Classification de sentiments par réseau de neurones récursif tensoriel

Thomas Moreau
Ecole polytechnique
thomas.moreau@polytechnique.edu

Bertrand Rondepierre
Ecole polytechnique
bertrand.rondepierre@polytechnique.edu

***Résumé*—Dans ce travail on s'intéresse à une problématique d'analyse de sentiments sur des phrases de taille courte à moyennes. Nous nous basons sur le travail de R. Socher [1] dont nous reprenons les grandes lignes en les précisant et en l'accompagnant d'une réimplémentation complète de l'algorithme en Python (ainsi que certaines figures pour la clarté de l'exposé). Les méthodes employées entre dans la catégorie du Deep learning, et en particulier dans la classe des réseaux de neurones récurrents.**

I. Introduction

Après avoir été abandonnées dans les années 80 devant le manque de technique d'entrainement efficace, le domaine du deep learning est à nouveau en plein essor depuis 2006 notamment depuis les travaux de G. Hinton et Y. Bengio [2], [3], [4] qui ont abouti à des technique d'entrainement efficaces. Ce problème d'entrainement des modèle est la raison qui avait freiné le développement des réseaux de neuronnes, et bien qu'étant loin d'être résolu, des progrès majeurs ont été faits. De plus, la combinaison des facteurs : mise au point de techniques d'entrainement effectives, augmentation des capacités de stockage, démultiplication de la puissance de calcul a ainsi permis d'atteindre l'état de l'art sur un large spectre de tâches allant de la reconnaissance d'objet au traitement des langues, et en passant par de nombreux domaines d'intérêt du machine learning.

A. Idées générales et inspirations du deep learning

L'idée générale du deep learning est de s'inspirer de la physiologie du cerveau humain pour mimiquer de possible mécanismes d'apprentissage. L'aspect profond vient d'une part de cette idée que le cerveau humain organise les neurones par couches et d'autre part de l'idée que les concepts ne s'apprennent pas en se définissant par de nombreux exemples particuliers mais plutôt en généralisant l'information que chacun des élément apporte.

Ceci est parfaitement illustré par l'exemple en figure 1 : bien qu'ayant une expérience limitée en l'occurrence sur des moyens de locomotion, nous sommes parfaitement capable d'inférer le type de chacun des véhicules du haut (scooter et vélo) malgré l'écart qu'ils entretiennent avec le modèle moyen. De plus, bien que le troisième exemple soit tout à fait inconnu et d'un type sujet à controverse, nous sommes parfaitement capable de reconnaître qu'il s'agit d'un véhicule, d'un mélange entre plusieurs classes connues entre le scooter, la voiture, le vélo etc.

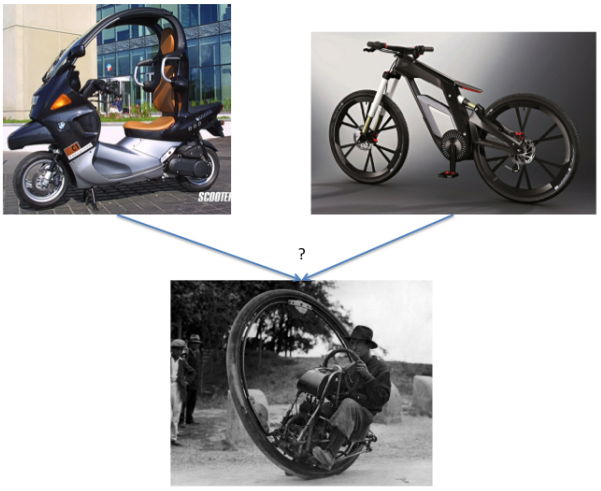


FIGURE 1: Trois exemples : en haut de gauche à droite : un scooter et un vélo clairement en marge du modèle moyen. En bas un moyen de locomotion étrange.

Le processus précédent est rendu possible par notre capacité à généraliser les exemples en concepts. Et c'est exactement la problématique qu'essaie d'aborder le deep learning, où l'aspect profond peut donc également être vu comme cette capacité à généraliser des exemples particuliers par de représentation de plus en plus abstraites suggérant cette idée de couches profondes. Par exemple pour des images ou des objets, on peut imaginer un modèle profond où une première couche constituera les arrêtes visibles au niveau des pixels, puis la seconde celle des parties des objets, puis celle des objets complets, puis les scènes complètes etc... En résumé sur le diagramme suivant :

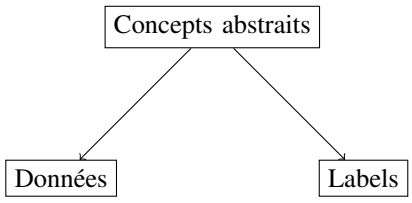


FIGURE 2: Approche logique

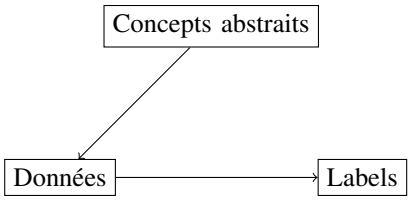


FIGURE 3: Approche classique, mais illusoire

Tout l'enjeu consiste à dire que finalement ce n'est pas vraiment le label qui va être le plus informatif pour construire ces concepts et ces représentations, et qu'il s'agit plus d'une conséquence que d'un identificateur clé. Pour cette raison, l'objectif majeur est d'inverser les transformations que subissent les concepts abstraits pour générer les données sur lesquelles nous travaillons. Une fois que l'on dispose de cela, prendre une décision devient facile.

B. Base du deep learning

1) Le modèle de neurone: Un neurone peut-être vu comme la plus simple unité computationelle prenant en entrée un signal $x^{input} \in \mathbb{R}^n$, et retournant une sortie x^{output} en combinant d'une part une opération affine et d'autre part un opération non-linéaire f :

$$x^{input} = x$$
$$x^{output} = f(Wx^{hidden} + B)$$

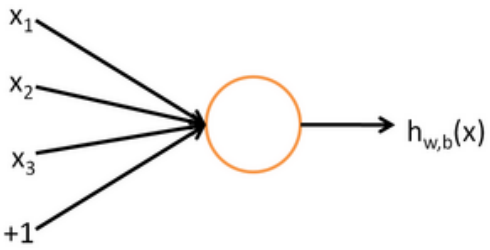


FIGURE 4: Un neurone simple

En général la non-linéarité est choisie entre une sigmoïde, une tangente hyperbolique ou encore des fonctions linéaires rectifiées. L'idée est alors de constituer un réseau avec un modèle graphique dans lequel on accumule un grand nombre de neurones répartis dans de (plus ou moins) nombreuses couches.

L'entrainement du modèle se fait alors soit de façon supervisée en ajoutant par exemple une couche de multi-layer perceptron et en optimisant le risque empirique, soit de façon non supervisée en entrainant le réseau comme un modèle génératif sur un critère de type maximum de vraisemblance. Le but est alors d'obtenir une représentation qui permette d'accomplir facilement un certain nombre de tâches, et ici en l'occurrence ce qui nous intéresse est la classification de sentiments.

2) Sur la nécessité de la non-linéarité: Une remarque importante qu'il convient de faire à ce niveau est celle de la présence de la non-linéarité f dans le modèle du neurone. Celle-ci est également une des idées clés du deep learning puisque dans la réalité, l'hypothèse de linéarité n'est certainement pas valables dans les représentations de bas niveau. En revanche c'est bien le cas dans l'espace des concepts dans lequel on essaye de travailler (ce qui n'est bien entendu pas encore réussi) où comme on l'a dit les décisions sont simples : différencier un vélo d'une voiture est très facile. Un modèle linéaire peut donc être parfaitement satisfaisant à condition d'être appliqué dans le bon espace. Cependant une composition de transformations linéaires reste linéaire et il est donc nécessaire d'introduire de la linéarité à un endroit donné. Cette non linéarité sur les représentations de bas niveau est évidente en considérant des exemples où deux objets vont être tout à fait identique du point de vue des concepts, mais avec des formes/textures/couleurs complètement différentes.

3) Le théorème d'approximation universelle: Une des garantie rassurante dans l'emploi de réseaux de neuronnes pour l'apprentissage de fonctions complexes est celle du théorème d'approximation universel (voir par exemple [5]). En effet, celui-ci affirme que pour des fonctions d'activation données, un réseau de neuronne est en mesure d'approximer n'importe quelle fonction continue sur un compact. Plus précisément :

Théorème 1 (Approximation universelle)**.** *Si f est une fonction continue, bornée, strictement croissante, en notant I_m l'hypercube $[0,1]^m$ et $C(I_m)$ les fonctions continues sur I_m alors pour toute fonction $r \in C(I_m)$ et $\varepsilon > 0$, il existe N et des constantes a_i, b_i, w_i respectivement dans $\mathbb{R}, \mathbb{R}, \mathbb{R}^m$ telles que*

$$F(x) = \sum_{i=1}^{N} a_i f(w_i^T x + b_i)$$

vérifie : $||F - r||_\infty^{I_m} < \varepsilon$. Autrement dit on a densité dans $C(I_m)$.

Ce théorème peut-être prouvé toujours valable dans le cas des rectified linear unit avec le max qui n'est plus borné et strictement croissant. La preuve repose sur l'approximation de fonctions convexes par des fonctions affines par morceau et le fait qu'une fonction continue peut s'écrire comme différence de deux fonctions convexes. Voir [6].

En résumé on a espoir de dire qu'il existe effectivement une fonction qui va permettre de mapper les entrées vers une représentation intéressante, et le théorème précédent permet de dire qu'il est effectivement possible d'apprendre ces fonctions avec un réseau de neurones.

Il faut cependant bien noter plusieurs choses : si l'approximation universelle est toujours valable, la question de quelles fonctions peuvent être approximées le plus efficacement avec des architecture de profondeur fixée reste toujours ouverte. En revanche on peut prouver que certaines fonctions représentables facilement avec une architecture de profondeur donnée requièrent avec une architecture de profondeur moindre un nombre de neurone exponentiellement plus grand. Cette idée suggère que la profondeur joue un rôle clef dans l'apprentissage de ces fonctions.

4) Entrainement des réseaux de neurones: Malgré les garanties théoriques issues de l'approximation universelle ou de la théorie de Vapnik-Chervonenkis, le problème reste d'entrainer effectivement et efficacement les modèles de réseau de neurones. En effet, ceux-ci exhibent des courbures pathologiques qui nécessiteraient l'emploi de méthode du second ordre afin d'être prise en compte, ce qui est malheureusement actuellement trop couteux du fait du nombre de paramètres qui est destiné à être grand.

L'idée est donc de se fixer un critère de type risque empirique ou maximum de vraisemblance selon le cadre dans lequel on se trouve, ce qui se réduit toujours à l'optimisation d'une fonction $E(\theta)$ où θ est l'ensemble des paramètres du modèle. Le nombre de paramètres étant toujours extrêmement élevé, les méthodes du second ordre impliquant notamment une inversion de la Hessienne sont totalement prohibées. On est alors réduit à toutes les méthodes du premier ordre de type gradient stochastique afin de faire face à des bases de données où le nombre d'exemplaires d'entrainement explose.

On calcule donc le gradient de E par rapport à ses paramètres. Ce calcul n'est pas toujours simple comme on le verra dans le cas particulier du modèle auquel on s'intéresse pour lequel on explicitera le calcul complet. En effet, du fait de la structure de couche du réseau (et en plus du caractère récurrent dans le cas qui nous intéresse), le calcul du gradient se fait par rétropropagation, c'est à dire que l'erreur se rétropropage du haut du réseau vers le bas. En effet, dans la mesure où les valeurs au nœuds supérieurs sont fonctions de celles aux niveaux inférieurs, on voit bien qu'en dérivant on va devoir propager de haut en bas ces gradients.

Les principes directeurs communément accepté à l'heure actuelle sont les suivants (voir également [7], ou les dernier travaux du groupe de Yann Le Cun et notamment le dernier article "No more pesky learning rate" :

1) Une bonne stratégie d'initialisation des paramètres : c'est le premier progrès réalisé par Hinton en 2006 qui a eu l'idée d'entrainer ses réseaux couche par couche et d'utiliser la sortie pour entrainer la couche suivante. Cette stratégie dite de "greedy layerwise training" permet d'obtenir la plupart du temps une initialisation raisonnable du modèle, qui on l'a constaté est expérimentalement capitale pour obtenir des résultats corrects.
2) Pour stabiliser les méthodes de type gradient stochastique, il est également préférable d'utiliser à chaque itération un ensemble dit mini-batch de quelques exemplaires pour obtenir une estimation moins bruitée du gradient plutôt qu'un seul exemplaire.
3) On utilise alors un algorithme du premier ordre de préférence assez robuste aux valeurs choisies pour les paramètres, typiquement le learning rate dont va grandement dépendre le résultat du modèle. Une méthode à learning rate fixé a peu de chance d'être fructueuse, et il est conseillé d'utiliser des méthodes d'adaptation du learning rate avec les itérations, par exemple RPROP [8], RMSPROP (Non publié, voir cours de machine learning de Geoff Hinton), AdaGrad [9] ou encore les méthodes de Newton tronquées (ou Hessian-free optimization, voir Mertens et al.[10]).
4) Il est enfin recommandé d'utiliser des architectures qui vont probablement être à la limite de l'overfitting, cependant ici c'est considéré comme une bonne chose, et pour éviter que cela arrive effectivement, on emploie des techniques telles que le dropout (consistant à aléatoirement éteindre des noeuds du réseaux pour éviter la coadaptation des unités), ou encore l'early stopping.

On mentionnera également des méthodes d'accélération de type momentum, gradient accéléré de Nesterov et autres. Ces méthodes sont subtilement reliées les unes aux autres comme l'explique un article non publié de Ilya Sutskever ("On the importance of momentum and initialization in deep learning")expliquant les rapports entre ces méthodes de gradient accéléré et des méthodes de type hessian-free.

II. Exposition de la problématique

Suivant le travail de Richard Socher, nous avons voulu appliquer une technique de deep learning réputée à l'état de l'art sur l'analyse de texte, et plus particulièrement sur l'analyse de sentiments.

5) Classification de sentiments: Comme expliqué dans l'article [1], si la classification binaire peut paraître un peu restrictive pour capturer les différentes nuances de langage, un modèle à 5 classes : très négatif, négatif, neutre, positif et très positif permet de capturer la majeure partie de l'information nécessaire afin de classifier les sentiments. Cette observation se fait par rapport à l'annotation de la base de donnée employée (présentée ci-après), où les annotateur disposaient de 7 niveaux de notation des sentiments, et n'utilisaient finalement majoritairement que le neutre et les extrêmes. Ceci a donc poussé à considérer un problème à 5 classes qui capture donc l'essentiel de la variabilité des données.

Ces observations se voient clairement sur la figure de l'article que l'on reprend en figure 5

L'autre observation qu'il est importante de faire est celle du sentiment ressenti pour les mots seuls : on constate qu'essentiellement les mots sont neutres sauf exception, et les nuances de sentiments apparaissent proportionnellement à la longueur des n-grammes. Ceci explique deux faits :

— D'une part le succès des approches de type sac de mots sur de longs textes qui permettent de capturer les sentiments par la présence de nombreux mots associés à des sentiments forts, et diluant ainsi les tournures subtiles et négations complètes.
— D'autre part l'échec de ces mêmes approches pour des phrases plus courtes (Twitter) dans lesquelles le nombre d'occurrences de mots traduisant clairement des sentiments est plus faible, et surtout qui peut être complètement mis en défaut par une négation ou double négation en tête de subordonnée par exemple.

On voit donc apparaitre la nécessite de prendre en compte non-seulement le sentiment que traduisent les mots, mais surtout la structure de la phrase, ceci incluant les nuances subtiles telles que les négations. Pour cette raison on s'intéresse à des modèles de réseaux de neurones dits récurent, qui permettent de prendre en compte la structure de la phrase (d'où le terme de récurrence).

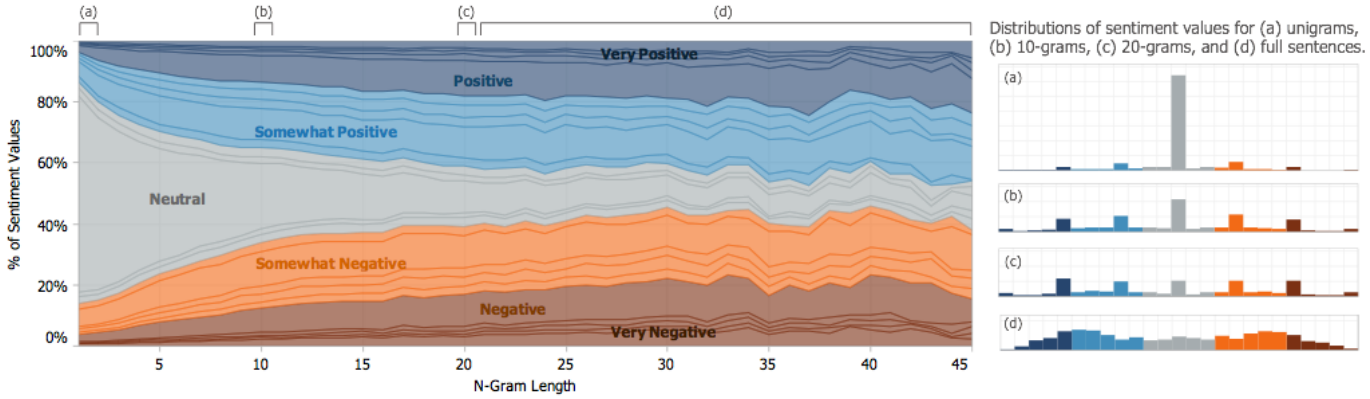


FIGURE 5: Observation de classification humaine sur la base sentiment treebank

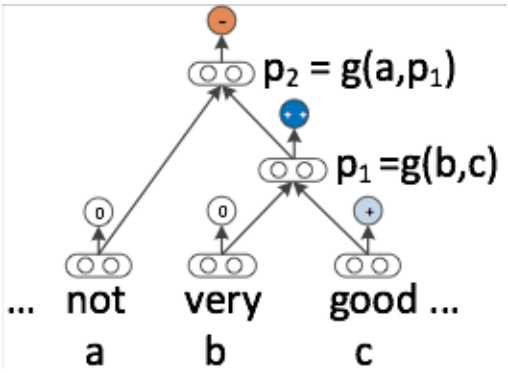


FIGURE 7: Schéma des réseaux de neurones récursifs

6) Base de donnée sentiment treebank: Pour cette article, Socher et al. ont introduit une nouvelle base de donnée pour les sentiments. Comme expliqué ci-dessus, il s'agit d'une base de données comportant des phrases extraites de critiques de films (rotentomatoes), qui ont été passées dans le parseur de Stanford pour construire un arbre qui représente la phrase et dans lequel chacun des nœuds a été étiqueté avec l'un des 5 labels en fonction du sentiment qu'il représente. On présente deux exemples en figure 6.

Cette base de donnée permet donc bien de prendre en compte les structures et subtilités telles que les négations de proposition comportant des termes positifs qui devient alors à connotation négative. Cela donne également la structure nécessaire à l'application d'un modèle récursif, mais demande donc de modifier quelque peu la structure d'un réseau de neurone pour prendre en compte la structure d'arbre. Nous l'exposons dans ce qui suit.

III. MODÈLE DE RÉSEAU DE NEURONE RÉCURSIF TENSORIEL

Un modèle de réseau de neurones récursif appliqué sur du texte se présente sous la forme de la figure 7.

Le principe est de construire un arbre à partir de la phrase à analyser, soit à l'aide d'un parseur comme celui de Stanford, soit en utilisant des structures fixes telles qu'un arbre de forme peigne à droite ajoutant un mot à chaque niveau. Bien sûr, pour être efficace, la structure doit prendre en compte la structure de la phrase et il est donc clair qu'utiliser le parseur plutôt qu'une structure fixe conduira à de meilleurs résultats.

Le point commun à tous ces modèles est le suivant : on choisit une représentation pour les mots (par exemple : vecteurs n-dimensionnels ou couples matrices vecteurs). Ensuite pour passer des nœuds disposant d'une représentation (les feuilles à la base qui sont des mots et qui l'ont donc déjà par construction du modèle), on utilise une opération de type neuronale pour construire la représentation des x^i (notation de la figure 7), ie pour le nœud i ayant les enfants i_1 et i_2 :

$$x^i = g(x^{i_1}, x^{i_2})$$

Où l'opération g est à définir en fonction du type de réseau choisi. On détaille ces deux points pour les trois modèles classiques (qui sont ceux présentés dans l'article de Socher)

Ensuite sur chacun des noeuds on ajoute un classifieur de type softmax :

$$x^i \longrightarrow y^i = \frac{exp(W_s x^i_j)}{\sum_j exp(W_s x^i_j)}$$

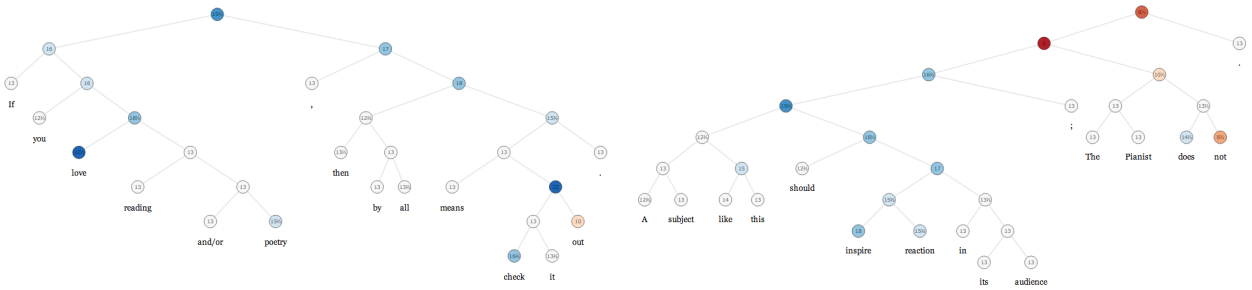
Où W_s est un paramètre du modèle qu'il faudra ajuster conjointement avec autres.

A. Modèles de la littérature

1) Réseau récursif simple: Dans ce modèle on choisi une représentation d-dimensionnelle pour les mots, et on utilise une opération neuronale non modifiée pour calculer la représentation des nœuds supérieurs à partir de leurs enfants (ici on choisit de travailler sur des digrammes généralisés, cependant le raisonnement est exactement le même si l'on souhaite utiliser des trigrammes ou des modèles plus larges) :

$$x^i = g(x^{i_1}, x^{i_2}) = f\left(W \begin{bmatrix} x^{i_1} \\ x^{i_2} \end{bmatrix} + b\right)$$

Où $W \in \mathbb{R}^{d \times 2d}$ et $b \in \mathbb{R}^d$ et f est la fonction d'activation du réseau. Dans ce modèle, l'essentiel de l'information est contenu dans la matrice W qui va encoder les relations de



(a) Arbre 20 (b) Arbre 311

FIGURE 6: Deux exemple d'arbres parsés et étiquetés.

compositionalité, ce qui n'est cependant pas tout à fait satisfaisant car ne laisse pas énormement de place pour capturer des structures subtiles. Cependant il est suffisamment peu complexe pour être entrainé correctement.

2) Réseau récursif vecteur-matrice: Pour ce modèle, au lieu de choisir une représentation d-dimensionnelle simple, on cherche à encoder les différentes relations de compositionnalité entre les mots dans une matrice associée dans la représentation de chacun d'entre eux. Ainsi les mots sont représentés par couple (x^i, X^i), où $x \in \mathbb{R}^d$, et $X^i \in \mathbb{R}^{d\times d}$ et les opérations à effectuer pour passer au niveau suivant sont alors :

$$x^i = g(x^{i_1}, x^{i_2}) = f\left(W \begin{bmatrix} X^{i_2}x^{i_1} \\ X^{i_1}x^{i_2} \end{bmatrix} + b\right)$$

$$X^i = g_M(X^{i_1}, X^{i_2}) = f\left(W_M \begin{bmatrix} X^{i_1} \\ X^{i_2} \end{bmatrix} + b_M\right)$$

avec W, W_M, b, b_M à nouveau les paramètres du modèle. Ce modèle a le mérite d'encoder les relation de compositionalité, mais ceci a un coup très élevé : pour chaque mot (le dictionnaire est destiné par nature à être large), il faut avoir un couple dont le nombre total de coefficients est $d + d^2$. Ceci peut rapidement exploser en terme de nombre de paramètre, et surtout en terme d'entrainement cela pose un gros problème car il faut optimiser sur tous ceux-ci, ce qui devient extrêmement difficile à faire efficacement avec des méthodes du premier ordre de type gradient stochastique.

3) Réseau récursif tensoriel: Ce modèle est celui qui a été retenu pour faire les essais puisqu'il allie la simplicité du modèle à une possibilité d'encoder les relations de compositionnalité de façon relativement explicite et ceci pour un coup guère plus élevé que le réseau récursif simple, et beaucoup moins élevé que pour un réseau récursif matrice-vecteur. On choisit donc de même que le simple un représentation des mots dans $\mathbb{R}^d$. La différence majeure se fait dans la fonction g où on ajoute un terme tensoriel censé prendre en compte l'aspect de compositionnalité, conjugué au terme linéaire simple. En bref :

$$x^i = f\left(\begin{bmatrix} x^{i_1} \\ x^{i_2} \end{bmatrix}^T V^{[1:d]} \begin{bmatrix} x^{i_1} \\ x^{i_2} \end{bmatrix} + W \begin{bmatrix} x^{i_1} \\ x^{i_2} \end{bmatrix}\right)$$

Où $W \in \mathbb{R}^{d\times 2d}$, V est un tenseur et pour $h = \begin{bmatrix} x^{i_1} \\ x^{i_2} \end{bmatrix}^T V^{[1:d]} \begin{bmatrix} x^{i_1} \\ x^{i_2} \end{bmatrix}$ on a : $h_j = \begin{bmatrix} x^{i_1} \\ x^{i_2} \end{bmatrix}^T V^{[j]} \begin{bmatrix} x^{i_1} \\ x^{i_2} \end{bmatrix}$, c'est à dire $V \in \mathbb{R}^{d\times 2d\times 2d}$. Il semble que ce modèle permet de trouver un intermédiaire entre un modèle trop simple, et un modèle trop complet difficilement entrainable. On voit dans la partie suivante comment effectuer l'entrainement, et en particulier le calcul des équations de rétropropagation.

4) Modèles de sacs de mots: Enfin, la dernière approche qui n'est pas du deep learning est celle des méthodes dites de sacs de mots consistant à fixer un vocabulaire (éventuellement en le réduisant ou en construisant un dictionnaire plus élaboré qu'un simple comptage), puis construire pour chaque exemplaire à classifier un histogramme des mots de ce vocabulaire présents dans le texte. Après cela, on utilise ces histogramme pour entrainer un classifieur (SVM, Bayésien naïf) utilisé ensuite sur les exemplaires de test. Cette démarche peut également être utilisée exactement de la même manière en augmentant le nombre de mots utilisés simultanément, en choisissant par exemple de travailler sur des bigrammes ou des trigrammes. Cependant comme on peut le voir il s'agit d'une estimation ne reposant ni sur la structure, ni l'ordre des mots présents. L'approche est cependant assez efficace, notamment sur de long textes.

B. *Critère d'entrainement*

Dorénavant on s'intéresse exclusivement pour nos développements au réseau de neurone tensoriel récursif, cependant le raisonnement s'applique exactement de la même manière aux autres approches.

En notant $\theta = (W, W_s, V, L)$ les paramètres du modèle (W le terme linéaire, V le tenseur, W_s le paramètre du softmax et enfin L le dictionnaire) on définit l'énergie suivante à partir de la distance de Kullback-Leibler (le signe moins est bien présent contrairement à la formule donnée dans l'article) :

$$E(\theta) = -\sum_{i\in Noeuds}\sum_{j=1}^{C} t_j^i \log y_j^i + \lambda||\theta||^2$$

Pour simplifier les notations, on réservera l'index i aux nœuds du réseau.

C. Équations de rétropropogation

On notera en préliminaire avec le softmax :

$$\sigma_j(y) = \frac{e^{y_j}}{\sum_k e^{y_k}}$$

que la dérivée est simplement :

$$\partial_i \sigma_j(y) = (\delta_{i,j} - \sigma_i(y))\sigma_j(y)$$

1) Gradient par rapport à W_s: On fait donc le calcul de la dérivée par rapport à $[W_s]_{kl}$ qui est différent des suivants puisqu'il ne s'agit pas encore de rétropropagation $y_j^i = \sigma_j(W_s x^i)$:

$$\begin{aligned}
\frac{\partial E}{\partial [W_s]_{kl}} &= -\sum_i \sum_j \frac{t_j^i}{y_j^i} \frac{\partial y_j^i}{\partial [W_s]_{kl}} \\
&= -\sum_i \sum_j \sum_{c=1}^{C} t_j^i \delta_{kc} (\delta_{cj} - \sigma_c(W_s x^i)) x_l^i \\
&= -\sum_i \sum_j t_j^i (\delta_{kj} - y_k^i) x_l^i \\
&= \sum_i (y_k^i - t_k^i) x_l^i
\end{aligned}$$

Donc chaque nœud i contribue au gradient par un terme $(y^i - t^i)(x^i)^T$.

2) Gradients par rapport aux autre θ: Pour comprendre le terme de rétropropagation, on va dériver par rapports aux paramètres autres que W_s :

$$\begin{aligned}
\frac{\partial E}{\partial \theta} &= -\sum_i \sum_j \frac{t_j^i}{y_j^i} \frac{\partial y_j^i}{\partial \theta} \\
&= -\sum_i \sum_j \sum_c t_j^i (\delta_{pj} - y_p^i) \left[W_s \frac{\partial x^i}{\partial \theta} \right]_c \\
&= \sum_i \sum_j (y_j^i - t_j^i) \left[W_s \frac{\partial x^i}{\partial \theta} \right]_j \\
&= \sum_i (y^i - t^i) W_s \frac{\partial x^i}{\partial \theta}
\end{aligned}$$

Pour le nœud i on note i_g et i_d les indices des fils gauche et droite. On a donc :

$$x_j^i = f\left(\begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix}^T V^{[j]} \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix} + W_{j:} \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix} \right)$$

On voit donc que $\partial x^i / \partial \theta$ va comporter deux parties : une partie qui constitue le gradient en propre pour le noeud i, et une partie en facteur de $\frac{\partial}{\partial \theta} \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix}$ qui va se rétropropager aux fils x^{i_g} et x^{i_d}. On note $\delta^{i,s} = W_s^T (y^i - t^i)$. On a alors en considérant que la fonction d'activation a une dérivée simple en fonction d'elle même comme c'est le cas pour $\tanh f'(x) = F(f(x))$

$$\begin{aligned}
\frac{\partial x_j^i}{\partial \theta} &= F(x_j^i) \left(2 \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix}^T V^{[j]} + W_{j:} \right) \frac{\partial}{\partial \theta} \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix} \\
&+ \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix}^T \frac{\partial V^{[j]}}{\partial \theta} \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix} + \frac{\partial W_{j:}}{\partial \theta} \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix}
\end{aligned}$$

On voit apparaître en première ligne la partie qui se rétropropage aux fils et en deuxième ligne la partie propre au nœud i. Ainsi le gradient total va s'écrire :

$$\frac{\partial E}{\partial \theta} = \sum_i (\delta^{i,comp})^T \mathcal{E}^{i,\theta}$$

En notant $\mathcal{E}_j^{i,\theta}$ la partie propre au nœud i, et $\delta_{j,:}^{i,prop}$ la partie rétropropagée, soit :

$$\mathcal{E}_j^{i,\theta} = \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix}^T \frac{\partial V^{[j]}}{\partial \theta} \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix} + \frac{\partial W_{j:}}{\partial \theta} \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix}$$

$$\delta_{j,:}^{i,prop} = F(x_j^i) \left(2 \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix}^T V^{[j]} + W_{j:} \right)$$

Cette dernière partie rétropropagée se somme sur les j pour donner un total :

$$\begin{aligned}
\delta^{i,down} &= \sum_j \delta_j^{i,comp} \delta_{j:}^{i,prop} \\
&= \sum_j 2(\delta_j^{i,comp} F(x_j^i)) V^{[j]} \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix} + W^T (\delta^{i,comp} \otimes F(x^i))
\end{aligned}$$

Et on a donc les updates suivants pour calculer les $\delta^{i,comp}$ en notant p_i le père du noeud i qui qualifient l'algorithme de rétropropagation :

$$\begin{aligned}
\delta^{i,comp} &= \delta^{i,s} + \delta^{p_i,down}[1:d] \text{ Si i est le fils gauche} \\
&= \delta^{i,s} + \delta^{p_i,down}[d+1:2d] \text{ Si i est le fils droit}
\end{aligned}$$

Reste donc à calculer les différentes parties propres pour chaque paramètre.

3) Par rapport à W: La partie propre de la dérivée par rapport à W_{kl} s'écrit :

$$\frac{\partial W_{j:}}{\partial W_{kl}} \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix} = \delta_{kj} \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix}_l$$

D'où la partie propre (contribution des noeuds pères uniquement) :

$$\delta_k^{i,comp} \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix}_l$$

Soit pour le noeud i une contribution :

$$\frac{\partial E^i}{\partial W} = \delta^{i,comp} \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix}^T$$

4) Par rapport au tenseur V: De même la partie propre pour le paramètre $V_{kl}^{[j]}$:

$$\frac{\partial E^i}{\partial V_{kl}^{[j]}} = \delta_j^{i,comp} \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix}_k \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix}_l$$

Soit un total :

$$\frac{\partial E^i}{\partial V^{[j]}} = \delta_j^{i,comp} \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix} \begin{bmatrix} x^{i_g} \\ x^{i_d} \end{bmatrix}^T$$

5) Par rapport au dictionnaire L: Enfin pour les coefficients de L, seuls les feuilles font une contribution lorsque $x^i = L_k$, qui se lit directement :

$$\frac{\partial E^i}{\partial L_k} = \delta^{i,comp} \delta(x^i = L_k)$$

IV. Résultats expérimentaux

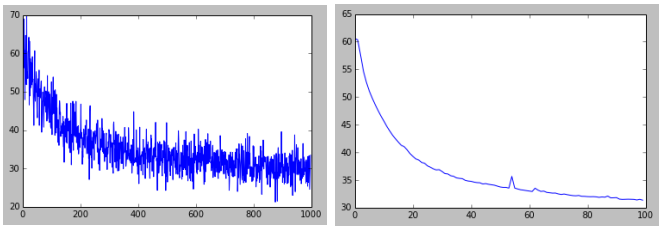
A. Méthodes d'entrainement retenues

Dans ce travail, on a voulu considérer deux méthodes d'entrainement : AdaGrad et RMSPROP qui sont deux méthodes online de gradient stochastiques applicables sur des minibatchs qui ont pour essence de réaliser une adaptation du learning rate au cours des itérations.

1) AdaGrad: Défini par Duchi et al. [9], AdaGrad est un schéma très simple présentant des garanties sous-linéaires dans les borne d'optimisation du regret dans le cas de l'apprentissage par renforcement. Il se résume de la façon suivante : à chaque itération t on calcule le gradient g_t, et on stocke la valeur $[a_t]_i = \sum_{t'=1}^{t} [g_t]_i^2$ et ce pour chacune des composantes j du gradient. On réalise alors l'update suivant :

$$\theta_j^{t+1} = \theta_j^t - \frac{\eta}{\sqrt{[a_t]_j}} [g_t]_j$$

Le learning rate η a alors une importance que très relative dans la mesure où il va seulement influer sur le poids dans les gradients de la première apparition d'une feature. L'idée derrière cet algorithme est de favoriser les features qui apparaissent rarement mais en étant très informatives par rapport aux features qui apparaissent très souvent sans donner beaucoup d'information. Ces dernières verront donc leur learning rate total beaucoup plus faible que pour les premières. On représente sur les courbes en figure 8 la progression de l'erreur avec les itérations.



(a) Erreur sur les mini-batch en fonction des itérations

(b) Erreur sur le set de validation avec les itérations

FIGURE 8: Erreurs en fonction des itérations pour AdaGrad

On note sur les courbes (qui sont uniquement des entrainements partiels) que les performance d'AdaGrad sont meilleurs que RPROP et la descente est beaucoup plus rapide à des valeurs intéressantes. C'est donc le mécanisme d'entrainement retenu.

2) RPROP et RMSPROP: RPROP et RMSPROP (Resilient backpropagation et resilient mean square backpropagation) sont deux variantes d'un même algorithme consistant à considérer uniquement le signe des gradients pour effectuer la descente. Le but est d'accélérer de plus en plus l'apprentissage dans les directions qui sont constamment choisies dans le même sens par le gradient, et de ralentir lorsque celui-ci change de signe.

Plus précisément RPROP se fait par construction sur des batch complets. Après avoir parcouru le training set et évalué le gradient total g_t, on effectue l'update $\theta^{t+1} = \theta^t - [\bar{g}_t]_\theta$ sur les paramètres où :

$$[\bar{g}_t]_\theta = \begin{cases} \eta_+ [\bar{g}_{t-1}]_\theta \text{ Si } [\bar{g}_{t-1}]_\theta \text{ et } [\bar{g}_t]_\theta \text{ sont de même signe} \\ \eta_- [\bar{g}_{t-1}]_\theta \text{ Sinon} \end{cases}$$

Où $\eta_+ > 1$ et $\eta_- < 1$ sont des constantes. Celles-ci traduisent les taux auxquels les vitesses de progression changent.

Le problème avec RPROP est qu'il ne s'applique pas à des versions mini-batch : en effet, l'utilisation du signe implique d'effectuer une opération du type $x/|x|$ sur le gradient total. Afin de réaliser la même chose en minibatch, il faut s'assurer que la constante reste la même sur tous les gradients, ce qui est impossible si celui-ci est recalculé à chaque minibatch. Ainsi, au lieu de diviser par la racine du carré pour obtenir la valeur absolue, on va plutôt conserver en mémoire une trace de la norme précédente, que l'on corrige un peu avec la nouvelle norme calculée. On a alors la variante RMSPROP en divisant par ces carrés moyens :

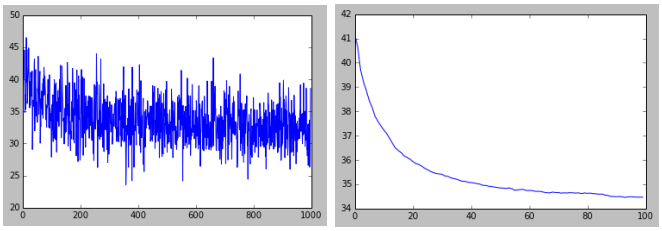
$$MeanSquare(\theta, t) = (1-\alpha) MeanSquare(\theta, t-1) + \alpha [g_t]_\theta^2$$

Avec habituellement $\alpha < 0.5$ et souvent $\alpha = 0.1$ (on ne souhaite pas que ce coefficient évolue trop rapidement sur le batch). On conserve la même stratégie d'update en tenant compte de ce point :

$$\theta^{t+1} = \theta^t - [\bar{g}_t]_\theta \frac{[g_t]\theta}{MeanSquare(t, \theta)}$$

Le terme dans la fraction étant censé donner quelque chose de l'ordre du signe du gradient (+1 ou -1 pour les coefficients donc). Cependant cette technique ne s'est pas montrée extrêmement efficace à l'usage, et a plutôt oscillé entre de mauvaises solutions comme le montre les graphes de la figure 9. Le comportement est représenté sur un millier d'itérations, mais le résultat est le même en allant plus loin on reste sur un plateau en oscillant. La courbe d'erreur avec les mini-batch est erratique du fait du gradient stochastique.

3) Choix des hyperparamètres: Pour le choix des hyperparamètres, du fait du temps nécessaire à l'entrainement complet d'un réseau de neurones (plusieurs heures avec notre code), nous sommes obligés de fixer un nombre d'itération volontairement trop faible pour obtenir un résultat optimal de l'ordre du millier, puis nous comparons les performances obtenues sur l'ensemble de train et de validation pour juger



(a) Erreur sur les mini-batch en fonction des itérations

(b) Erreur sur le set de validation avec les itérations

FIGURE 9: Erreurs en fonction des itérations

de la qualité des hyperparamètres utilisés. Pour notre modèle, les hyperparamètres sont :

— λ pour la régularisation L^2.
— η le learning rate de l'AdaGrad.
— La taille des mini-batchs à chaque itération.

Pour cette raison on effectue une recherche des paramètres optimaux sur de grilles de puissances de 10 (du fait du temps nécessaire à l'entrainement). La procédure d'entrainement étant très approchée, il est d'autant moins crucial de choisir des paramètres optimaux avec une grande précision, mais un ordre de grandeur correct est tout à fait suffisant.

4) Early stopping: En emploie une méthode dite UP_s issue de [11] (Livre "Neural network : a Trick of Trade"). L'idée est de regarder le nombre d'itérations consécutives au cours desquelles l'erreur sur l'ensemble de validation augmente. Cela se résume simplement ainsi *UP_s : si l'erreur de validation augmente sur s itérations consécutives, alors on arrête l'algorithme.*

Le choix de cette méthode a été fait en comparant avec l'erreur de généralisation dite GL_2 pour laquelle on arrête l'algorithme lorsque l'erreur de généralisation excède un seuil α donné :

$$GL_\alpha : \text{stop lorsque } 100(E_{val}(t)/E_{opt}(t) - 1) > \alpha$$

Ce dernier critère conduisait cependant à des arrêts beaucoup trop rapides puisque la courbe d'entrainement d'un réseau de neurones est beaucoup moins lisses que celles d'autre techniques.

Nous avons donc choisi un critère d'arrêt UP_s à $s = 3$ et une vérification toutes les 100 itérations (le calcul de l'erreur sur l'ensemble de validation rend l'itération beaucoup plus longue que sur un mini-batch, on doit donc le faire seulement à un certain intervalle pour rester rapides).

B. Résultats

Nous utilisons le même split entre train/validation/test que Richard Socher afin d'être certain de comparer les bons chiffres. Un entrainement complet prend entre 3 et 6 heures.

1) Problème à 5 classes: Nous ne sommes à ce jour pas parvenus à reproduire les résultats de l'article de Socher. Le calcul des gradients a été vérifié à de multiple reprise, que ce soit le calcul théorique ou le calcul numérique (comparaison par différences finies). Les valeurs des hyper-paramètres sont dans les range indiqués, la technique d'entrainement est la même, l'architecture du réseau également, et enfin les ensembles train/validation/set sont aussi les mêmes. Pourtant nous sommes toujours en deçà des performances mentionnées dans l'article, et sommes toujours en train d'essayer d'améliorer les résultats. Nous reportons les chiffres suivants à l'heure actuelle :

	Tous nœuds	Racines
Résultat	0.78	0.35

Avec la matrice de confusion suivante :

/	Très négatif	Négatif	Neutre	Positif	Très positif
Très négatif					
Négatif					
Neutre					
Positif					
Très positif					

2) Exemple de phrases:

V. CONCLUSION

RÉFÉRENCES

[1] R. Socher, A. Perelygin, J. Wu, J. Chuang, C. D. Manning, A. Y. Ng, and C. Potts, "Recursive deep models for semantic compositionality over a sentiment treebank," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Stroudsburg, PA : Association for Computational Linguistics, October 2013, pp. 1631–1642.

[2] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural Comput.*, vol. 18, no. 7, pp. 1527–1554, Jul. 2006. [Online]. Available : http://dx.doi.org/10.1162/neco.2006.18.7.1527

[3] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, Jul. 2006.

[4] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, "Greedy layer-wise training of deep networks," in *Advances in Neural Information Processing Systems 19*, B. Schölkopf, J. Platt, and T. Hoffman, Eds. Cambridge, MA : MIT Press, 2007, pp. 153–160.

[5] M. H. Hassoun, *Fundamentals of Artificial Neural Networks*, 1st ed. Cambridge, MA, USA : MIT Press, 1995.

[6] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, "Maxout Networks," *ArXiv e-prints*, Feb. 2013.

[7] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," *CoRR*, vol. abs/1206.5533, 2012.

[8] M. Riedmiller and H. Braun, "A direct adaptive method for faster backpropagation learning : The rprop algorithm," in *IEEE INTERNATIONAL CONFERENCE ON NEURAL NETWORKS*, 1993, pp. 586–591.

[9] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *J. Mach. Learn. Res.*, vol. 12, pp. 2121–2159, Jul. 2011. [Online]. Available : http://dl.acm.org/citation.cfm?id=1953048.2021068

[10] J. Martens, "Deep learning via hessian-free optimization." in *ICML*, J. Fürnkranz and T. Joachims, Eds. Omnipress, 2010, pp. 735–742. [Online]. Available : http://dblp.uni-trier.de/db/conf/icml/icml2010.html#Martens10

[11] L. Prechelt, "Early stopping - but when ?" in *Neural Networks : Tricks of the Trade, volume 1524 of LNCS, chapter 2*. Springer-Verlag, 1997, pp. 55–69.