Analyse de sentiments en utilisant l'apprentissage machine

Anthony Abboud
1681547

Abass Zeidan
1666449

Clint Phoeuk
1737731

Abstract

Introduction

De nos jours, il y a plusieurs formes de divertissement telles que les films, les livres et les jeux vidéo. Quelques uns sont aimées par l'audience tandis que d'autres ne le sont pas. Pour toutes formes de divertissements, il y aura toujours des critiques, négatives ou positives. Cette information peut être importante pour les spécialistes du marketing pour déterminer ce que les gens aiment en général.

Alors, dans ce projet, nous allons essayer de déterminer si une critique est positive ou négative en utilisant les réseaux de neurones. Puis, nous voulions aussi savoir quels hyperparamètres affecteraient la performance de notre modèle.

Dans les sections qui suivent, nous allons détailler la recherche qui a déjà été faite sur le sujet. Puis, nous allons aborder l'approche théorétique que nous avons utilisé pour résoudre le problème. Par la suite, nous allons présenter et discuter des résultats de l'expérience.

Mise en contexte

Dans la litérature, nous avons trouvé quelques papiers de recherche qui utilisaient une approche avec les *Support Vector Machines* (SVM) (Pang, Lee, and Vaithyanathan 2002; Mozetič et al. 2018) et une autre approche qui employait un réseau de neurones récurrent (RNN) avec une couche récursive (Timmaraju and Khanna 2015) pour résoudre le problème.

L'approche SVM est utilisée pour pouvoir classifier des données. Le but du SVM est de construire une ligne séparatrice qui est appelée une *hyperplane* afin de séparer les données comme dans la figure 1. Dans l'expérience de (Mozetič et al. 2018), il y a trois classes possibles: positive, neutre ou négative. Alors, pour déterminer à quelle catégorie appartient une critique, les auteurs ont créé un modèle SVM avec deux hyperplanes. Ce modèle sépare les critiques positives et neutres par une hyperplane et en utilise une autre pour séparer les critiques neutres et négatives. Ceci permet de séparer les données en trois classes.

Cependant, depuis l'avènement de l'approche SVM, de nouvelles technologies ont été mises au point. Les auteurs

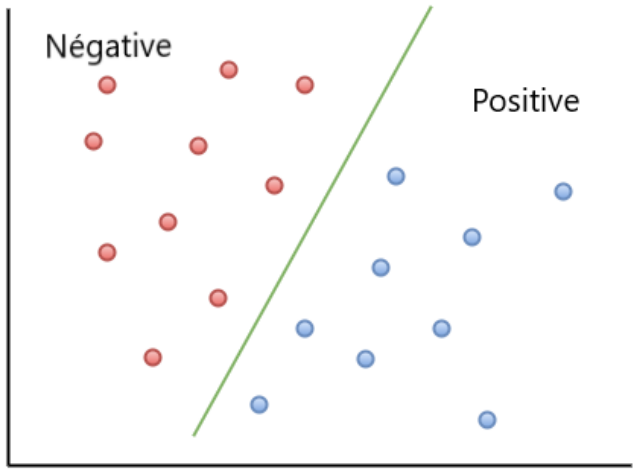

Figure 1: Support Vector Machine

(Timmaraju and Khanna 2015) ont donc décidé d'appliquer ces nouvelles technologies pour classifier une critique. Ainsi, pour une approche plus moderne avec les réseaux de neurones (Neural Network aka NN), ils ont construit un réseau qui combinait un NN récurrent et un récursif. Ce réseau est présenté dans la figure 2.

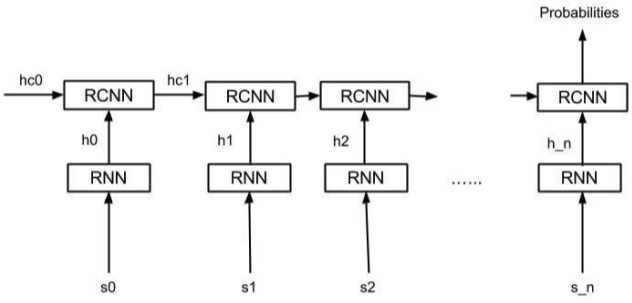

Figure 2: Recursive-Recurrent Neural Network (Timmaraju and Khanna 2015)

Chaque phrase est passée dans la couche récursive et cette couche émet en sortie un vecteur et un sentiment. Puis,

cette information est passée dans la couche récurrrente. Par la suite, les probabilités qu'une critique soit positive ou négative en sortent.

Dans leur recherche, ils ont trouvé qu'il n'y avait pas d'améliorations à la méthode traditionnelle qui était d'utiliser une approche avec SVM. On peut voir les comparaisons des différentes architectures dans le tableau 1

Approach	Test Accuracy
Method 3.1, Mean Word - RecNN	82.35%
Method 3.2, Mean Prob - RNN	81.8%
Method 3.3, RNN-Affine	81.42%
Method 3.4, SVM-RBF	76.69%
Method 3.4, SVM-Linear	83.28%
Method 3.5, SVM-Linear	**86.496%**
Method 3.5, 2-layer NN	83.94 %
Method 3.6, RecNN-RNN	83.88%

Table 1: Résultats des différentes architectures

Approche utilisée

Dans le contexte de notre projet, nous avons décidé d'utiliser un réseau de neurones récurrent (RNN) unidirectionelle et bidirectionnelle afin d'en observer les résultats. Dans les sous-sections qui suivent, nous allons vous expliquer les réseaux que nous avons utilisé et les détails de l'expérience.

Détails des modèles

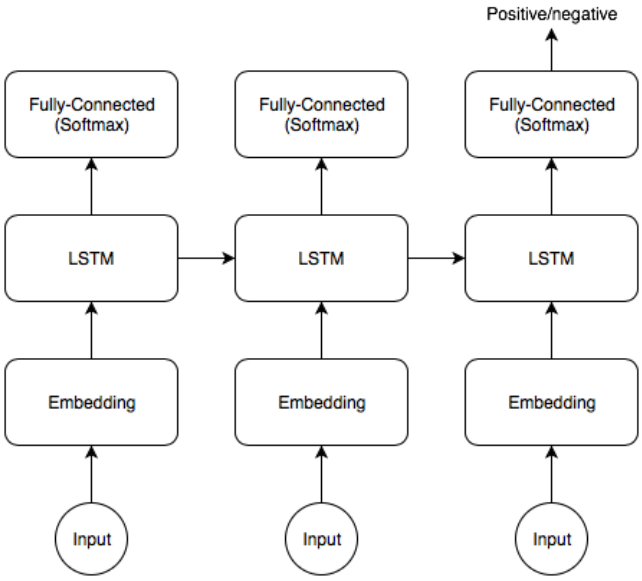

Figure 3: RNN unidirectionel

RNN unidirectionnel Dans la figure 3, nous pouvons voir les différentes couches de notre RNN. Notre première couche consiste en une "embedding layer" (ou couche d'encodage). Cette couche prend en entrée une phrase - qui est représentée par un vecteur contenant les index des mots dans un dictionnaire - afin de l'encoder dans un nouveau vecteur qui sera ensuite retourné. Il est important de noter que ce n'est pas du *one-hot encoding*.

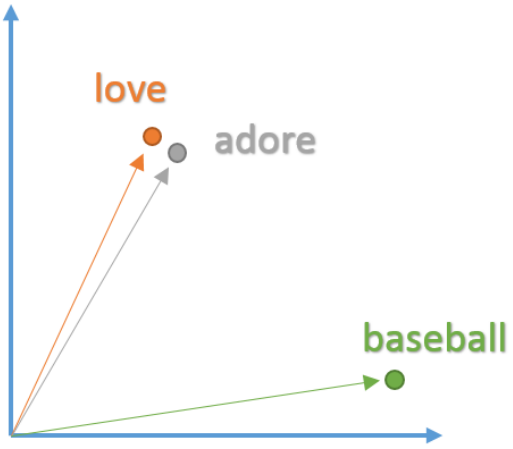

Figure 4: Embedding layer word context graph

Dans la figure 4, on peut voir que les mots *love* et *adore* sont proches tandis que le mot *baseball* est éloigné. Ceci est dû au fait que les mots *love* et *adore* sont très similaires lorsqu'ils sont placés dans une phrase. Donc, ces deux mots peuvent avoir le même sens dans la plupart des phrases ainsi, lorsque ces deux mots passent dans la couche "embedding", ils ont à peu près le même "encodage". Par contre, le mot baseball n'a pas la même signification. Il sera alors représenté par un vecteur totalement différent.

Puis, le vecteur en sortie de la couche "embedding" est passé dans une couche LSTM. Dans un LSTM, on peut observer une séquence: chaque unité nécessite l'exécution de la précédente unité. Cette restriction s'explique simplement par le fait que la sortie d'une unité est pris comme entrée de l'unité subséquente. Ainsi, à l'exception des unités au extrémités de la couche, chaque unité intermédiaire prend deux entrées et produit deux sorties. L'une des deux entrées vient de l'unité précédente et l'autre vient de la couche "embedding". Intuitivement, l'une des deux sorties est passé à la prochaine unité tandis que l'autre est passé à la prochaine couche: "fully connected layer".

La couche "fully connected" va permettre de générer un résultat afin de déterminer la classification de la critique. Ce résultat est généré à l'aide d'une fonction d'activation. Dans notre cas, il s'agit de la fonction "softmax" qui calcule les probabilités selon lesquelles une critique appartient à l'une ou l'autre des classes. Evidemment, la classe ayant la probabilité la plus élevé est celle à laquelle la critique sera attribuée.

De plus, il semble important de préciser que parralèlement, une regression s'effectue. En effet, lors de l'entrainement de notre modèle, il est nécessaire de constamment mettre à jour les différents paramètres afin que les résultats générés par la fonction "softmax" soient de plus

en plus précis jusqu'à atteindre un niveau de confiance considérablement élevé.

Afin d'accentuer votre compréhension du fonctionnement du modèle, clarifions la méthode d'entrainement et de prédiction que le modèle utilisera.

À chaque critique à travers laquelle le modèle passera, il analysera les mots qu'elle contient afin d'établir des statistiques entre le mot en question, sa fréquence d'apparition et la classification réelle de la critique qui le contient. Par exemple, pour le mot *love* dont on a parlé précédemment, le modèle pourrait conclure comme suit:

- 99 apparitions dans des critiques positives;
- 1 apparition dans une critique négative.

Ainsi, ce mot apportera à une critique une forte connotation positive donc une probabilité élevé quant à sa classification en tant que critique positive.

Après l'entraintement du modèle, vient la prédiction de l'appartenance d'une critique. Cette étape est relativement simple pour le modèle. Evidemment, toute critique à prédire passera par les différentes couches afin de finalement générer les bonnes probabilités de son appartenance à l'une ou l'autre des catégories, ce qui peut sembler compliquer. Expliquons cela en terme de mots. Le modèle va simplement analyser chaque mot de la phrase afin de déterminer sa connotation. En fonction des différentes connotations, les probabilités finales seront générées. Selon le niveau de confiance associé à la connotation d'un mot, le niveau de confiance selon lequel une critique appartient à une certaine classe, sera plus ou moins élevé.

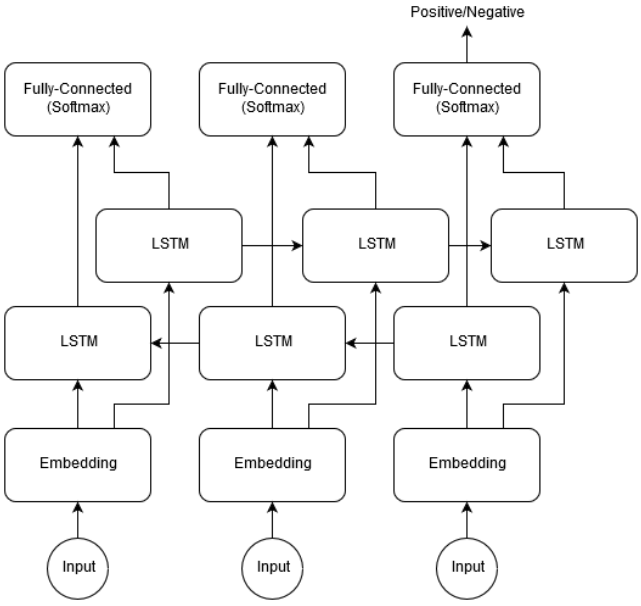

Figure 5: RNN bidirectionnel

RNN bidirectionnel Dans la figure 5, nous avons les différentes couches de notre RNN bidirectionnel. Comme vous pouvez le constater, il est similaire au premier modèle qui était un simple RNN. La différence avec le modèle bidirectionnel est que nous avons ajouté une deuxième couche LSTM qui servira à capturer le contexte dans le sens inverse d'une phrase. Par exemple, pour la phrase "This restaurant is very good at keeping me away from it!", notre premier modèle voit les mots "very good" et considère que la critique est positive, mais ce n'est pas le cas. Par contre, avec notre deuxième modèle, nous pourrons capturer les mots "keeping me away from it" et deviner que la critique est négative. Nous pensons que ce modèle aura une meilleure performance que la première.

Détails des expérimentations

Pour notre jeu de données, nous avons utilités des critiques dans la base de données du site IMDB - qui répertorie une large base de données de films - ainsi il s'agit de critiques liées aux films répertoriés sur ce site. De cette base de données, nous avons choisi de ne sélectionner "que" 10000 mots (un nombre indéterminé de critiques) afin d'entrainer notre modèle en supposant qu'il s'agit d'un nombre considérable de mots sur lesquelles le modèle pourra ensuite se baser afin de classifier une critique.

Dans notre cas, le langage utilisé est l'anglais. Cependant, nous aurions pu choisir n'importe quelle langue dans la mesure ou le modèle ne se base pas sur notre compréhension de la langue afin de prédire mais plutôt sur les correspondances qu'il peut observer dans la base de données. Ainsi, il suffirait simplement d'entrainer le modèle dans une certain langage afin qu'il soit capable de prédire la classification de critiques écrites dans ce même langage.

Expérimentation 1 Dans cette première expérience, nous voulons effectué des tests sur la façon dont les hyperparamètres *n-epoch*, *learning-rate*, *batch-size*, *dropout* et *optimizer* affectent notre modèle. Pour cela, lors de la réalisation des expériences, nous avons décidé de faire varier un hyperparamètre à la fois en conservant les autres à leur valeur par défaut.

Résultats

#	n_epoch	learning_rate	batch_size	dropout	optmizer	loss	accuracy
1	default	default	default	default	default	0.48306	0.7964
2	1	default	default	default	default	0.63814	0.652
3	7	default	default	default	default	0.52481	0.7956
4	10	default	default	default	default	0.58156	0.8032
5	default	0.001	default	default	default	0.47768	0.82
6	default	0.0005	default	default	default	0.46774	0.8208
7	default	0.00005	default	default	default	0.4801	0.7748
8	default	default	100	default	default	0.58679	0.7924
9	default	default	300	default	default	0.52502	0.7828
10	default	default	500	default	default	0.46699	0.7872
11	default	default	default	0.1	default	0.50428	0.758
12	default	default	default	0.5	default	0.43104	0.8108
13	default	default	default	0.9	default	0.53419	0.7784
14	default	default	default	default	sgd	0.69317	0.4868
15	default	default	default	default	momentum	0.69317	0.5
16	default	default	default	default	adagrad	0.69311	0.5112
17	5	0.0005	200	0.5	adam	0.42829	0.82

Table 2: Accuracy and Loss on with on different parameters on some test sets

Les différentes expériences, les valeurs de leurs paramètres ainsi que les résultats de l'expérience - soit le 'loss' et le 'accuracy' - sont énumérés dans le tableau 2.

La première expérience est celle pour laquelle l'ensemble des hyperparamètres sont à leur valeur par défaut:

- n-epoch = 5
- learning-rate = 0.0001
- batch-size = 200
- dropout = 0.8
- optimizer = adam

Ces paramètres ont été choisis lors de nos recherches et en s'inspirant des différents exemples d'implémentations qu'on a pu consulter. Cette expérience nous servira de référence lors de l'analyse des résultats des expériences suivantes.

Les expériences 2 à 16 sont réalisées en changeant un hyperparamètre à la fois afin de correctement cerner l'impact de ce paramètre sur la performance de notre modèle. Comme on peut le remarquer dans le tableau 2, nous avons sélectionner pour chaque hyperparamètre trois valeurs qui tournent autour de la valeur par défaut (en se rapprochant parfois des extrêmes). Le but de ces variations est d'observer leurs impacts sur le modèle.

Finalement, l'expérience 17 regroupe les valeurs des hyperparamètres pour lesquelles le duo 'accuracy'/'loss est maximale. Par exemple, pour *dropout*, la valeur 0.5 a été sélectionnée car c'est celle qui a donné la meilleure 'accuracy' ainsi que la meilleure 'loss' parmis les expériences 11, 12 et 13. Le but de ce mélange était de vérifier si le fait de mettre ensemble les paramètres qui ont individuellement donné les meilleurs résultats, permettait d'obtenir un résultat encore meilleur.

Expérimentation 2 Dans cette deuxième expérimentation, nous cherchons à voir si notre modèle bidirectionnel est bien plus efficace que le modèle précédent.

En se référant à l'analyse de l'expérimentation 1 (cibas), on remarque que les variations des hyperparamètres *dropout* et *optimizer* n'affecte pas vraiment l'efficacité de notre modèle unidirectionnel. Donc, dans le cas du modèle bidirectionnel, nous ne conservons que les valeurs optimales des hyperparamètres ayant affectés le modèle avec une plus grand ampleur (*n-epoch*=5, *learning-rate*=0.0005, et *batch-size*=200). Ceci nous permettra d'éviter de répéter des tests dont les résultats sont déjà évidents et qui ne nous mèneront nulle part.

De ce fait, dans ce deuxième modèle, nous conserverons les valeurs optimales de *n-epoch*, *learning-rate*, et *batch-size*, mais étudieront la variation de deux autres hyperparamètres que l'on puisse affectera grandement l'efficacité de ce deuxième modèle: *output-dim*, qui représente la grandeur du vecteur utilisé à la sortie des couches embedding ainsi que *vocab*, qui représente le nombre de mots dans le vocabulaire.

Les graphiques de *accuracy* et *loss* de chacune de ces expériences peuvent être retrouvées dans la section *Annexes*.

Le tableau suivant présente les résultats de cette deuxième expérience avec notre modèle bidirectionnel.

#	n_epoch	learning_rate	batch_size	output_dim	vocab	loss	accuracy
1	5	0.0005	200	128	10000	0.4056	0.8364
2	5	0.0005	200	64	10000	0.54036	0.77
3	5	0.0005	200	256	10000	0.71026	0.5716
4	5	0.0005	200	128	5000	0.69895	0.5276
5	5	0.0005	200	128	20000	0.47858	0.8052
6	5	0.0005	200	256	20000	0.3728	0.8546

Table 3: Accuracy and Loss on with on different parameters on some test sets for bidirectionnal model

Analyse et discussion

Expérimentation 1

Commençons tout d'abord par analyser notre expérience initiale, l'expérience de référence. Comme on peut le voir sur les figures 6 et 7, la 'accuracy' est croissante jusqu'à une valeur égale à 0.7964 tandis que la 'loss' est décroissante jusqu'à une valeur de 0.48306 (extraites du tableau); ces deux paramètres semblent inversement proportionnels. Les résultats obtenus avec les paramètres par défaut atteignent une 'accuracy' presque égale à 80%. Ce résultat est considérablement bon sachant que nous n'avions eu besoin que de quelques minutes pour entrainer le modèle avec un nombre de epochs égale à 5, ce qui est considérable étant donné l'ampleur de notre modèle constitué notamment d'un LSTM.

Abordons ensuite les expériences 2 à 16 qui constituent en fait en une analyse de l'impact de chacun des hyperparamètres sur notre modèle:

- n-epoch: Rajouter 5 epochs n'améliore pas considéralement la 'accuracy' tandis que la 'loss' semble également augmenté. Cependant, l'augmentation de la première n'est pas profitable au dépend de l'augmentation de la seconde. De plus, utiliser moins de epochs - 1 seul pour notre expérience - entraine une baisse considérable de la 'accuracy'. Ainsi on considérera notre valeur par défaut comme étant la meilleure valeur en supposant que 5 epochs sont suffisants et nécessaires afin d'entrainer notre modèle à partir de la base de donnée choisie. (cf. figures 8 et 9 de l'annexe)
- learning-rate: En faisant varier cet hyperparamètre, on remarque qu'il n'y a pas d'améliorations pour une valeur plus petite mais, lorsqu'on multiplie par 5 la valeur par défault, on observe une amélioration considérable de 3%. Dépasser ce pallier entraine par contre une baisse en terme de performance, i.e. une baisse légère de la 'accuracy' et hausse de la 'loss'. Ainsi, la meilleure valeur pour le learning-rate est 0,0005. (cf. figures 10 et 11 de l'annexe)
- batch-size: Les valeurs choisis afin de tester le batch-size sont 100, 200 (valeur par défaut), 300 et 500. On observe que 100, 300 et 500 conduisent toutes les trois à une 'accuracy' inférieure. Ainsi, la valeur 200 semble conduire à un maximum locale de la 'accuracy' pour une 'loss' très peu supérieure à celle obtenu avec un batch-size égale à 500. On considérera donc notre valeur par défaut 200 comme étant la plus adaptée. (cf. figures 12 et 13 de l'annexe)

- dropout: Dropout est une technique de régularisation qui permet une meilleure généralisation d'un modèle en abandonnant certains neurones du réseau de neurones. Ainsi, la valeur attribué à cet hyperparamètre se situe entre 0 et 1, avec 0 signifiant que rien ne sera abandonné et 1 signifiant l'inverse. Notre valeur par défaut est 0.8 et pour mener à bien notre analyse, nous avons choisis les valeurs 0.1, 0.5 et 0.9 afin d'observer l'impact de ces variations. Finalement, ce dernier est considérable et on remarque une 'accuracy' qui s'élève à 81% pour une valeur de dropout égale à 0.5. (cf. figures 14 et 15 de l'annexe)
- optimizer: Pour ce dernier hyperparamètre, on remarque simplement que les variations d'optimisateurs proposées ne sont pas du tout adaptées à la situation car il y a un déca d'environ 30% entre la 'accuracy' obtenue avec ces variations et celle obtenue avec l'optimisateur par défaut 'adam' que l'on gardera finalement. (cf. figures 16 et 17 de l'annexe)

Après analyse, on remarque que les hyperparamètres se classifient par différents niveaux d'importance en se référant à leur impact sur le modèle. Le batch-size, à l'échelle à laquelle nous l'utilisons, semble être l'hyperparamètres ayant le moins d'impact sur notre modèle étant donné des variations moindres de la 'accuracy'. Cependant, avec une valeur de batch-size égale à 500, on observe tout de même un impact considérable sur la 'loss' du modèle à ne pas négliger. Le nombre d'epochs, le learning-rate et le dropout ont tous les trois un impact considérable sur le système dans le sens où jouer avec chacun de ces paramètres peut entrainer des variations de 5 à 15% sur la 'loss' et la 'accuracy'. Ainsi, le fait de bien sélectionner ces valeurs permettra d'améliorer considérablement le modèle, dans notre cas spécifiquement mais probablement même dans la majorité des cas. Enfin, il ne reste que l'optimisateur sélectionné. Cet hyperparamètre peut avoir des conséquences drastiques sur le système en termes de performances. En effet, comme on peut le voir dans le tableau de résultats des expériences, si l'optimisateur n'est pas correctement choisis la performance du système baisse drastiquement. De plus, le nombre de epochs et le dropout peuvent également avoir des conséquences drastiques si les valeurs sélectionnées ne sont pas adéquates. Prenons par exemple le cas de l'expérience 2 pour laquelle on a un nombre de epoch égale à 1. On remarque que la 'accuracy' et la 'loss' ont tous les deux varié négativement de 15% environ. De plus, la valeur de dropout peut avoir des conséquences drastiques sur le système pour une valeur 1 car cela signifierait d'abandonner tous les neurones du réseaux. Cette expérience n'a pas été réalisée mais le résultat est facilement prévisible.

Finalement, nous avons utiliser les meilleures valeurs des hyperparamètres déduit à partir des expériences 2 à 16 afin de réaliser notre 17e et dernière expérience. Nous avons réalisé cette expérience en se demandant si le fait de rassembler les meilleures valeurs pourrait entrainer une performance encore jamais égalée lors de nos précédentes expériences. Après la réalisation de l'expérience, on remarque que en effet le fait d'utiliser, pour chaque hyperparamètre, la valeur qui a donné le meilleur résultat, permet d'obtenir une performance supérieure aux précédentes expériences. On le remarque en observant les valeurs de 'loss' et 'accuracy' qui représentent ici la meilleure combinaison obtenue jusqu'à présent. La valeur de la 'loss' est la plus basse parmi toutes les expériences et celle de 'accuracy' est égale à 0.0008 près à la plus haute valeur obtenue.

Expérimentation 2

Comme on a pu le préciser précédemment, nous voulions comparer ce nouveau modèle bidirectionnel constitué de deux LSTM à l'ancien modèle unidirectionnel constitué d'un simple LSTM. Pour cela, nous avons conservé les paramètres que nous avons défini comme optimaux suite à l'analyse des résultats des différentes expériences réalisées sur le premier modèle. Cependant, nous avons ajouté deux hyperparamètres sur lesquels nous avons décidé de conduire plus de tests:

- *output-dim* qui représente la taille du vecteur en sortie de la couche d'encodage
- *vocab*, le nombre de mots du vocabulaire connu par le modèle

On peut observer les résultats de ces expériences dans le tableau 3 ci-dessus.

L'expérience 1 utilise ici les mêmes valeurs des hyperparamètres que l'expérience 17 réalisée sur l'ancien modèle. On remarque qu'on obtient déjà de meilleurs résultats avec une 'loss' égale à 0.3728 et une 'accuracy' de 0.8546 soit 85% environ. Cela est du à l'utilisation d'un LSTM bidirectionnel, ce qui permet de mieux cerner le contexte du texte "lu".

On a ensuite joué avec les valeurs des deux hyperparamètres choisis comme on peut le voir pour les expériences 2 à 5. Nous avons remarqué qu'en les faisant varier individuellement, le résultat variait considérablement, mais seulement négativement, car on obtient des résultats inférieurs à ceux de l'expérience 1 et même inférieurs à la solution optimale du précédent modèle. Nous avons émis l'hypothèse suivante quant à cette situation: "cela s'explique peut-être par le fait que plus le vocabulaire du modèle est vaste, plus il faut augmenter le nombre d'unités en sortie afin de ne pas perdre de l'information lié au contexte du bout de texte étudié".

Dans le but de conclure sur cette hypothèse, nous avons réalisé l'expérience 6 pour laquelle nous avons doublé la valeur de *output-dim*, le nombre d'unités de la couche d'encodage et des LSTM, ainsi que *vocab*, le nombre de mots dont est constitué le vocabulaire du modèle. On observe pour cette expérience des résultats considérablement supérieurs aux précédents meilleurs résultats dans la mesure où la 'loss' et la 'accuracy' se sont tous les deux améliorés d'environ 3%. Pour confirmer plus certainement cette hypothèse nous avons aussi réalisé une expérience avec *output-dim*=64 et *vocab*=5000. Les résultats de celle-ci sont inférieurs aux précédents tout en restant cohérent avec les résultats des expériences 1 et 7, et avec notre hypothèse.

Finalement, après ces nombreuses expériences, nous concluons que le modèle constitué d'une couche RNN bidirectionnel (la bidirectionnalité étant déterminée par deux

LSTM) avec les bonnes valeurs des différents hyperparamètres est nettement plus performant qu'un modèle avec un RNN unidirectionnel (un seul LSTM). Les valeurs finales des hyperparamètres pour lesquelles le résultat est optimale sont les suivants:

- n-epoch = 5
- learning-rate = 0.0001
- batch-size = 200
- dropout = 0.5
- optimizer = adam
- output-dim = 256
- vocab = 20000

Avec ces valeurs, nous obtenons une 'loss' d'environ 37% et une 'accuracy' d'environ 85%.

Limitations

Durant notre expérience, nous avons eu quelques limitations qui auraient pu avoir un impact sur la performance du modèle. Une de ces limitations est le nombre de mots inconnus que le réseau apprend. Dans notre expérience, étant donné que nous n'avions les ressources nécessaires pour faire apprendre au modèle tous les mots d'une langue, il se peut qu'une critique ait des mots que le réseau ne connaisse pas. Ceci peut induire une baisse de performance parce que le réseau ne peut pas savoir si le mot inconnu a un sens positif ou négative.

Alors, pour améliorer les performance de notre réseau, on pourrait entrainer notre réseau avec plus de mots différents possibles pour que notre modèle soit plus complet. Ceci aiderait notre modèle à faire plus de liens entre les mots et une revue positive ou négative.

Analyse de l'approche utilisée pour notre apprentissage

Pour en apprendre davantage sur le sujet, nous avons recherché des papiers de recherche et des vidéos qui pouvaient nous expliquer les différentes façons de classifier une critique. Nous avons décidé de procéder de cette façon afin de s'inspirer des travaux antérieurs réalisés sur le sujet. Ces recherches ont éventuellement soulevé plus de questions, notamment sur certains concepts évoqués qu'on ne connaissait pas (comme la couche d'encodage) ou qui n'étaient pas clairs (comme les réseaux RNNs). Ainsi, nous avons approfondi nos recherches afin de récolter plus d'informations sur ces concepts là dans le but de comprendre les différentes méthodes présentées.

Au niveau du code, nous avons choisi la librairie TFLearn car l'implémentation du modèle requis est nettement facilité à l'aide de la librarie TensorFlow. Cette librairie facilite tellement les choses de part son abstraction de haut niveau, qu'elle limite notre liberté d'implémentation et il devient donc difficile d'écrire du code dont la structure est différente de celles déjà réalisées dans les différents exemples. Cependant, cela nous a aidé à accélérer le processus d'obtention et d'explication des résultats dans la mesure où nous n'avions pas à nous soucier des détails d'implémentation des méthodes déjà fournies. Par contre, pour mieux comprendre le fonctionnement de l'implémentation, il faudrait le coder nous-même à l'aide de pytorch ou même en commençant de rien, "from scratch".

Conclusion

En conclusion, nous avons utilisé les techniques d'apprentissage machine pour déterminer si une critique était positive ou négative. Dans notre projet, nous avons construit deux modèles RNN, l'un unidirectionnel et l'autre bidirectionnel. Alors, nous avons effectué deux ensembles d'expériences pour déterminer quel modèle avait une meilleure précision. Dans ces ensembles, nous avons fait plusieurs expérimentations pour savoir les meilleurs hyperparamètres et aussi pour savoir quels hyperparamètres pouvaient avoir un impact important sur nos réseaux de neurones. Pour le premier modèle, nous avons trouvé que le nombre de epochs, le taux d'apprentissage et la probabilité de dropout avaient affecté la précision du modèle. Puis, nous avons trouvé que pour le deuxième modèle, le nombre de mots dans le vocabulaire du modèle et la taille du vecteur à la sortie de la couche d'encodage avaient un impact significatif sur la performance de celui-ci. Nous avons remarqué que le modèle bidirectionnel était plus performant que le modèle unidirectionnel. Ceci est dû au fait qu'il arrive à mieux saisir le sens de la phrase étant donné qu'il analyse plus amplement le contexte de celle-ci en la "lisant" dans les deux sens.

Lien GitHub

Le code source et les résultats référés dans ce rapport peuvent être retrouvés au lien suivant: `https://github.com/AnthonyAbboud/INF8225-TP4-Sentiment-Analysis.git`

References

Deshpande, A. Perform sentiment analysis with lstms, using tensorflow. `https://www.oreilly.com/learning/perform-sentiment-analysis-with-lstms-using-tensorflow`. Accessed: 2018-04-07.

Mozetič, I.; Torgo, L.; Cerqueira, V.; and Smailović, J. 2018. How to evaluate sentiment classifiers for Twitter time-ordered data? *ArXiv e-prints*.

Pang, B.; Lee, L.; and Vaithyanathan, S. 2002. Thumbs up? sentiment classification using machine learning techniques. In *Proceedings of EMNLP*, 79–86.

tflearn. GitHub tflearn/bidirectional. `https://github.com/tflearn/tflearn/blob/master/examples/nlp/bidirectional_lstm.py`. Accessed: 2018-04-13.

tflearn. GitHub tflearn/examples. `https://github.com/tflearn/tflearn`. Accessed: 2018-04-08.

tflearn. GitHub tflearn/lstm. `https://github.com/tflearn/tflearn/blob/master/examples/nlp/lstm.py`. Accessed: 2018-04-08.

Timmaraju, A., and Khanna, V. 2015. Sentiment analysis on movie reviews using recursive and recurrent neural network architectures.

Turney, P. D. 2002. Thumbs up or thumbs down? semantic orientation applied to unsupervisedclassification of reviews. *CoRR* cs.LG/0212032.

Annexes

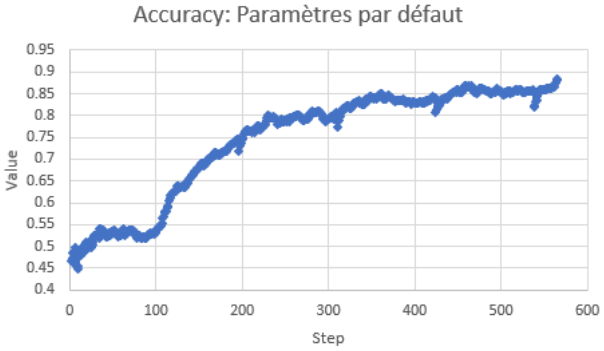

Figure 6: Accuracy with default parameters

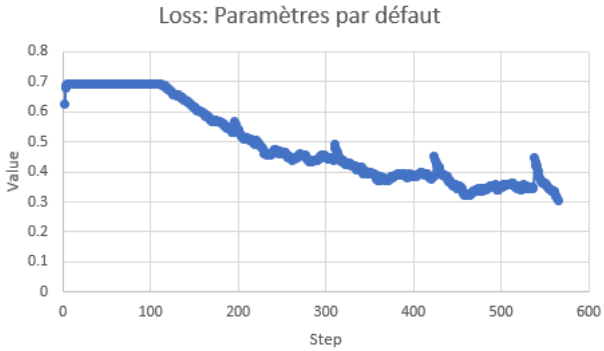

Figure 7: Loss with default parameters

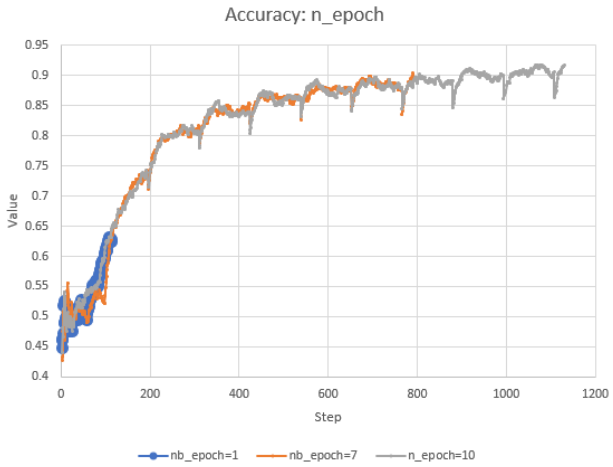

Figure 8: Accuracy with n-epoch varying

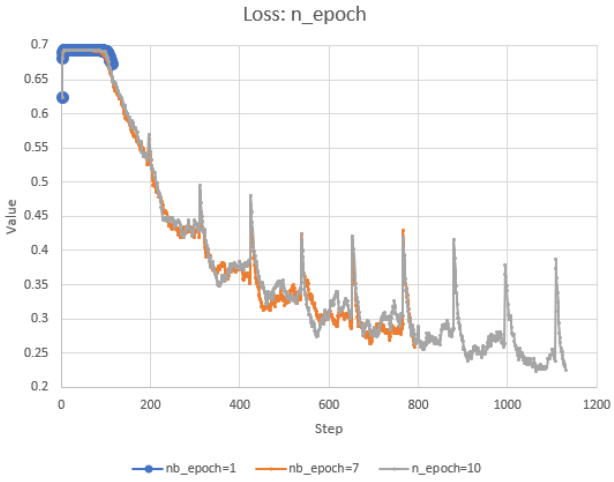

Figure 9: Loss with n-epoch varying

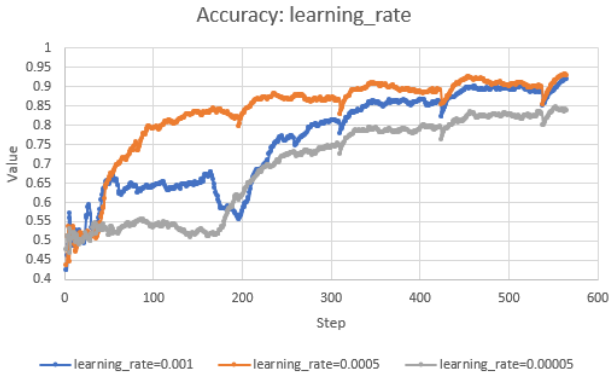

Figure 10: Accuracy with learning-rate varying

Figure 11: Loss with learning-rate varying

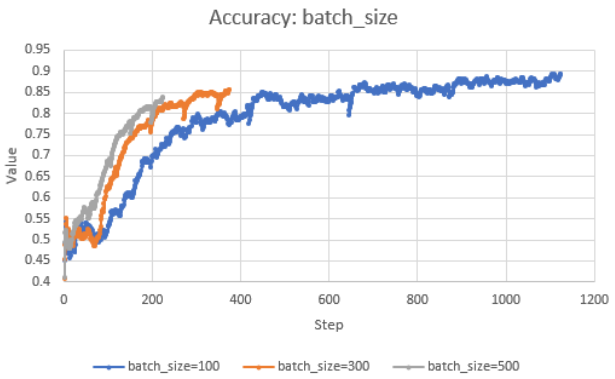

Figure 12: Accuracy with batch-size varying

Figure 13: Loss with batch-size varying

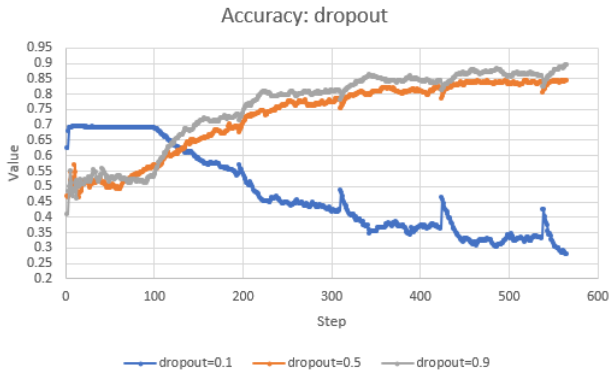

Figure 14: Accuracy with dropout varying

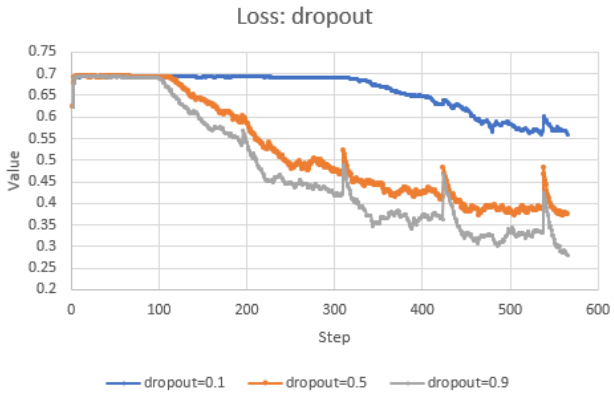

Figure 15: Loss with dropout varying

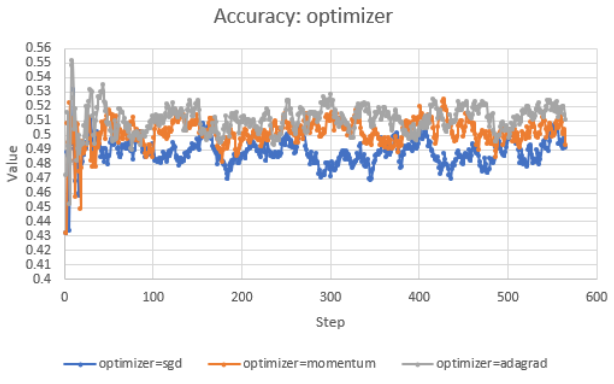

Figure 16: Accuracy with optimizer varying

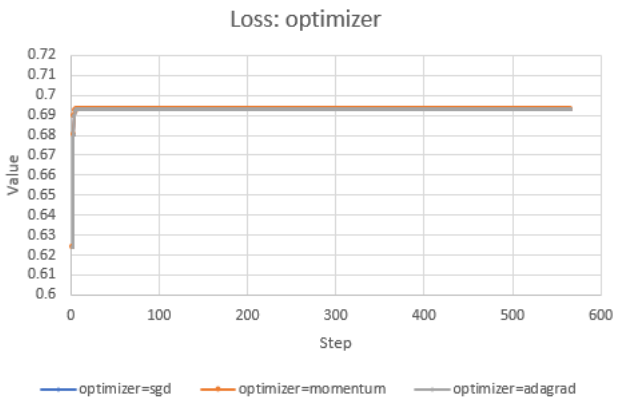

Figure 17: Loss with optimizer varying

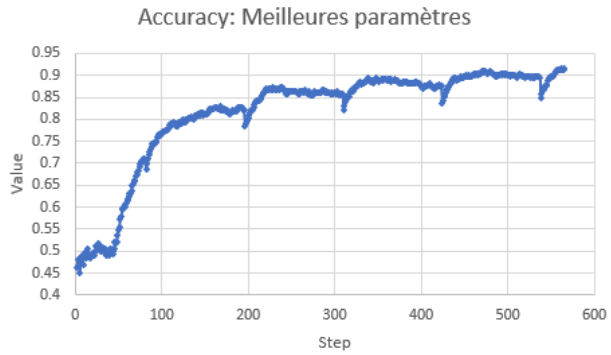

Figure 18: Accuracy with final varying

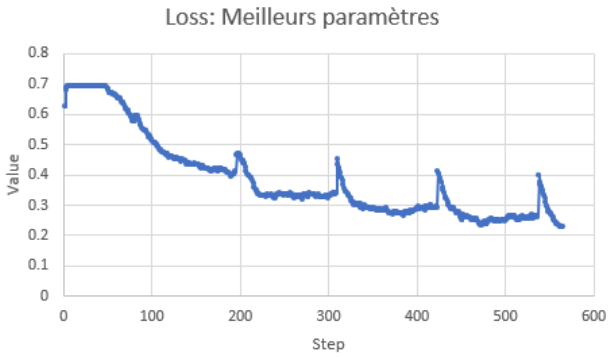

Figure 19: Loss with final varying

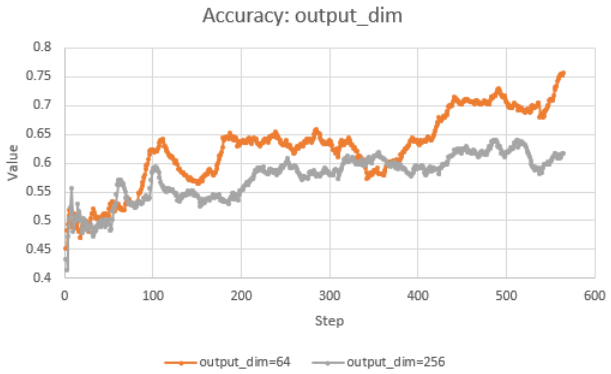

Figure 20: Accuracy with output dim varying in Bidirectional LSTM

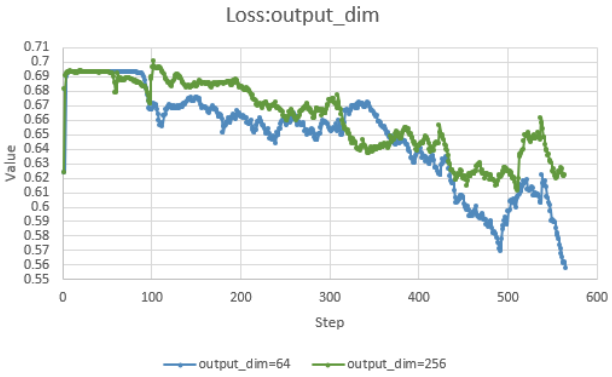

Figure 21: Loss with output dim varying in Bidirectionnal LSTM

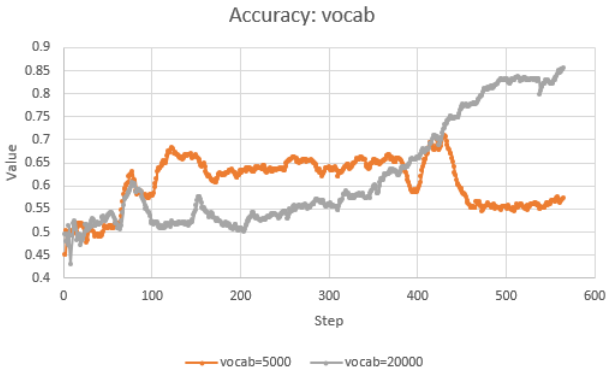

Figure 22: Loss with vocab varying in Bidirectionnal LSTM

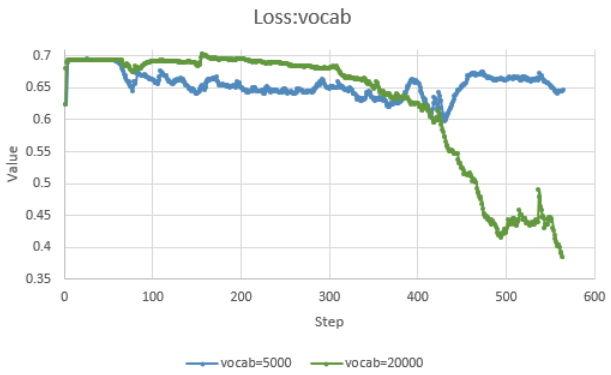

Figure 23: Loss with vocab varying in Bidirectionnal LSTM

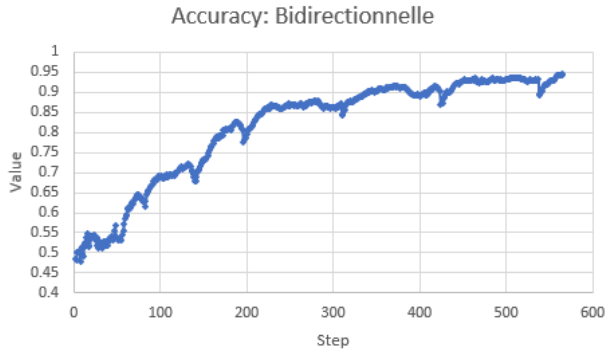

Figure 24: Accuracy with final varying in Bidirectionnal LSTM

Figure 25: Loss with final varying in Bidirectionnal LSTM