Adnene KHALBOUS
Ariel NATAF
Alexis SAVVA
Jean-Baptiste CHANIER

Vendredi 17 Décembre 2021

Master 2 Machine Learning pour la Science des Données
Université de Paris

Projet MLSD : Deep Learning

À l'attention de Mr Falissard

Exercice 1 : Analyse Exploratoire

Pour ce projet, nous disposons de trois jeux de données, correspondant respectivement aux données d'entrainement, de validation et de test. Chaque observation est composée d'un énoncé de pathologie, tandis que la variable à prédire est le code correspondant dans la classification médicale CIM-10.

Au total, les jeux de données comptent respectivement 181763, 30000 et 30000 observations.

On constate qu'il existe 9248 CIM différents dans le jeu de données train.

Le label le plus fréquent, « S062 », qui correspond à la « Lésion traumatique cérébrale diffuse » est apparu 1131 fois.

Il apparait que le jeu de données est déséquilibré : Il n'y a pas autant d'exemples pour chaque code CIM. Ce phénomène semble naturel puisque le jeu de données correspond à des cas réels et que certaines pathologies sont plus fréquemment rencontrées que d'autres.

Ceci peut être illustré par le graphique suivant, qui décrit la répartition des effectifs dans pour les données d'entrainement :

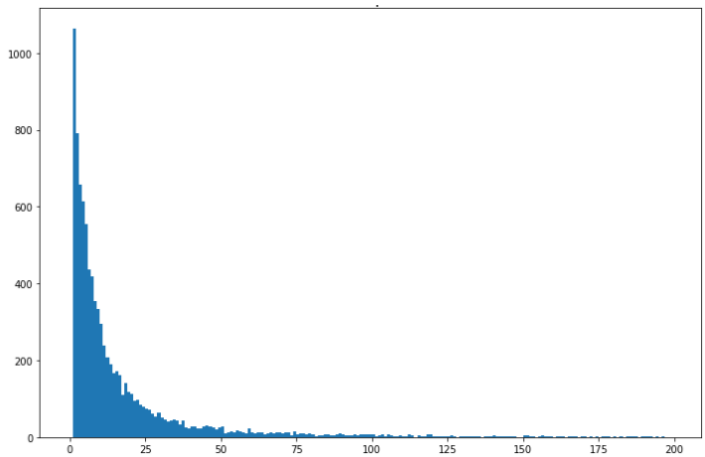


Figure 1 : Répartition des effectifs des labels

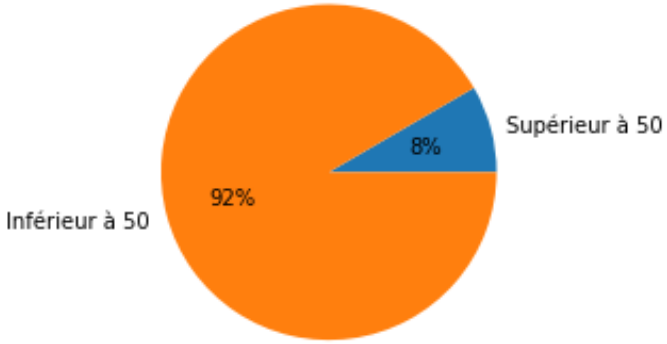


Figure 2 : Effectifs supérieurs et inférieurs à 50 individus

Les deux tiers des labels CIM étudiés sont apparus moins de 10 fois dans le jeu de données d'entrainement.

Ce déséquilibre est important à prendre en compte dans le choix de nos métriques qui nous permettront d'évaluer la qualité de nos modèles : Par exemple, l'exactitude, qui est fréquemment utilisée pour des problèmes de classification, n'est pas complètement fiable si le jeu de données est peu équilibré. En effet, imaginons un problème simple avec 2 classes, ou l'une des classes représente 95% des individus en présence. Dans ce cas, un modèle qui n'apprendrait à prédire seulement ce label obtiendrait 95% d'exactitude ! Et ce, tout en étant littéralement le pire discriminateur possible.

Nous avons choisi de conserver l'exactitude comme métrique d'évaluation puisque le nombre de classes est important, ce qui permet d'éviter la situation précédemment explicitée. Nous avons choisi également d'évaluer nos modèles au regard du F1-Score pondéré par les poids de chaque classes au regard de son effectif.

Enfin, les textes rassemblent au total 35602 tokens différents.

Exercice 2 : Prédiction de la première lettre de l'étiquette

Dans cette partie, nous chercherons à résoudre une version simplifiée du problème, soit la prédiction de la première lettre uniquement du code CIM-10 du texte considéré. Nous tirons ainsi parti du caractère hiérarchique de l'étiquette, c'est-à-dire le fait que dans chaque caractère du label possède sa propre signification : Par exemple, deux étiquettes dont les trois premiers caractères sont communs désigneront dans de nombreux cas des pathologies proches.

En construisant des nouveaux jeux de données après modification des labels, nous sommes en mesure de présenter les analyses suivantes sur les classes à prédire. Les quatres classes les plus présentes dans le jeu de données d'entrainement, leurs effectifs et leur significations médicales sont les suivantes :

1. **« C »** : 21 461, Tumeurs
2. **« I »** : 18 153, Maladies du système circulatoire
3. **« T » , « S »** : 11 258, 10 820 Blessures, intoxications et certaines autres conséquences de causes externes
4. **« Q »** : 10 113 Malformations congénitales, déformations et anomalies chromosomiques

Si l'on prend toutes les lettres du jeu d'entrainement en compte, la répartition des effectifs de chaque classe est la suivante :

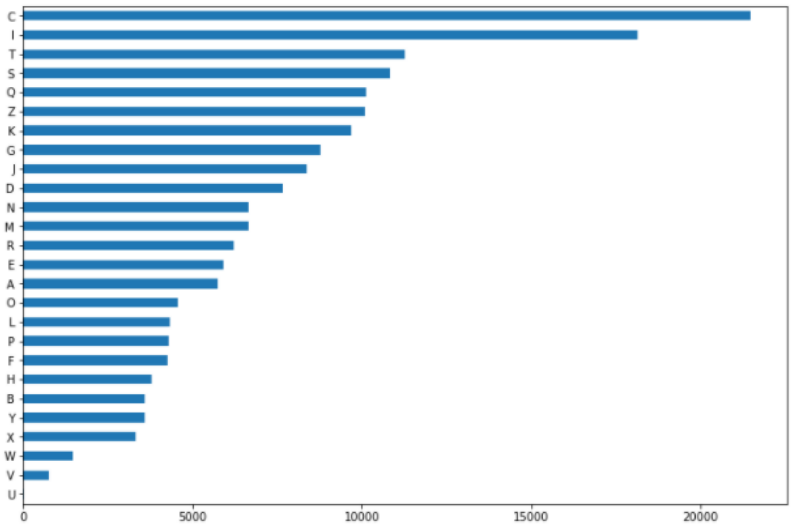


Figure 3 : Prédiction de la première lettre

Pour cette partie, comme chaque observation prend la forme d'une suite de mots, nous avons utilisé une architecture centrée autour des couches de neurones « Long-Short Term Memory » (LSTM), qui sont à même de considérer le caractère séquentiel des données.

Pour être à même de prendre en compte des données textuelles, nous avons dû mettre chaque token pris rencontré dans le jeu de données d'entraînement sous la forme d'un vecteur. Cette opération, appelée « Embedding », a été rendu possible par la combinaison d'une couche « TextVectorization » et d'une couche d'embedding en entrée du modèle. Ainsi, chaque token a correctement été projeté dans un espace de grande dimension. Pour les tokens inconnus, soient les tokens présents uniquement dans le jeu de validation et de test, nous avons ajouté un système basé sur la présence d'un token « Unknown », qui permet de couvrir ce cas. La prédiction est opérée avec la combinaison d'une couche dense et d'une fonction d'activation de type softmax.

Le choix des hyperparamètres du modèle :

- Nombre d'epochs : 10
- Taille des batchs : 64
- Loss : Entropie croisée catégorielle
- Optimiseur : Adam
- Dropout : 0.1
- Nombre de neurones par couche : 50 (LSTM), 26 (Dense)
- Dimension de l'embedding : 100

L'optimiseur Adam est une référence en deep learning, correspondant à une combinaison entre les algorithmes RMSprop et du moment, qui permettent d'ajuster la mise à jour des poids du modèle en prenant en compte le calcul du gradient des poids à partir du batch étudié mais également les valeurs utilisées mises à jours précédentes. En tant qu'algorithme d'optimisation, son rôle est de permettre la limitation des oscillations dans des directions non-souhaitées et d'accélérer la descente de gradient dans la direction désirable.

Les résultats obtenus par le modèle à l'issue de son entrainement sont les suivants :

Métrique	Entraînement	Validation	Test
Exactitude	0.922	0.813	0.815
F1-Score	0.923	0.815	0.816

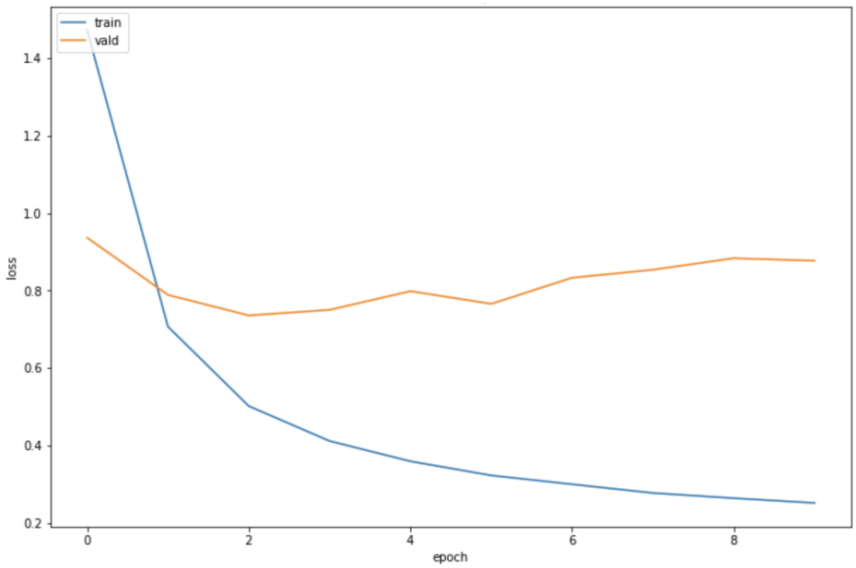


Figure 4 : Evolution des loss d'entrainement et de validation avec dropout = 0.1

En considérant l'évolution de la loss du modèle, nous avons choisi de conserver le modèle avec les poids disponibles à l'epoch 2, étant donné que c'est celle qui propose une valeur de loss de validation la plus faible. On remarque également que le surapprentissage est rapidement observé, ce qui signifie que l'architecture du modèle est trop complexe pour le problème étudié, et que le modèle est en capacité d'apprendre par cœur la distribution des données en seulement quelques epochs.

Les solutions dans ce cas sont donc d'augmenter la valeur de dropout (solution privilégiée de manière générale), ou d'ajouter des termes de régularisation des poids à la loss (régularisation L1 ou L2). Nous avons choisi ici d'augmenter le dropout, le plaçant à 0.3, ce qui signifie concrètement que pour chaque couche du modèle, 30% des neurones seront désactivés à tour de rôle pendant l'entrainement (contre 10% initialement).

Les résultats de ce second entraînement sont les suivants :

Métrique	Entraînement	Validation	Test
Exactitude	0.916	0.814	0.815
F1-Score	0.917	0.812	0.817

Les résultats sont très proches de ceux obtenus lors du premier entrainement, il est cependant à noter que les performances du modèle sont moins légèrement moins élevées sur le jeu d'entrainement, ce qui est une conséquence logique de l'augmentation du dropout puisque le modèle est volontairement rendu moins complexe avec ce jeu de données. Les résultats sur les jeux de validation et de test sont quasiment identiques car lors de l'inférence, tous les neurones sont réactivés, ce qui ramène à une situation identique à celle du premier modèle.

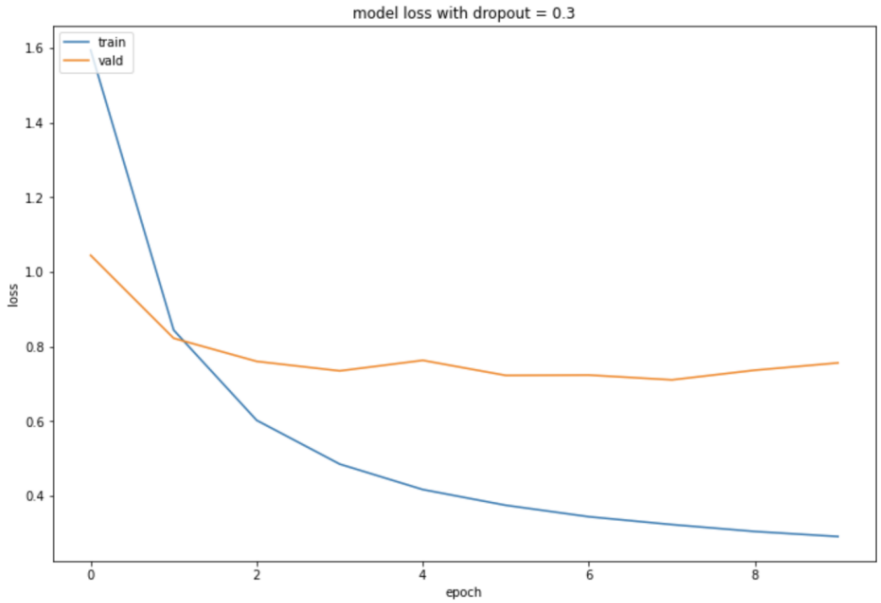


Figure 5 : Evolution des loss d'entrainement et de validation avec dropout = 0.3

L'effet du dropout se fait ressentir : On peut voir sur ce graphe que la loss de validation atteint son minimum à l'epoch 7 (et non plus l'epoch 2), nous avons réussi à repousser légèrement dans le temps le phénomène de surapprentissage.

Exercice 3 : Prédiction de l'étiquette entière

Dans cette partie nous ne cherchons plus à créer un modèle qui prédit la première lettre, mais les 4 (ou occasionnellement 3) caractères qui composent le code CIM10 du diagnostic en entré. Pour cela nous avons réalisé un modèle séquentiel avec un embedding, une couche LSTM, une couche de convolution à une dimension, un GlobalAveragePooling1D et une couche dense.

L'exactitude pour la première lettre est de 76% ; pour la deuxième lettre est de 70% ; pour la troisième de 64%.

La précision diminue plus on avance dans la séquence. On a de moins bons résultats pour le premier caractère. Nous passons de 81 % dans l'exercice précédent à 76% pour la prédiction du premier caractère. Ces résultats étaient prévisibles car la convolution réduit à la complexité des données et peut entrainer une perte d'information. Malgré cette performance inférieure, ce modèle permet d'avoir des résultats bien plus intéressants grâce à la possibilité de prédire les caractères suivants.

Dans cette partie, l'architecture du modèle est choisie comme similaire à celle utilisée lors de l'exercice 2, à l'exception de deux couches supplémentaires : La première est une couche de convolution en une dimension, en effet, si l'on représente chaque mot simplement par son indice dans le vocabulaire, la séquence textuelle peut être interprétée comme une matrice de taille 1*max_sequence_length. Pour cet exemple, nous avons choisi un nombre de filtres de 32. Le second

changement apporté à l'architecture du modèle est le passage de l'utilisation d'une couche LSTM simple à une couche LSTM bidirectionnelle. Concrètement, une couche bidirectionnelle permet à la fois de considérer le sens de lecture classique (Du mot le plus à gauche vers le mot le plus à droite) et le sens inverse (Du mot le plus à droite vers le mot le plus à gauche). Au niveau du calcul matriciel, il s'agit simplement de la concaténation de deux matrices, chacune correspondant au résultat de la couche dans un sens précis. Ainsi, lorsqu'un mot en particulier est considéré, le modèle a accès à la fois au traitement des mots précédents, mais également au traitement des mots suivants. L'utilisation des couches bidirectionnelles rend le modèle beaucoup plus lourd, et allonge le temps d'entraînement (qui est déjà considéré comme conséquent avec des LSTM unidirectionnelles). Ici, la couche bidirectionnelle compte 128 neurones. Ce nombre est particulièrement important mais correspond à l'augmentation de la complexité de la tâche à réaliser par le modèle.

Enfin, pour gérer les labels inconnus (présents dans les jeux de validation et de test mais non dans le jeu d'entraînement), nous avons choisi d'ajouter une étiquette <UNK> au dictionnaire des mots créé à partir du jeu d'entrainement. De cette manière, lorsque le modèle rencontre une étiquette qu'il ne connait pas, son traitement en reste toujours possible : elle sera recodée en étiquette inconnue. Nous avons préféré cette approche à celle qui consisterait à considérer avant l'entrainement l'ensemble des étquettes (labels de validation et de test compris), car notre méthode nous a semblé être plus proche d'un cas réel où il est tout à fait possible qu'un modèle ait à gérer des labels qu'il n'aurait jamais rencontré durant son entraînement.

Les résultats obtenus pour cet exercice sont les suivants :

Métrique	**Entraînement**	**Validation**	**Test**
Exactitude	0.809	0.553	0 .549
F1-Score	0.814	0.563	0.558

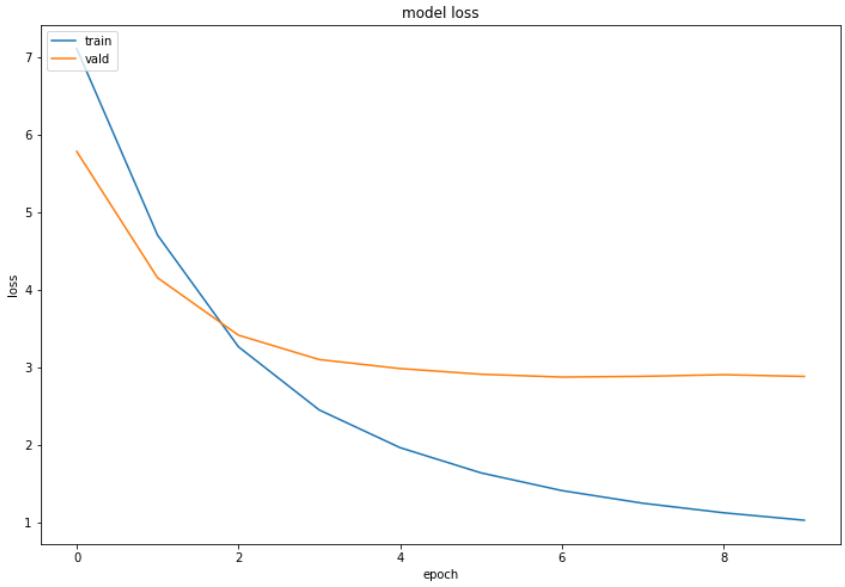


Figure 6 : Evolution des loss d'entrainement et de validation pour la prédiction de l'étiquette complète

On remarque que les exactictudes de validation et de test sont largement moins intéressantes que dans les exemples précédents (environ 55% contre 82% environ). Ceci est une conséquence directe de l'augmentation massive du nombre de classes à prédire, soit un passage de 26 classes à 9248.

En étudiant le graphique des loss du modèle, il est à noter que le phénomène de surapprentissage est évité lors de tout l'entraînement (pas d'augmentation de la loss de validation). Cependant, il pourrait être intéressant d'augmenter la complexité de l'architecture du modèle : En effet, un écart aussi important entre les exactitudes d'entrainement et de validation démontre que toute la complexité des données est loin d'avoir été saisie par le modèle.

Enfin, en addition des prédictions de la première lettre et de l'étiquette complète, nous avons également lancé la procédure de prédiction sur des labels réduits à leurs deux, puis trois premiers caractères. Les résultats en accuracy sur les données de test sont synthétisés dans le tableau suivant :

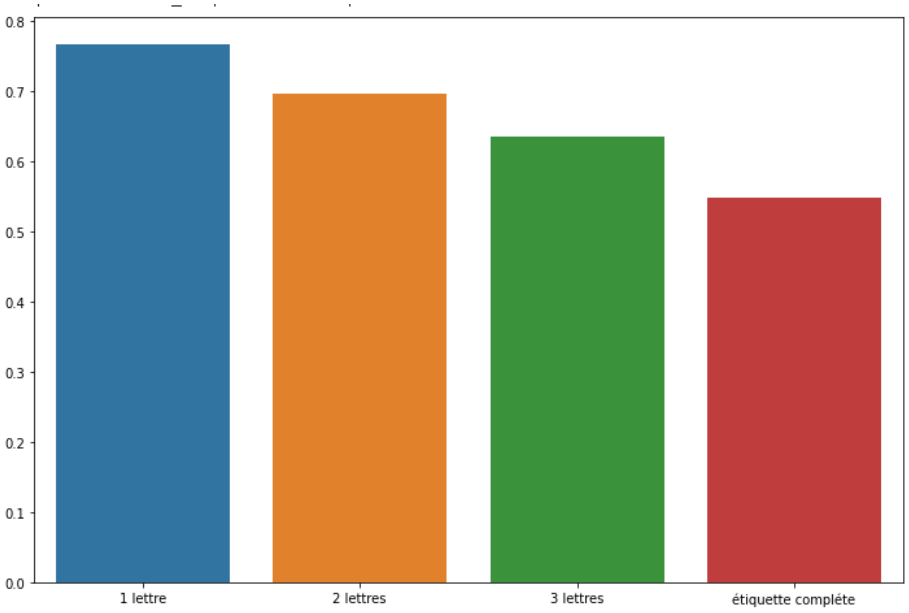


Figure 7 : Evolution des exactitudes en fonction de la taille considérée du label

Ces résultats correspondent à ce qui pouvait être attendu : Plus l'on considère de caractères et plus le nombre de clase augmente, ce qui rend la classification plus complexe pour le modèle et par conséquent, s'accompagne d'une baisse d'exactitude.

Exercice 4 : Modèle Sequence2Sequence

Dans cette partie, nous allons tirer parti du fait que l'étiquette CIM-10 peut être prédite caractère par caractère, puisque chaque nouveau caractère renvoie en fait à une catégorie plus précise de la classification, et a donc un sens en lui-même.

Exemple :

- **« S… » :** Blessures, empoisonnements et certaines autres conséquences de causes externes liées à des régions corporelles uniques
- **« S86. » :** Lésion du muscle, de l'aponévrose ou du tendon de la jambe inférieure
- **« S860 » :** Blessure du tendon d'Achille

Selon cette configuration, les données d'entrée mais également les sorties prédites par le modèle sont toutes deux des séquences : En effet, les données d'entrée sont similaires aux parties précédentes, soient des suites ordonnées de vecteurs correspondant chacun à un mot, et cette fois, la sortie du modèle est une séquence de vecteurs « one hot » (vecteur de taille du vocabulaire, rempli de 0 à l'exception d'un coefficient 1 à l'index du caractère dans le vocabulaire), chacun correspondant à un caractère rencontré dans la classification (26 lettres et 10 chiffres).

De plus, nous nous trouvons dans un cas particulier de la configuration « Sequence To Sequence », puisque la taille des séquences d'entrée mais également la longueur de l'étiquette prédite, ne sont pas fixées à l'avance. En effet, certaines étiquettes du jeu de données ne comptent que 3 caractères et non 4. Ce cas précis est désigné dans la littérature anglaise sous le terme de « Many to Many ».

En deep learning, l'architecture de référence pour ce cas de figure est l'architecture « Encodeur-Décodeur », qui peut être intuitivement résumé de cette manière : Un premier réseau de neurone, appelé « Encodeur », va chercher à projeter toute la séquence d'entrée dans un espace de grande dimension. Les vecteurs obtenus seront transmis en tant qu'inputs à un second réseau, appelé « Décodeur », qui va permettre de proposer une séquence de caractère correspondante.

L'exemple traditionnel pour illustrer cette architecture sont les modèles dits de « Neural Machine Translation » (NMT) ou « Modèle de Traduction Neuronale » en Français, et dans lequel la phrase dans le langage source est encodée entièrement puis transformée dans le langage cible par le décodeur.

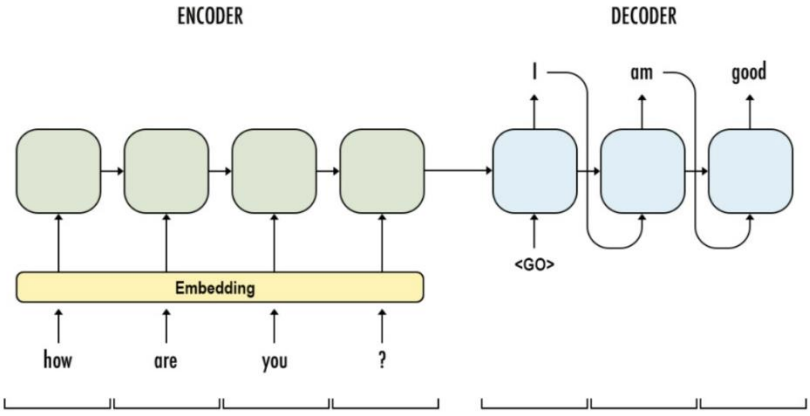


Figure 5 : Exemple de traduction neuronale

Pour construire notre code, nous avons d'ailleurs adapté un tutoriel utilisant un modèle encodeur / décodeur pour la traduction, à l'origine de l'Anglais vers le Marathi.

Pour gérer le problème des séquences de taille non-définies, une combinaison de deux méthodes est appliquée :

- Procédure de padding : Pendant l'entraînement, on rajoute des mots (entrée de l'encodeur) et des caractères (entrée du décodeur) de manière à ce que les séquences aient toutes une taille fixe, choisie en fonction de la taille maximale rencontrée dans le jeu de donnée. Les vecteurs ajoutés ne seront pas pris en compte par le modèle, leur unique rôle étant de permettre de fournir aux modèles des matrices de taille uniformes. Le padding n'est pas appliqué lors de l'inférence du modèle et ces tokens ne sont logiquement pas pris en compte lors du calcul des métriques en phase d'entrainement, ni lors du calcul de loss de la fonction objectif du modèle. Cette procédure est gérée par le fait que nous paddons

avec des 0, puis que nous spécifions à la couche d'embedding le fait de mettre de côté ces éléments (à l'aide du paramètre « mask_zeros » de la couche d'embedding de keras).

- Utilisation de tokens spéciaux « <s> » et « </s> ». Ces tokens, utilisés dans les données qui approvisionnent le décodeur, permettent de faire l'inférence un caractère à la fois, et donc de gérer le problème de la taille non-fixée pour la séquence de sortie. En effet, lors de son entraînement, le décodeur aura comme entrée pour chaque observation, la liste des caractères de l'étiquette, précédée du token de début de séquence <s>. Lors de l'inférence, la séquence initiale est uniquement constituée de ce token, et son association avec les états cachés des couches LSTM de l'encodeur permettront la prédiction du premier caractère, et ainsi de suite. Le token de fin de séquence, </s>, est placé après le dernier caractère de l'étiquette (avant l'application du padding) et signale que la prédiction du modèle doit s'arrêter pour cet individu. Evidemment, lors de la phase d'entraînement, les séquences source et cible qui constituent l'entrée et la sortie du décodeur sont décalées d'un élément, de manière à ce que le décodeur ne puisse pas connaître à l'avance le caractère qu'il est censé prédire.

En pratique, la procédure complète de l'entraînement du modèle seq2seq comprend en fait deux décodeurs distincts, avec des architectures similaires : Le premier est utilisé pour permettre l'entraînement de l'encodeur, de manière à assurer une projection des séquences textuelles de bonne qualité. Le second décodeur, qui sera le complémentaire de l'encodeur précédent, est utilisé pour l'inférence (prédiction des labels).

Il découle de ces éléments que lors de la phase d'inférence, la prédiction des étiquettes a deux conditions d'arrêt différentes : Soit la taille maximale pour une étiquette a été atteint (label de taille 4), soit le token de fin de séquence a été prédit prématurément (label de taille 3). Au début de la prédiction, il s'agit simplement de fournir au modèle le token de début de séquence ainsi que les états cachés des LSTM de l'encodeur, puis de lui donner itérativement en entrée le token qui vient d'être prédit.

De nouveau, pour gérer les tokens inconnus (présents dans les jeux de validation et de test mais non dans le jeu d'entraînement), nous avons choisi d'ajouter un token <UNK> au dictionnaire des mots créé à partir du jeu d'entrainement.

Les paramètres d'entrainement du modèle sont listés ci-dessous :

- Batch Size : 512
- Dimension d'embedding sur les mots : 200
- Epoch : 10
- Optimiseur : Adam
- Nombre de neurones des couches LSTM : 150
- Loss : Entropie croisée catégorielle

Le paramètre de batch size correspond au nombre de données individuelles prises en compte par le modèle avant un calcul de gradient. Ici, une batch size large permet d'accélérer l'entraînement du modèle puisque le modèle n'aura que peu (relativement) de fois à faire d'itérations de l'algorithme de descente de gradient.

Un nombre de neurones LSTM de 150 dans les couches respectives de l'encodeur et du décodeur est considéré comme important, mais nous avons choisi de conserver cette valeur en prenant en compte la complexité de la tâche visée. En rendant le modèle aussi complexe, le

risque de surapprentissage est important, ce paramètre sera probablement à ajuster par la suite.

De nouveau, nous utilisons l'entropie croisée catégorielle qui permettra de comparer termes à termes le vecteur cible encodé en « one-hot », ainsi que la prédiction du modèle issue de la couche dense utilisant comme fonction d'activation la fonction softmax. La formule de cette loss est mentionnée ci-après, et tire avantage du fait que le coefficient du vecteur cible, dénoté y_i, ne peut prendre que les valeurs 0 ou 1 :

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\ \text{size}}} y_i \cdot \log \hat{y}_i$$

Figure 6 : Entropie croisée catégorielle

Les résultats obtenus avec la combinaison encodeur / premier décodeur (modèle d'entraînement), sont les suivants :

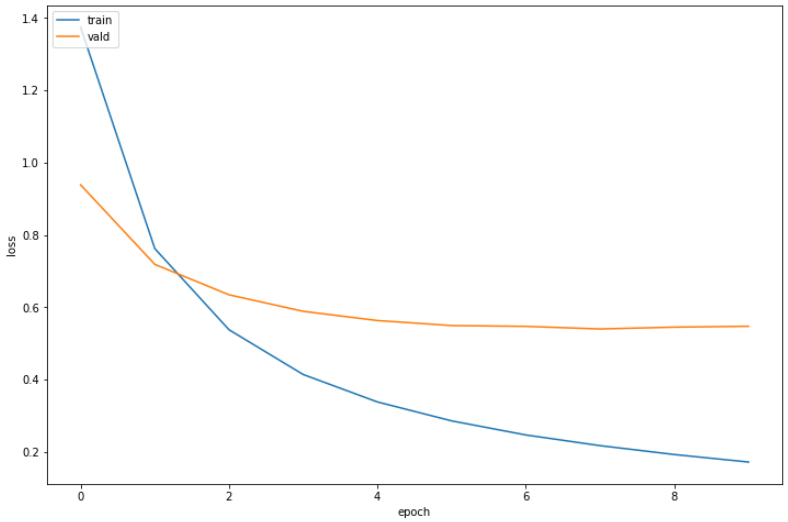


Figure 9 : Evolution des loss pendant l'entrainement du seq2seq

L'entraînement s'est déroulé correctement : La combinaison entre le nombre d'epoch et la complexité du modèle a permis une stabilisation de la loss de validation. Nous pouvons conserver les poids qui correspondent à la dernière epoch ($10^{ème}$).

Les résultats au regard des métriques sont les suivants :

Métrique	**Entraînement**	**Validation**
Exactitude	0.928	0.814
F1-Score	0.789	0.692

Il apparait que ce modèle est ostensiblement plus efficace dans la prédiction de l'étiquette complète que le modèle présenté pour la partie 3 (accuracy en validation de 0,81 contre 0.56), ce qui prouve la régularité de la considération de l'étiquette comme une séquence de caractères.

Pour illustrer les prédictions du modèle d'inférence, voici une capture d'écran de prédictions sur le jeu de test (prises au hasard) :

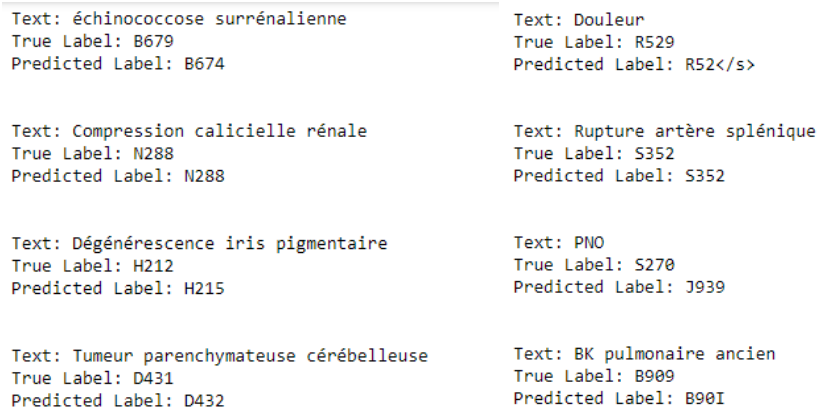


Figure 10 :Prédiction des labels par le modèle seq2seq

À partir de ces exemples, on remarque que le modèle a réussi à tirer avantage de la décomposition de l'étiquette comme une liste de caractères : Il est ainsi fréquent que le modèle réussisse à prédire les trois premiers caractères, mais échoue sur le quatrième, ce qui correspond au cas où le modèle a correctement cerné la pathologie, mais à manqué une subtilité dans la classification.

Exemple de ce phénomène : Sur la capture d'écran de gauche, on peut voir que le modèle a échoué dans la classification d'un texte en confondant les deux étiquettes suivantes :

- D431 : Tumeur à évolution imprévisible ou inconnue de l'encéphale, infratentoriel

- D432 : Tumeur à évolution imprévisible ou inconnue de l'encéphale, sans précision

Enfin, on peut également remarquer que le système qui permet de prédire des étiquettes de trois caractères est au point : Lorsque le token de fin de séquence est proposé à l'inférence, la prédiction de l'étiquette est arrêtée.

Exercice 5 : Pour aller plus loin

Traitement 1 : Préprocessing par lemmatisation

Pour les quatres premiers exercices, nous n'avons appliqué aucun préprocessing aux données textuelles. Dans cette section, nous allons appliquer une procédure appelée « Lemmatisation », soit un traitement lexical qui consiste à traiter différentes occurrences de même lexèmes de manière à les représenter sous une forme commune (canonique) appelée « Lemme ».

Pour des données textuelles en français, on pourrait rencontrer la séquence textuelle suivante : « Les petits garçons et les petites filles jouaient ensemble dans la cour de récréation ». Ici, les mots « petits » et « petites », manifestation d'un même lexème, seraient recodés de la même manière, par exemple par le lemme « petit ».

La lemmatisation permet en conséquence de forcer l'uniformisation du traitement de mots proches au niveau du sens par le modèle, ce qui évite les désagréments qui pourraient découler de considérations vis-à-vis du genre, nombre, personne ou mode pour un même lexème.

Pour cette partie, nous avons repris le modèle de la partie 3, sans modifier l'architecture, de manière à évaluer correctement l'apport du préprocessing. Pour les mêmes raisons, les valeurs des hyperparamètres d'entraînement n'ont pas été changées.

Les résultats sont les suivants :

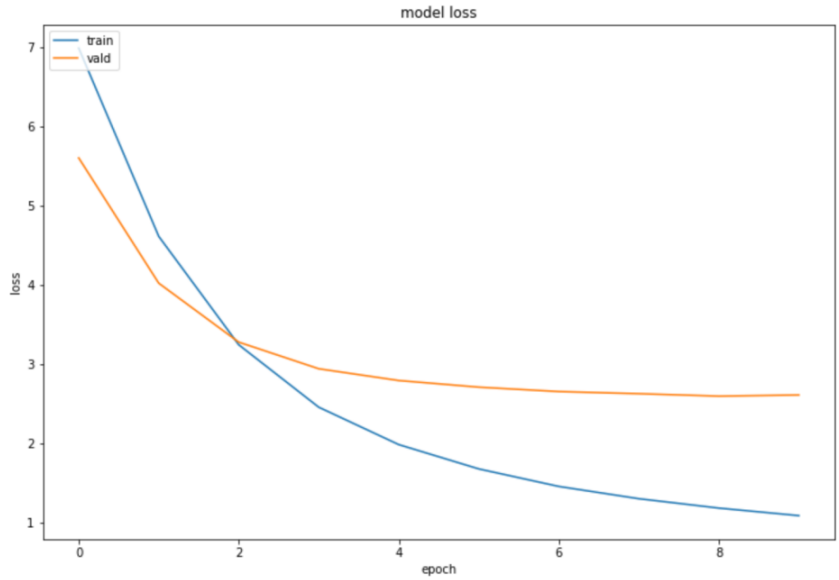


Figure 10 : Evolution des loss pendant l'entraînement après lemmatisation

L'entraînement s'est déroulé de manière efficace, la loss de validation a bien pu être stabilisée sans commencer à remonter.

Les résultats des prédictions au regard de l'exactitude sont les suivants (comparaison avec le modèle sans préprocessing de la partie 3) :

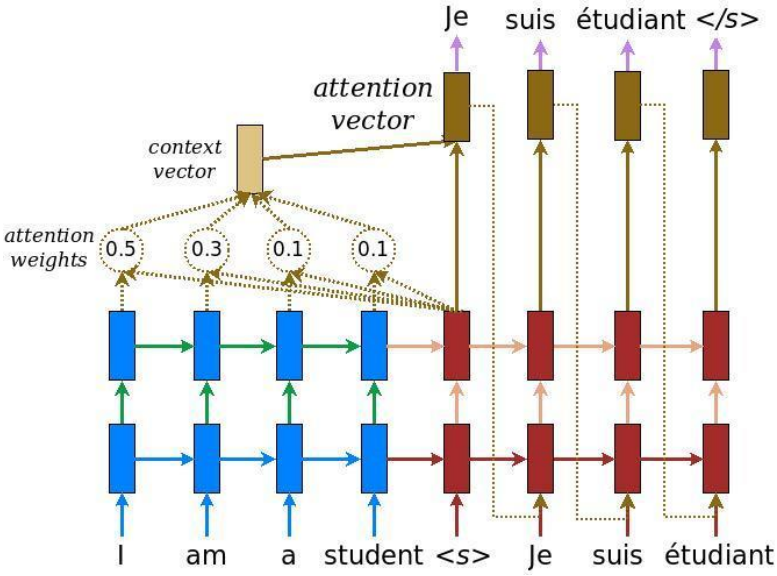
Métrique	**Entraînement**	**Validation**	**Test**
Sans Préprocessing	0.809	0.553	0 .549
Avec Lemmatisation	0.723	0.563	0.559

On remarque que l'apport de la lemmatisation est peu important, voire qu'il aboutit à une baisse des performances : Par exemple, sur le jeu de test, le préprocessing amène à une augmentation minime de l'accuracy (0,1%).

L'explication que nous proposons pour ce phénomène est que la lemmatisation n'est pas adaptée à des vocabulaires aussi technique et précis que le jargon médical. En effet, la perte d'information du passage des formes inflexées vers les lemmes est fortement préjudiciable pour une tâche dans laquelle les moindes subtilité dans le choix des mots doit être pris en compte.

Traitement 5 : Les modèles RNN encodeurs décodeurs avec attention

Dans ce traitement, nous avons utilisé un modèle RNN encodeurs décodeurs avec attention. Comme son nom l'indique, le modèle fonctionne avec un encodeur et un décodeur comme dans l'image suivante :



Avec ce modèle, nous pouvons comprendre comment le modèle apprend et sur quelle partie de la phrase d'entrée est porté l'attention de celui-ci lors de la traduction.

Le modèle fonctionne de la manière suivante :

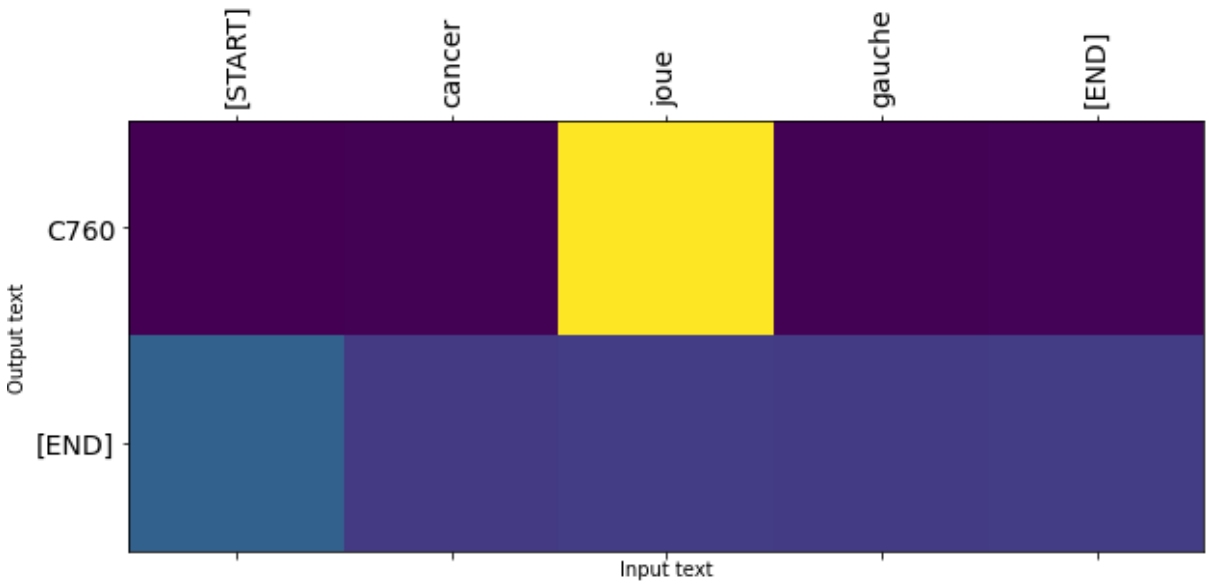
- Nous appliquons un nettoyage des données sur le RawText et les labels (ajout des tokens START et END)
- Un encodeur
- Un décodeur

L'entrainement du modèle a été fait sur 3 époques. Nous constatons que le modèle arrive à prédire avec exactitudes certain ICD10. Dans d'autre cas le modèle se trompe complètement.

Cela peut s’expliquer par le déséquilibre des classes dans notre datasets et par le aux nombres d’époques choisit qui est très bas.

Dans les images ci-dessous, on peut voir quel mot est porteur de l’information pour le modèle.

```
predicted label : C760
true label : C760
```



Dans ce cas-ci, le mot porteur de l’information pour le modèle est le mot « joue ».