cas d'homophones. Ce modèle est surprenant par sa capacité d'apprentissage des règles rythmiques et de rime qui régissent le corpus. Malheureusement, la cohérence sémantique est bien moindre par rapport au modèle basé sur les caractères.

Enfin, Juntao Li (2018) [10] reprend l'idée de génération de poèmes classiques chinois et perfectionne l'architecture encodeur-décodeur en utilisant un des plus grands succès récents de l'IA : un auto-encodeur variationnel. Les encodeurs-décodeurs ont un défaut assez gênant : la représentation vectorielle donnée en entrée du RNN générateur a été produite par un RNN qui cherche uniquement à minimiser une fonction de perte. Si théo-

riquement cette représentation est censée contenir l'information sémantique du poème à générer, elle n'est pas régulière et la minimisation de la perte peut amener à du sur-entraînement. Les auto-encodeurs variationnels sont donc des encodeurs-décodeurs où la représentation vectorielle des mots clés donnés en entrée est régularisée pour qu'elle contienne des propriétés propices à la génération. Effectivement la génération de poésie ne peut pas être évaluée simplement, une sortie avec une perte miniscule à la fin de l'entraînement d'un réseau de neurones peut s'avérer bien pire qu'une sortie avec un entraînement plus court et une perte plus conséquente. En plus de cet auto-encodeur variationnel, Li rajoute un discriminateur à la suite du réseau pour induire un apprentissage de la cohésion thématique : le but du discriminateur est de reconnaitre le vers généré par rapport aux mots-clés rentrés par l'utilisateur (le titre du poème) et le vers précédent, et s'il réussit à bien reconnaître qui est qui, les résultats de son évaluation sont pris en compte par le décodeur pour qu'il puisse réussir à flouer le discriminateur dans le futur. Selon l'évaluation de Li, cette méthode est bien meilleure que les précédentes pour la génération de poèmes classiques chinois ! Pour la génération en français, Tim Van de Cruys (2019) [11] utilise des GRUs (Gated Recurrent Units), réseaux proches des LSTM et de performance comparable, et prédit, avec une architecture encodeur-décodeur également, à partir d'un vers le vers suivant en partant de la fin du vers, donc de la rime. Une particularité de son réseau est qu'il s'entraine sur un corpus générique : le modèle basé sur les caractères de Hopkins et Kiela pouvait également s'entraîner sur un large corpus mais restait un corpus poétique, Van de Cruys utilise un corpus bien plus large ; L'aspect poétique est contraint par des distributions de probabilités à priori sur la sortie du réseau de neurones.

ACTE II : Implémentation du Modèle

Scène I : Corpus et prétraitement

Notre modèle est un simple LSTM entraîné sur un corpus de pièces de théâtre faisant partie de Dramacode [2] un projet regroupant un grand nombre de pièces de théatres Françaises dans plusieurs formats. Notre corpus contient les pièces dramatiques et comiques de Racine, Pierre et Thomas Corneille, Molière et Voltaire, principalement composées d'alexandrins, et relativement proches en terme de période poétique. Le principe étant de laisser le réseau apprendre le maximum de contraintes simplement à partir du corpus composé uniquement d'alexandrins, une première tâche consiste donc à nettoyer le corpus pour n'obtenir que des alexandrins corrects, cette forme étant notre cible de génération. Nous avons donc téléchargé toutes les pièces convenables des auteurs sus-cités au format texte, et nous les avons passées au crible grâce à des scripts Python afin de supprimer les lignes vides, les vers qui n'était pas des alexandrins ou qui ne respectaient pas le schéma de rimes plates - la pièce peut contenir des lectures de lettres en prose, des chants, des passages en décasyllabe, du vieux français dans le cas de Thomas Corneille, ou encore un personnage qui s'exprime avec un accent qui est retranscrit dans le texte. Notre modèle s'entraîne au niveau du mot, pour pouvoir profiter de la dépendance des LSTM par

2. `http://dramacode.github.io/`

rapport à quelques dizaines de mots précédant le dernier mot prédit par le générateur, pour pouvoir garder une certaine cohérence thématique. Ainsi, pour fournir au modèle de bons mots, nous devons les séparer par des espaces. Les ponctuations sont donc considérées comme des mots à part, sauf dans des cas précis : l'apostrophe est collée au mot la précédant, pour éviter que le modèle génère des apostrophes solitaires et augmente la taille du vocabulaire en prenant en compte des mots comme "l" ou "j" qui existent uniquement dans le cas d'une apostrophe. La syntaxe inversée en fin de vers est aussi un cas important : si l'on considère que "deviens-je" doit être séparé en trois mots différents, le mot rime du vers sera "je", qui peut donc rimer avec "dussé-je". C'est pourquoi dans le cas d'une syntaxe inversée en fin de vers, on considère le tout comme un mot.

Pour faire apprendre la rime au modèle, nous devons insérer des balises de catégorie de rime à chaque vers (voir ACTE III pour la nécessité d'une telle démarche). Nous devons donc récupérer à partir du corpus toutes les catégories de rime possible, en conservant la distinction entre rimes masculines et féminines, ainsi que singulières et plurielles. Pour cela, nous partons du travail de Valérie Beaudouin dans sa thèse [12] et utilisons donc les catégories de rimes répertoriées dans une partie de notre corpus. Nous étendons donc ces catégories pour inclure l'entièreté du corpus. Nous utilisons le langage Python pour coder notre prétraitement et notre modèle, à l'aide de la librairie Pytorch. Nous utiliserons également Plint, codé par Antoine Amarilli, ainsi que la librairie fast-bleu pour l'évaluation.

Le corpus, composé de 158 320 alexandrins, est ensuite décomposé vers par vers. Chaque vers se voit attribuer un token de début (<sos>) et de fin (<eos>) de vers. Ensuite, pour chaque vers du corpus, nous plaçons une balise propre à la catégorie de rime du vers, juste avant le mot rime, cette balise étant le rimème de la catégorie, comme défini par Valérie Beaudouin. Ainsi, notre corpus est constitué d'une succession de distiques pour lesquels chaque vers est de la forme "<sos> première partie du vers avant le mot rime <balise de rime> mot-rime <eos>", par exemple : "<sos> Impatients désirs d' une illustre -ans[e] vengeance <eos>". Enfin, on sépare le corpus en un ensemble de séquences d'entrée X et un ensemble de séquences de sortie Y. Ces deux types de séquences ne diffèrent que de deux mots : Y est décalée de un mot par rapport à X. Le but est pour le réseau de prédire la séquence de sortie à partir de la séquence d'entrée, mot par mot. Pour le LSTM, la séquence d'entrée est vue comme une séquence temporelle dont il faut prédire l'état suivant à chaque étape.

Scène II : Le modèle

Comme on ne peut pas faire de calculs sur des mots, à chaque mot est associé un nombre unique, un indice. Au final, on se retrouve avec un vocabulaire de 28918 mots, à qui est donc associé un nombre entre 0 et 28917. La méthode classique des modèles de langue voudrait que pour les opérations futures, le mot soit représenté comme un vecteur de la taille du vocabulaire, contenant des zéros partout sauf à la position de l'indice du mot, où se trouverait un 1. Cette méthode s'appelle le one-hot-encoding. Or avec la taille de notre vocabulaire, malheureusement obligatoirement grande puisque l'on veut générer de la poésie, faire des opérations sur de tels vecteurs prendrait beaucoup trop de place

en mémoire. De plus, ainsi codés, les mots perdent leur relation entre eux et le modèle ne peut pas apprendre de ces relations dans le corpus. On utilise alors le word-embedding.

Le word-embedding consiste à réduire chaque mot en un vecteur à valeurs réelles, de dimension bien plus petite que les vecteurs one-hot-encodés. De plus, lorsque deux mots ont des vecteurs de word-embedding quasiment similaires, cela signifie qu'ils sont proches d'une manière ou d'une autre dans le corpus. Ainsi, le word-embedding conserve les relations observées entre les mots du corpus, et permet de passer outre les problèmes de mémoire causés par le one-hot-encoding. Ce word-embedding fait partie de notre réseau de neurones, il en est la première couche. Il s'entraîne sur le corpus pour pouvoir proposer de meilleurs vecteurs qui caractérisent bien les mots.

Après avoir transformé les mots en leurs indices, et découpé cette longue suite de nombre en X et Y, nous divisons encore ces séquences en trois ensembles : un ensemble d'entraînement, sur lequel le modèle va s'entraîner et progresser, un ensemble de validation, sur lequel le modèle va prédire les séquences de sortie par rapport aux séquences d'entrées et qui va permettre de voir si le modèle arrive à bien prédire sur un ensemble nouveau, et réorienter le réseau si ce n'est pas le cas, et enfin un ensemble de test qui va servir à la fin de l'entraînement en montrant la perte finale obtenue après que le réseau se soit bien entrainé.

Les séquences d'entraînement vont passer mot par mot dans le modèle : un mot sera embeddé, donc réduit en un vecteur de dimension 500, puis passera par deux couches LSTM de dimension 1024. Les couches LSTM sont composées de plusieurs portes, et peuvent choisir d'oublier le mot, de le garder en mémoire, ou bien d'écrire soit leur nouvel état caché, soit l'état de sortie de la couche. Elles ont donc une sorte de mémoire courte dans laquelle elle peuvent garder un mot, mais ont aussi des paramètres qui dépendent de tous les mots qui passent dans la cellule, jusqu'à une certaine limite, car sinon les calculs de gradients deviennent longs et l'entraînement s'éternise. Tous les mots de la séquence passent par ces couches, et l'état caché résultant de ce passage passe dans une couche dense qui va s'occuper de la classification et retourner un vecteur de la taille du vocabulaire. Une fois une fonction softmax appliquée à ce vecteur, il contient la probabilité que le dernier mot de la séquence de sortie Y (et donc le mot à prédire suivant la séquence d'entraînement) soit le mot dont l'indice est aussi l'indice de la probabilité dans le vecteur. Une fonction de perte est calculée entre le vecteur de sortie du modèle et la séquence de sortie. En calculant le gradient de cette fonction, le modèle met à jour ses paramètres (pour l'embedding, pour le fonctionnement des couches LSTM et pour la classification), le gradient permettant de savoir quels paramètres sont responsables de la perte et comment les changer pour la diminuer. En itérant sur tout le corpus d'entraînement plusieurs fois, les paramètres du modèle diminuent la perte, le modèle prédit mieux la sortie.

Scène III : La génération

Une fois le modèle entraîné, il est temps de passer à la génération. Pour cela, on donne au modèle une séquence d'entrée. Ce peut être un, deux ou trois vers. On commence systématiquement la génération par un token de début de vers, il faut donc que

la séquence initiale finisse par un token de fin de vers. On utilise trois méthodes de génération différentes dont on va comparer les résultats. D'abord, l'approche la plus simple est dite "gloutonne" : elle consiste a prendre dans le vecteur de probabilités le mot le plus probable et le rajouter à la séquence d'entrée et avancer ainsi de suite jusqu'à avoir un poème généré de la taille désirée par l'utilisateur. Malheureusement cette approche maximise la probabilité du mot suivant à chaque étape de la génération mot par mot, ce qui ne veut pas dire que la séquence générée est la plus probable. La deuxième méthode, appelée beam-search, résout ce problème : une fois la séquence d'entrée donnée au modèle, au lieu de garder uniquement le mot le plus probable, on garde les k mots les plus probables. On a donc dix "branches" (voir Figure 1 [3]) possibles. Ensuite, on applique le modèle pour chacun de ces dix mots (sur chaque branche), et l'on multiplie le vecteur de probabilité obtenu sur chacune des branches par la probabilité d'avoir obtenu le mot précédent : ainsi on a la probabilité de la séquence de mots générés, et c'est cette probabilité qu'on cherche à maximiser. Parmi toutes les possibilités sur les dix branches on choisit les 10 chemins qui ont la plus forte probabilité de séquence. On procède ainsi jusqu'à la longueur voulue. Au final, la branche choisie est celle qui a la plus grande probabilité parmi les dix branches que l'on a conservé. Ce n'est peut-être pas à la séquence avec la plus grande probabilité, mais avec un k grand, on peut s'en approcher et le résultat est meilleur que pour l'approche gloutonne.

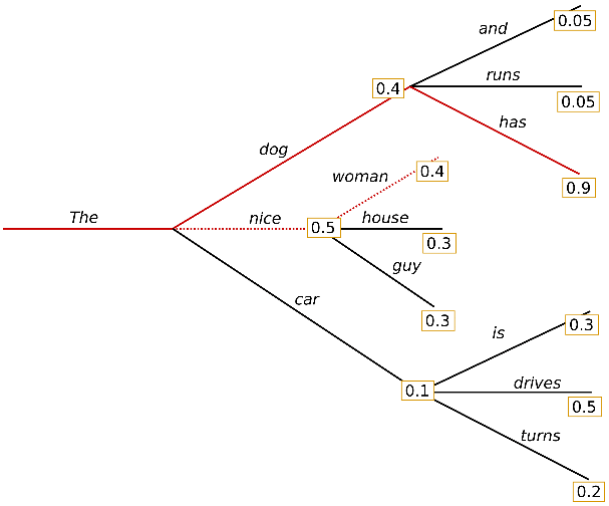

FIGURE 1 – Exemple d'un beam-search : même si le mot le plus probable après "The" est "nice", on voit que "The dog has" est plus probable que "The nice woman", résultat obtenu avec l'approche gloutonne. Avec un beam search de taille 2, on conserve "dog" et "nice" comme branches, et ensuite "has" et "woman" qui maximisent la probabilité des branches après deux pas de temps.

Les dernières méthodes sont des méthodes d'échantillonnage, ou sampling. Nous présentons premièrement le top-k sampling. Cette méthode rend possible des choix de mots qui ne le seraient pas avec le beam-search, ce qui est voulu en poésie : en effet certains mots peuvent être peu probables car ils sont moins présents au long du corpus, mais ils sont essentiels à l'aspect poétique. Ces mots feront baisser la probabilité d'une branche

3. Toutes les illustrations des méthodes de génération sont tirées de cette page web : `https://huggingface.co/blog/how-to-generate`

par rapport à une autre branche qui prend moins de risque, et ne sera donc pas choisie parmi les k branches les plus probables, k n'excédant généralement pas 10 dans les approches par beam-search courantes. Le top-k sampling consiste à ne garder du vecteur de probabilités que les k mots les plus probables (avec cette fois un k plus grand, de l'ordre de 50 ou 100) et sur cette distribution de probabilités tronquées, réaliser un échantillonnage aléatoire pour choisir le mot suivant la séquence d'entrée.

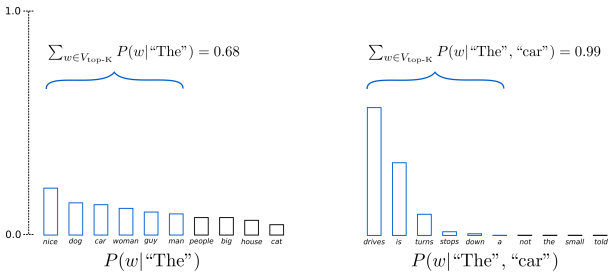

FIGURE 2 – Exemple de top-k sampling : on ne garde ici que les 6 mots les plus probables et l'on réalise un échantillonnage sur ces six mots. On voit que lors de l'échantillonnage, on a plus de chance de tirer "nice" que "man", et après avoir tiré "car" on voit que la distribution de probabilité peut être très inégale : peu de mots sont très probables et le reste est très peu probable. Néanmoins le top-k sampling laisse une chance de tirer "stops" après la séquence "The car", ce qui n'est pas aberrant mais est peu probable : dans un beam-search de taille 2, il n'aurait pas pu être choisi par exemple.

Le top-k sampling possède un inconvénient majeur : comme on peut le voir dans la Figure 2, la distribution de gauche est assez plate, en choisissant les 6 premiers mots, on exclut par exemple "people", qui est pourtant un mot très probable après le contexte "The". À l'inverse, dans la distribution de droite, on conserve le mot "a", très peu probable après "The car", pour l'échantillonnage. On aimerait donc pouvoir adapter le nombre de mots choisis pour l'échantillonnage en fonction de la distribution de probabilité. C'est là l'idée du top-p sampling ou nucleus sampling. Le paramètre p est une probabilité seuil, c'est à dire que l'on conserve les mots les plus probables pour l'échantillonnage jusqu'à ce que leur probabilité cumulée dépasse p. On fixe usuellement p entre 0.9 et 1.

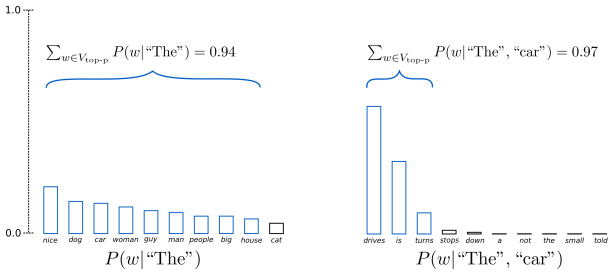

FIGURE 3 – Exemple de top-p sampling : on ne garde ici que les mots les plus probables dont la probabilité cumulée dépasse p = 0.92 et l'on réalise un échantillonnage sur ces mots. On a bien réglé l'inconvénient du top-k sampling : après "The", on a plus de possibilités pour l'échantillonnage, la génération s'en trouvera plus diversifiée. On évite également de choisir des mots trop peu probables lorsque la distribution est piquée comme à droite, tout en laissant sa chance à un mot comme "stops", peu probable mais acceptable.

On introduit dans le beam-search et les méthodes de sampling une *température*. Cette température est un nombre positif qui va plus ou moins modifier la distribution de probabilités donnée en sortie par le modèle. Elle va accentuer les probabilités fortes et diminuer les probabilités faibles si elle est basse (de 0 à 1) et les réduire si elle est haute (au delà de 1). Ainsi, le modèle sera plus sûr de lui avec une température basse, son choix sera plus marqué et sûr. Mais en contrepartie, il va prendre moins de risque et renvoyer une sortie très proche de ce qu'il connaît, des textes sur lesquels il s'est entraîné. Avec une haute température, le modèle a plus de mal à faire un choix et il prend des risques : cela peut résulter en une sortie moins cohérente et plus farfelue, mais souvent plus surprenante et poétique.

ACTE III : Résultats et Évaluation

Scène I : Focus sur la métrique

Dans un premier temps, nous voulions observer les résultats du modèle sans aucune contrainte : pas de balises pour indiquer la rime, pas de règle de génération forcée. Le modèle, bien que sommaire, fut surprenant dans son apprentissage de la métrique. Pour les premiers résultats observés, le corpus ne contenait pas uniquement des alexandrins car dans certaines pièces étaient présents des chants ou des lectures de lettres en prose ou en d'autres structures de vers. La grande majorité des vers générés par toutes les méthodes, et pour toutes les valeurs de température, sont métriquement corrects : ils contiennent douze syllabes, et une césure à l'hémistiche. Après le nettoyage du corpus, nous avons relancé la génération et avons obtenu plusieurs résultats. En voici quelques uns :

Et si tu ne le peux, tu m'en souviens assez.
Je n'ai pas le loisir de me voir dans mon âme,

Et ce n'est pas pour moi qu'il faut que je m'explique.
Je n'ai plus rien à dire, il n'en est pas besoin.

Mais je te en réponds, si tu ne fais courir,
Que c'est pour t'affranchir de toute ta vertu.

J'ai su par ce faux bruit de quoi se hasarder,
Et que je dois aimer mon sang ... Mais que dis-tu ?

Ces distiques ont été généré respectivement par une méthode gloutonne, par un beam search de taille 20 et de température 1, un top-k sampling de taille 100 et de température 0.7, et un top-p sampling avec p = 0.7 et de température 1 à partir de l'entrée suivante :
Je ne t'en parle plus, va, sers la tyrannie,
Abandonne ton âme à son lâche génie ;

On voit que ces distiques tiennent compte du subjonctif présent dans l'entrée et le poursuivent. On remarque que malheureusement, si la métrique est bien respectée, la rime ne l'est pas du tout. On remarque également un manque de cohérence sémantique, et quelques fois de rigueur syntaxique dans les méthodes de sampling : le bigramme

Évaluation de vers sans rime		
Méthode de génération	SelfBLEU-4	% d'erreur métrique (Plint)
glouton	0.93	0.9
beam, k=20, t=1	0.97	0.6
top-k, k=100, t=0.7	0.39	11
top-p, p=0.7, t=1	0.33	14

TABLE 1 – Le selfBLEU humain est d'environ 0.31, c'est donc le top-p sampling qui s'en rapproche le plus. Au niveau de l'erreur métrique, les résultats sont attendus : avec une forte utilisation de mots courts et beaucoup de répétition, les méthodes gloutonne et beam se trompent peu souvent sur la métrique. Les méthodes de sampling sont moins fiables sur ce point.

"te en" n'est présent à aucun moment dans le corpus et est pourtant échantillonné. On remarque que des mots très courts sont souvent choisis par le modèle. C'est assez logique puisque le modèle favorise les mots fréquents et a tendance à les utiliser à outrance, leur attribuant une forte probabilité. Toutefois les résultats des méthodes de sampling sont intéressants par leur originalité : les mots "maux", "hasarder", et "affranchir" sont peu probables et ont pourtant une chance d'être représentés, ce qui est souhaitable. La sortie du top-p sampling est spéciale : le découpage du deuxième vers n'est pas classique, ce qui est rafraichissant. Dans l'évaluation, il est important de noter que les méthodes gloutonne et beam-search donnent des résultats extrêmement répétitifs.

Nous avons utilisé plusieurs métriques pour évaluer les vers générés par les différentes méthodes. Nous avons généré 100 séquences de 200 mots pour chacune des quatre méthodes différentes, suivant un distique différent en entrée pour chacune des 100 générations. Sur cet ensemble, nous pouvons calculer le score SelfBLEU [4] sur les quadrigrammes de chaque génération par rapport aux 99 autres, dont on calcule la moyenne. Cette métrique donne une idée de la similarité entre les générations, plus le SelfBLEU est faible, plus la méthode de génération donne des résultats variés. Nous avons également utilisé le programme Plint pour établir la proportion de vers métriquement corrects parmi les 100 générations.

Dans la Table 1, le score selfBLEU sur les quadrigrammes des deux premières méthodes nous permet de nous rendre compte de la similarité très forte entre les générations au contexte pourtant différent : en effet le selfBLEU diminue lorsqu'un quadrigramme n'est pas présent dans les autres générations. Un selfBLEU aussi élevé est bien témoin de la répétition et de l'inoriginalité des de ces méthodes. On remarque qu'on a dans le pire des cas une précision de 86% sur la métrique selon Plint, ce qui est remarquable étant donné que le modèle n'a aucune restriction quant à sa forme de génération. Il est maintenant temps de s'attaquer à la rime.

4. Défini par Zu et al.`https://arxiv.org/pdf/1802.01886.pdf`

Scène II : La difficulté de rimer

Si la métrique semble être apprise aisément par le modèle, la rime lui pose problème. Dans les résultat précédent, une rime fait figure d'exception et est clairement aléatoire. Il nous fallait alors rajouter une contrainte sur la rime que le modèle pourrait apprendre.

Notre première approche a donc été de modifier les balises de fin de vers. Au lieu de faire terminer chaque vers par un symbole <eos>, nous voulons le faire terminer par un symbole <eosK>, où K est un nombre associé à la catégorie de rime du vers. Il existe 542 catégories de rimes possibles, dépendant de la graphie et de la phonétique du dernier mot d'un vers. Nous pensions qu'avec cet ajout le modèle serait capable de comprendre que les deux vers d'un distique finissent toujours par des mots de la même catégorie, et pourrait donc générer des rimes. En entraînant le modèle sur un nombre limité de catégories les plus fréquentes -et donc en supprimant du corpus les vers appartenant à des catégories peu fréquentes, celui-ci n'apprend toujours pas à rimer de manière significative. Néanmoins on obtient parfois des distiques contenant des mots rime appartenant à la même catégorie, comme le distique suivant :

Si je suis innocent, c'est pour me faire roi ; <eos6>
Je n'ai point à regret que ce coeur n'est plus à moi. <eos6>

On remarque en revanche que la métrique n'est pas correcte sur le deuxième vers.

Une intuition fut alors de déplacer la balise de rime dans le vers. En réinstaurant la balise de fin de vers <eos>, nous avons voulu déplacer la balise de rime juste après le mot-rime, en espérant que l'incapacité du modèle à apprendre la rime était due à la présence de ponctuation entre le mot-rime et la balise, pour avoir des vers de ce type : "<sos> Impatients désirs d' une illustre vengeance -ans[e] <eos>". Cependant, une erreur d'indice a décalé la place de la balise avant le mot-rime, les vers du corpus étaient donc de cette forme : "<sos> Impatients désirs d' une illustre -ans[e] vengeance <eos>". Curieux d'observer les résultats avec de tels vers,nous étions surpris de voir descendre la valeur de la perte de validation lors de l'entraînement : en effet cette perte qui restait proche 3.5 jusqu'alors, descendait jusqu'à 3.1. Plus incroyable encore, la rime semblait beaucoup plus fréquente dans les résultats ! En essayant d'implémenter la forme de vers voulue, avec la balise de rime après le mot-rime, la perte descend également (vers 3.2) mais la rime est aussi fréquente que lorsque la balise était en bout de vers. Pas de doute, c'est bien ce qui au départ était une erreur qui a débloqué le modèle pour apprendre à rimer !

En regardant la Table 2, on voit qu'introduire la rime dans le modèle n'est pas sans effet : la justesse de la métrique en est amoindrie, surtout dans le cas des méthodes d'échantillonnage. Pour ce qui est du pourcentage de rime, les méthodes gloutonne et beam sont avantagées par la répétition qui les caractérisent. Toutefois, des rimes présentes sur les trois quarts des générations à partir d'échantillonnage sont un résultat satisfaisant. Lorsque l'on réfléchit à la manière dont un réseau LSTM travaille, en lisant les séquences d'entrée de gauche à droite, le fait de mettre la balise de rime avant le mot-rime semble être effectivement un bon choix, le réseau apprenant quels mots placer après une telle balise. Dans l'autre sens, bien que deux balises successives soient toujours les mêmes, le LSTM a du mal à faire prévaloir cette information par rapport au contexte court avant

Évaluation de vers avec rime			
Méthode de génération	SelfBLEU-4	% d'erreur métrique (Plint)	% de rime
glouton	0.96	3.6	93
beam, k=20, t=1	0.98	2.3	83
top-k, k=100, t=0.7	0.46	30	74
top-p, p=0.7, t=1	0.39	31	74

TABLE 2 – On remarque une augmentation du selfBLEU pour toutes les méthodes : la contrainte de rime induit moins de diversité entre les poèmes. On remarque également une augmentation du pourcentage d'erreur métrique, surtout pour le sampling : le modèle porte plus d'importance à la justesse du mot-rime qu'à la justesse syllabique. Les pourcentages de rimes sont assez satisfaisants en sachant que la contrainte est très légère.

Erreurs syntaxiques et sémantiques				
Méthode de génération	Syntaxe	Sémantique	Nb de vers différents	Nb de vers
glouton	7	2	26	115
beam, k=20, t=1	4	4	29	110
top-k, k=100, t=0.7	29	67	127	127
top-p, p=0.7, t=1	41	80	133	133

TABLE 3 – Compte des vers comportant des erreurs syntaxiques et sémantiques, et compte de la diversité. Si les deux premières méthodes sont les plus performantes en syntaxe et en sémantique, elles sont bien moins diverses que les deux dernières.

la balise (le mot-rime et la ponctuation) qui change souvent. Nos vers sont désormais capables de rimer, mais une question subsiste : ont-ils du sens ?

Scène III : Évaluation humaine

Une réponse courte serait non. Effectivement, notre modèle produit pour les méthodes gloutonne et beam des vers très répétitifs qui ont du sens individuellement, mais très peu lorsqu'ils sont mis bout à bout. Inversement, les méthodes d'échantillonnage perdent en cohérence sémantique très rapidement, un vers pouvant n'avoir que très peu de sens, et manquent même quelque fois de rigueur syntaxique. Nous conduisons une analyse syntaxique et sémantique binaire sur plusieurs vers générés par les différentes méthodes présentées : si aucune erreur n'est présente dans le vers, on lui affecte la valeur 0. si une ou plusieurs erreurs sont détectées, on lui affecte la valeur 1. Le compte de ces tests est tenu dans la Table 3.

Ces derniers résultats sont peu encourageant pour ce qui est de la performance de notre modèle à générer du Français correct et plein de sens. Néanmoins, certaines générations sont parfois à la fois très poétiques mais également assez cohérentes. Nous avons donc mis en place un questionnaire dans lequel nous avons glissé quatre distique humains, ainsi qu'un distique généré par chaque méthode présentée. Les participants doivent deviner si chaque distique a été écrit par un humain ou par une machine. Les résultats sont présentés en Table 4.

Classification de distiques : humain ou machine	
Méthode de génération	% de participants choisissant "humain"
glouton : Je vous l'ai déjà dit, je vous l'ai dit, Madame, / Ce que j'ai fait pour vous le prix de votre flamme,	50
beam : Et je ne puis souffrir que par un juste effort / Je n'ai pu l'acquérir, je n'ai plus de rapport.	25
top-k : Et je laisse l'hymen d'un si cruel devoir / Qu'un éclat de fierté me faisait recevoir.	50
top-p : Que je sois criminel ! Ma sœur et l'injustice / Aux tendresses du ciel je m'ôte Bérénice,	16,7

TABLE 4 – Sur un total de 12 participants, la moitié a été flouée par l'approche gloutonne, très cohérente, mais aussi par le top-k sampling, moins cohérent mais bénéficiant d'un caractère poétique prononcé.

Conclusion et perspectives d'amélioration

Nous avons vu les différents travaux qui ont structuré le domaine de la génération de la poésie, et nous avons présenté notre approche, qui était de partir avec un LSTM simple et de voir s'il pouvait apprendre un maximum de contraintes, seulement à partir d'un corpus. Les résultats directs sur la métrique ont été très encourageants, et une simple contrainte sur la rime a été suffisante pour que les vers générés correspondent dans une grande partie des cas à des alexandrins classiques. Finalement, si les premières approches de génération sont très répétitives, elles peuvent au moins créer un distique cohérent et poétique, et d'un autre côté, l'échantillonnage, lorsqu'il est grammaticalement correct et a un minimum de sens, permet de créer des compositions originales et belles. Une grande perspective d'amélioration peut se faire sur la cohérence sémantique et syntaxique du modèle. Une idée serait d'utiliser des modèles de langage à cache, c'est à dire des modèles qui gardent en mémoire les prédictions récentes pour s'améliorer lors de leur prochaine prédiction. Nous pouvons aussi imaginer utiliser un modificateur de probabilités qui pénalise les mots courts et répétitifs pour pouvoir créer plus de diversité dans des méthodes comme la beam search, ou bien récompenser les mots qui forment une structure syntaxique correcte dans le cas du sampling.

Références

[1] R. Queneau, *Cent Mille Milliards de Poèmes*. Paris, France : Gallimard, 1961.

[2] H. Manurung, G. Ritchie, and H. S. Thompson, "Towards A Computational Model Of Poetry Generation," 2000.

[3] P. Gervás, "An expert system for the composition of formal Spanish poetry," *Knowledge-Based Systems*, vol. 14, no. 3-4, pp. 181–188, 2001.

[4] H. Gonçalo Oliveira, "PoeTryMe : a versatile platform for poetry generation," vol. 1, article 21, 08 2012.

[5] X. Zhang and M. Lapata, "Chinese poetry generation with recurrent neural networks," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, (Doha, Qatar), pp. 670–680, Association for Computational Linguistics, Oct. 2014.

[6] R. Yan, "i, poet : Automatic poetry composition through recurrent neural networks with iterative polishing schema," in *IJCAI*, 2016.

[7] Q. Wang, T. Luo, D. Wang, and C. Xing, "Chinese song iambics generation with neural attention-based model," in *IJCAI*, 2016.

[8] M. Ghazvininejad, X. Shi, Y. Choi, and K. Knight, "Generating topical poetry," in *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, (Austin, Texas), pp. 1183–1191, Association for Computational Linguistics, Nov. 2016.

[9] J. Hopkins and D. Kiela, "Automatically generating rhythmic verse with neural networks," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1 : Long Papers)*, (Vancouver, Canada), pp. 168–178, Association for Computational Linguistics, July 2017.

[10] J. Li, Y. Song, H. Zhang, D. Chen, S. Shi, D. Zhao, and R. Yan, "Generating classical Chinese poems via conditional variational autoencoder and adversarial training," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, (Brussels, Belgium), pp. 3890–3900, Association for Computational Linguistics, Oct.-Nov. 2018.

[11] T. Van De Cruys, "La génération automatique de poésie en français," in *Conférence sur le Traitement Automatique des Langues Naturelles (TALN-RECITAL)*, (Toulouse, France), July 2019.

[12] V. Beaudouin, " Rythme et rime de l'alexandrin classique. Étude empirique des 80 000 vers du théâtre de Corneille et Racine," (Ecole des Hautes Etudes en Sciences Sociales (EHESS), France), Jan. 2000.