Léonard Fromont
Pierre-Louis Lugiery
Benjamin Vallois

Projet TAL - Réseau de neurones pour tagging

Université Paris Diderot- Master 1 Linguistique Informatique
Juin 2019

I- Théorie

Dans cette partie nous commençons par présenter la tâche de TAL qui nous a occupé: en quoi consiste-t-elle, quels sont ses enjeux. Puis nous entrons dans la partie informatique du projet, la plus importante.

1) Un peu de linguistique pour commencer: le pos tagging.

a) Une définition simple du pos-tagging

L'étiquetage morphosyntaxique, ou, en anglais, le *part-of-speech tagging,* est une tâche classique en Traitement Automatique du Langage. Elle consiste à attribuer à chacun des mots d'une phrase passée en input la catégorie morphosyntaxique (*part-of-speech* en anglais) la plus appropriée à partir d'un ensemble de catégories considérées.

b) Enjeux: qu'est-ce qu'un bon étiquetage?

Soient la phrase: *"On pourra toujours parler à propos d'Averroès de décentrement du sujet."* (extraite du corpus fr-ud) , et l'inventaire de catégories morphosyntaxiques suivant: Pron, NOUN, Verb, Det, Adj, Adv, Punct, Aux, CCONJ , SCONJ, PropN,ADP (pour préposition).

Dans le modèle implémenté, c'est-à-dire avec l'ensemble des pos considérées ici et avec nos conventions d'étiquetage, la phrase étiquetée est la suivante:

(i) *"On/Pron pourra/AUX toujours/ADV parler/Verb à/ADP propos/NOUN d'/ADP Averroès/NOUN de/ADP décentrement de/ADP le/Det sujet/Noun ./PUNCT".*

Il faut insister sur le fait que cet étiquetage morphosyntaxique est le meilleur DANS NOTRE modèle. Il pourrait exister d'autres étiquetages tout à fait pertinents, et certains phénomènes méritent d'être soulignés ici.
En effet, considérons les étiquetages suivants:

-pourra/AUX parler/VERB: ici le verbe pouvoir est considéré comme un auxiliaire ce qui n'a rien de consensuel en français, contrairement à l'anglais. On aurait aisément pu imaginer un étiquetage du type: "*pourra/Verb parler/Verb",* voire l'introduction d'une catégorie spéciale pour les verbes à l'infinitif.
Ce choix concerne d'autres verbe à montée ou à contrôle du sujet; ainsi devoir et vouloir sont aussi considérés comme des auxiliaires.

-à/ADP propos/NOUN d'/ADP: ici, *à propos de,* qui aurait pu être considéré comme une locution prépositionnel à traiter comme un seul mot, a été considéré comme formé de trois mots indépendants et est analysé dans un sens compositionnel.
On aurait pu imaginer un étiquetage ou cette locution reçoit un et un seul tag; (*à propos de)/ADP*

-du -> de/ADP le/DET: l'occurrence de surface du est considéré dans sa forme sous-jacente c'est-à-dire comme la contraction de la préposition "de" et de l'article "le". Certains taggers font le choix d'attribuer à l'occurrence "du" soit la catégorie de préposition, soit celle de déterminant..

Pour travailler avec les outils du TAL nous avons fait le choix de nous fier aux conventions d'annotations choisies par les annotateurs humains qui ont étiqueté manuellement ce corpus. Nous aurons aussi confiance dans la cohérence des annotations: parfois des dizaines de linguistes annotent le même corpus et il est difficile (coûteux) de s'assurer que toutes ces conventions sont respectées à la lettre et que la séquence "*pourra parler"* ne sera pas étiquetée *"pourra/Verb parler/Verb"* (ou d'on ne sait quelle autre manière) dans la suite du corpus.

Néanmoins certains schémas d'annotations sont considérés comme étant l'état de l'art en matière d'annotation morpho-syntaxique pour le français, nommément le guide d'annotation French Tree Bank (Abeillé et al., 2003) ou encore le guide d'annotation Multitag (Villemonte De La Clergerie et al., 2008) (source: Rabary, Lavergne, Névéol 2015).

Il existe certainement d'autres exemples de phénomènes morphosyntaxiques qui font l'objet de choix, de débats et d'ambiguïté mais nous pensons que cet échantillon permettra au lecteur d'avoir un aperçu des enjeux linguistiques lié à l'étiquetage morphosyntaxique.

Ainsi donc, dans l'absolu, il existe DES étiquetages correctes pour une phrase donnée mais il y aura toujours UN étiquetage meilleur que les autres si l'on considère un modèle d'annotation. C'est exclusivement la deuxième partie de cette proposition qui nous intéressera dans la suite.

Ainsi on s'abstraira des enjeux linguistiques du pos-tagging et nous concentrerons sur l'aspect purement computationnel de la tâche; Comment évaluer la précision de notre modèle automatique?

c) Métrique d'évaluation:

Pour évaluer notre modèle on choisit d'utiliser la précision par token: quel pourcentage de mots a été bien annoté, c'est-à-dire, quel pourcentage de mots a reçu le même tag que dans une annotation humaine.

$$Acc = \frac{Nombre\ de\ mot\ bien\ annotés}{Nombre\ de\ mots}$$

Nous aurions pu choisir à la place le pourcentage de phrase bien annotées: dans cette métrique dès qu'un mot est mal annoté, la phrase entière est considérée comme mal annotée.

La formule de la précision devient:

$$Acc = \frac{Nombre\ de\ phrases\ bien\ annotés}{Nombre\ de\ phrases\ dans\ le\ corpus}$$

Nous venons donc de présenter la tâche de tal qui nous a occupé pour ce projet. Nous allons maintenant présenter l'approche technique et informatique du problème.

II) Le vif du sujet: la théorie des réseaux de neurones

1) Les classifieurs

La tâche de pos tagging est réalisée à l'aide d'un classifieur. Un classifieur est un objet qui associe à un objet passé en input la catégorie qui d'après ce qu'il a **appris**, a la plus grande probabilité d'être la catégorie de cet objet. Cet objet passé en entrée doit respecter un certain format dicté par le modèle de sorte d'être **interprétable en terme de caractéristiques.** Ce format permettra de réaliser des opérations mathématiques sur nos objets.
Nous verrons d'abord quelle est le domaine, la représentation mathématiques d'un classifieur, et quels sont ces paramètres. Nous verrons notamment quel est ce format de nos objets passés en input (ici chaque mot à tagger). Ensuite nous verrons de quelle manière le classifieur apprend, et ferons apparaître les fondements mathématiques de cette technologie. Nous aborderons enfin les avantages, les inconvénients et limites de notre modèle et proposerons des idées d'amélioration.

Cette partie s'appuie largement sur l'ouvrage de Yoav Goldberg: Neural Network Methods in Natural Language Processing.

a) Représentation vectorielle des objets

Le premier défi consiste à transformer nos objets linguistiques- nos mots, notre corpus- en objets mathématiques. Pour cela nous allons coder dans des vecteurs les caractéristiques (**features** dans la suite) des mots du corpus à partir d'hypothèses fortes et centrales en *NLP.* Ces features que nous avons choisi se basent sur ce que l'on appelle l'hypothèse distributionnelle: les mots qui apparaissent dans les mêmes contextes linguistiques partagent des significations similaires. En particulier cela veut dire que les mots partageant les mêmes contextes d'apparition auront la même catégorie morphosyntaxique. Ce que nous choisirons comme features sera:

Le mot précédent le mot à tagger, le mot à tagger, le mot suivant le mot à tagger.

Cette représentation en contexte permet de capter des similitudes entre des mots différents et constituera l'input de notre classifieur.

Reste à les transformer en objet mathématiques. Pour cela on associe à chaque mot du vocabulaire un indice propre. Prenons d'ailleurs un exemple.
Considérons le mini-corpus suivant, constitué de deux phrases:

1. Le chat dort
2. Le chien dort

“le” reçoit l’indice 0, “chat” reçoit l’indice 1, “dort” reçoit l’indice 2, “chien” reçoit l’indice 3.
Pour nos contextes nous aurons besoin de deux mots “bidons” supplémentaires: les mots qui codent le contexte “début de phrase” et “fin de phrase”. On peut voir une phrase de la manière suivante:

1. BEGIN Le chat dort END.

BEGIN et END reçoivent l’indice 4 et 5 respectivement.

Les mots sont donc codés par des entiers qui correspondent à leur indice.

Maintenant, en concaténant l‘indice du mot précédent, du mot suivant et du mot à tagger, on obtient une représentation vectorielle du contexte.

Dans “le chat dort”, le contexte pour chat sera la concaténation du vecteur one hot de “le”,”dort”,et “chat” dans cet ordre, soit: [1,3,2]. Ce vecteur constituera l’input de notre classifieur.

Nos occurrences ont désormais tous une représentation vectorielle dans $\mathbb{R}^3$**.** A partir de ces vecteurs, notre classifieur en construit d’autres, plus complexes: il concatène les représentations en plongement des mots (**embeddings)**. Ces représentations sont apprises à partir des contextes d’apparition des mots. Ils permettent de capter des similitudes sémantiques ce qui doit faciliter la tâche du classifieur; avant même de réaliser sa prédiction, le classifieur “sait” déjà que des mots comme pomme et poire partagent des contextes d’apparitions très proches.
Ces embeddings sont tous de même taille. Ainsi les occurrences de nos corpus sont tous représentés par la concaténation des vecteurs d’embeddings de 3 mots. Ceci va nous permettre de réaliser des opérations algébriques intéressantes. On note d la taille de ces vecteurs.

b) Notre classifieur

i) La fonction de prédiction

Nous classifions nos objets à l’aide d’un perceptron multicouche à activation non-linéaire ou encore réseau de neurone (RN dans la suite). Les réseaux de neurones sont une classe d’algorithme d’apprentissage automatique supervisé.
Pour bien comprendre le fonctionnement d’un RN, il faut commencer par sa version la plus simple: un **perceptron linéaire sans couche cachée**.

Conformément aux conventions de notations établies par Goldberg on note **x** l’input de notre classifieur, c’est-à-dire le vecteur d’observation d’un mot, et y sa classe gold (**x** est en gras et minuscule car c’est un vecteur, y est en minuscule car c’est un réel). On a défini dans la partie précédente le domaine mathématique de **x.** y quant à lui est l’indice de la classe gold parmi toutes les catégories morphosyntaxiques considérées.

Notre classifieur, étant donné, donc, un input **x,** commence par associer un score à chaque catégorie morphosyntaxique. Sans surprise, ces scores sont contenus dans un vecteur $\hat{\mathbf{y}} \in \mathbb{R}^c$, ($c = ¿ \textit{catégories possibles} ¿$. On parle d'un **perceptron multiclasse**.

C'est **ŷ** qu'on appelle la prédiction du classifieur.

Le calcul des scores se fait par utilisation d'une fonction linéaire f:

$$f(\mathbf{x}) = \mathbf{x} \cdot \mathbf{W} + \mathbf{b} \quad \text{(i)}$$
$$\hat{\mathbf{y}} = \mathbf{x} \cdot \mathbf{W} + \mathbf{b} \quad \text{(ii)}$$

$\hat{\mathbf{y}} \in \mathbb{R}^c$, $\mathbf{x} \in \mathbb{R}^d$, $\mathbf{W} \in \mathbb{R}^{d*c}$, $\mathbf{b} \in \mathbb{R}^c$

La classe prédite est:

$$\hat{y} = \text{argmax}(\mathbf{x} \cdot \mathbf{W} + \mathbf{b})$$

ŷ est la prédiction: un vecteur de taille nombre de classes, contenant les scores de chaque classe pour l'input x. Ceci apparaît mieux dans l'équation (ii).
x est le vecteur des embeddings concaténés. Ce n'est pas à proprement parler l'input du classifieur mais plutôt l'argument de la fonction.
W est la matrice de poids de notre modèle.
b est le vecteur contenant les biais appliquées à chaque classe.
x · **W** dénote le produit scalaire entre le vecteur d'observation **x** et la matrice de poids **W.** Ce produit scalaire est un vecteur. Nous y reviendrons dans la partie suivante.

Pour passer d'un vecteur de score qui associe un réel à chaque classe, à un vecteur de score qui lui associe une **probabilité** on passe le vecteur d'output de la dernière couche par la fonction softmax. La prédiction est en fait:

$$\hat{\mathbf{y}} = \text{softmax}(f(\mathbf{x})) = \text{softmax}(\mathbf{x} \cdot \mathbf{W} + \mathbf{b}) \quad \text{(iii)}$$

W et **b** sont les **paramètres** du modèle. Il convient maintenant de les introduire et d'expliquer comment ils vont être optimisés.

ii) Les paramètres du classifieur

On a fait l'hypothèse que la fonction f peut permettre de réaliser un tâche de classification sur un ensemble de données. On est donc à la recherche d'une fonction de la forme f(**x**)= **W.x+b**. Ici, **x** est fixé, ce sont nos observations. Il nous revient donc d'optimiser ce qui n'est pas fixé, c'est-à-dire **W** et **b**. Le rôle de l'apprentissage est de trouver les **W** et **b** tels que notre fonction se comporte comme on veut, c'est-à-dire qu'elle classifie bien nos données. **W** et **b** sont

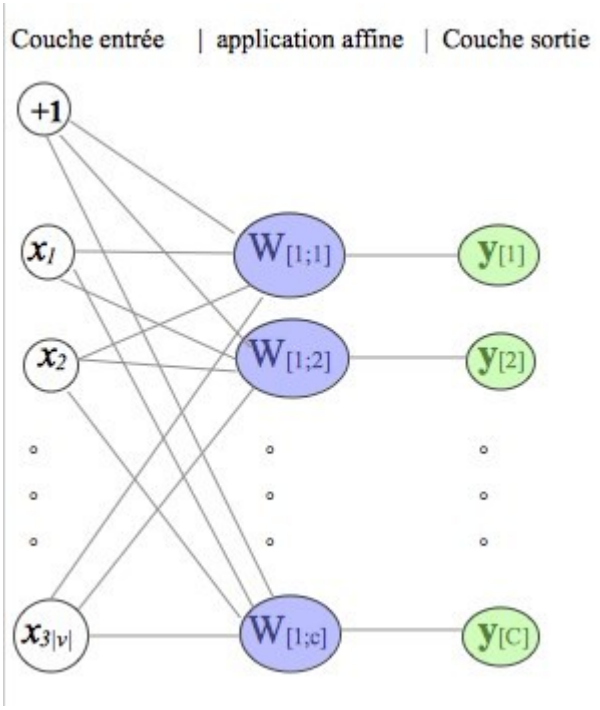
appelés les paramètres du modèle. On note les paramètres Θ . Dans la définition de la fonction, on peut faire apparaître explicitement ces paramètres:
f(**x**; Θ) = **x** · **W** + **b**. (Goldberg). On peut le lire comme f(**x**) étant donnés les paramètres Θ.
Pour définir nos paramètres on apprend sur un corpus d'entraînement. L'apprentissage est dit supervisé.

- On commence par initialiser ces paramètres sur des valeurs aléatoires. (Pour plus de précisions sur la méthode d'initialisation, voir la partie implémentation).
- On donne un mot à tagger à notre modèle. S'il se trompe on lui fait faire une mise à jour des paramètres. Sinon on ne fait pas de mise à jour. C'est la méthode passive, et c'est celle qu'on a retenu pour notre réseau de neurones, pour des raisons computationnelles.

Que représentent **W** et **b**?

W est la matrice de poids de notre modèle. Elle contient c colonne, une part classe. Chaque colonne est à lire comme le vecteur de poids de chaque classe. On ne dit rien du tout en disant qu'une catégorie morphosyntaxique a les contextes d'apparition des mots qui lui appartiennent; l'idée d'une représentation vectorielle des mots par leur contexte peut donc être étendue aux catégories morphosyntaxiques elles-mêmes. On va représenter ces catégories morphosyntaxiques en utilisant le même contexte, les mêmes features que pour les mots. Sans surprise chaque colonne de la matrice est un vecteur dans $\mathbb{R}^d$.
On retrouve le domaine mathématiques présenté dans la partie précédente:
$\mathbf{W} \in \mathbb{R}^{d*c}$.

Le deuxième paramètre **b** correspond aux biais c'est-à-dire un levier supplémentaire pour faire bouger nos hyperplans séparateur dans l'espace vectorielle. Le biais permet de passer d'une représentation linéaire à une représentation affine.
Ce schéma maison du fonctionnement d'un perceptron linéaire permet de comprendre un première fois la métaphore du réseau de neurone:



Dans notre schéma la couche la couche d'application affine contient les mêmes valeurs que la couche de sortie. Nous avons néanmoins fait apparaître 2 couches redondantes pour la clarté.

iii) Limite de l'application linéaire, et introduction des réseaux de neurones.

La limite de cette représentation affine apparaît lorsque l'on a affaire à des **données non linéairement séparables** c'est-à-dire, pour un espace vectoriel en 2 (respectivement 3) (respectivement plus de 3) dimensions, il n'existe pas d'ensemble de droites (respectivement plans) (respectivement hyperplans) qui sépare nos objets entre objets appartenant à la même catégorie. (pour un exemple précis, voir la partie sur XOR).

Les catégories morphosyntaxiques dans la représentation vectorielle qu'on en a choisit, et probablement dans l'absolu, sont précisément non linéairement séparables. Il nous faut introduire une modèle plus puissant, capable de produire des séparateurs non linéaires. Néanmoins les paramètres **W** et **b** introduits en section **II,1,b,i** restent les mêmes. Ce qui change c'est la fonction de prédiction, et la manière d'apprendre.

En fait, en s'inspirant du fonctionnement du cerveau humain, et plus particulièrement de l'activation de neurone, les chercheurs ont essayé d'utiliser une **fonction d'activation** non linéaire dans une couche cachée (celle du milieu). Cette architecture sera validé par le théorème d'approximation universelle (Hornik, 1991) qui dit qu'un réseau à une couche cachée avec un nombre fini et suffisant de neurones peut approximer toute fonction Lebesgue intégrable (les fonctions continues sur $\mathbb{R}^n$ le sont) . Bien que le théorème confirme l'existence d'un bon approximateur, il n'existe à l'heure actuelle aucun théorème pour définir ce réseau.
Dans la partie implémentation nous verrons qu'en pratique, on obtient de bien meilleurs résultats avec plusieurs couches cachées. Trouver un bon approximateur revient pour l'instant à

tester de nombreux hyper-paramètres, dont la fonction d'activation. On rentre alors dans le domaine nébuleux de l'apprentissage automatique.

Il existe une batterie de fonctions comme sigmoid, tanh ou relu, capables, en théorie, d'approximer ces fonctions non linéaires. Pour notre implémentation nous avons opté pour la fonction tangente hyperbolique (tanh), qui a l'avantage d'être centrée en 0 , contrairement à sigmoid dont l'intervalle image est [0;1]. Cette fonction est continue et dérivable sur son ensemble de définition $\mathbb{R}^n$.

Le vecteur d'output de notre perceptron linéaire correspond désormais au vecteur des valeurs dites de préactivation de la première couche cachée de notre réseau de neurone. C'est sur ces valeurs de préactivation que l'on va appliquer la fonction tangente hyperbolique. On note z[i;j] la valeur de préactivation du neurone j de la couche i. On note a[i;j] la valeur d'activation du neurone j de la couche i.

Ainsi, dans notre réseau de neurone, la prédiction se fait par la fonction

$$\mathbf{f(x)=} \tanh(\mathbf{x} \cdot \mathbf{W} + \mathbf{b}) \quad \text{(i)}$$

et devient

$$\hat{\mathbf{y}} = \text{softmax}(\tanh(\mathbf{x} \cdot \mathbf{W} + \mathbf{b})) \quad \text{(ii)}$$

La classe prédite devient: $\hat{y}$= argmax (softmax(tanh($\mathbf{x} \cdot \mathbf{W} + \mathbf{b}$))).

Les domaines de définitions restent les mêmes. $\hat{\mathbf{y}}$ est toujours le vecteur des scores (par probabilité) de chaque classe pour l'input x, étant donnés les paramètres Θ=**W**, **b.**

La partie de l'algorithme qui calcule la prédiction en avançant de la couche d'input à la couche d'output est appelée la passe avant *(forward propagation).* La partie de l'algorithme qui va réajuster les vecteurs de poids est appelé la passe arrière *(backward propagation).* Avant de la présenter, il convient d'expliciter en quoi consiste mathématiquement un problème d'optimisation.

2) Optimisation et apprentissage

a) La notion de perte

Dans l'idéal, étant donné un jeu de données d'entraînement ($x_{1:n}$, $y_{1:n}$¿, notre classifieur doit renvoyer les prédictions $y_{1:n}$ pour les inputs $x_{1:n}$. Les fonctions de pertes permettent de quantifier à quelle point notre classifieur se trompe lorsqu'il fait une prédiction. A-t-il donné un très gros score à la classe adverbe pour l'input "tortue", et un faible score pour la classe "nom"? N'a-t-il que péniblement attribué la classe de verbe à l'input "partons"? C'est ce que la fonction de perte va nous dire.

La fonction de perte prend en argument un vecteur de prédiction et une classe gold et renvoie un réel. Plus ce réel est grand, plus la perte est grande, et à l'inverse lorsque la perte est nulle, le réel est nul.

Pour un exemple la perte s'exprime par:

$$L(\hat{y}, \boldsymbol{y}) = L(f(\boldsymbol{x}; \Theta), \boldsymbol{y})$$

1.1

On peut également définir la perte global sur tout le corpus étant donnés nos paramètres. C'est la moyenne des pertes pour chaque exemple.

$$\mathcal{L}(\Theta) = \frac{1}{n} \sum_{i=1}^{n} L(f(\mathbf{x}_i; \Theta), \mathbf{y}_i)$$

1.2

L'objectif de l'apprentissage est de minimiser cette perte globale. Formellement il faut faire une prédiction de nos paramètres

$$\hat{\Theta} = \underset{\Theta}{\operatorname{argmin}}\, \mathcal{L}(\Theta) = \underset{\Theta}{\operatorname{argmin}}\, \frac{1}{n} \sum_{i=1}^{n} L(f(\mathbf{x}_i; \Theta), \mathbf{y}_i)$$

1.3

Notre problème se réduit maintenant à un problème de minimisation d'une fonction de perte, i.e la recherche d'un minimum global ou local de fonction. La manière dont on va construire notre fonction de perte va,en principe, nous garantir de trouver un de ces minimums à l'aide des outils de l'analyse.

b) Notre choix perte

Pour notre modèle le choix de l'entropie croisée, ou encore negative log likelihood (NLL), s'impose.
On veut que nos paramètres maximisent la vraisemblance de nos données. La NLL permet de modéliser cet objectif (maxent). Sa formule est la suivante:

$$Lcross - entropy(\hat{y}, y) = -\log\left(\hat{y}_{[y]}\right)$$

1.4

Il faut minimiser cette fonction de perte.

c) Apprentissage par gradient

En combinant les équations (ii), et 1.2 et 1.4 il apparaît qu'il faut minimiser la perte moyenne qui s'exprime qui, dans le cas d'une réseau de neurone à une couche cachée à activation tanh et avec une activation softmax sur la couche de sortie, s'exprime par:

$$L(\Theta)=\frac{1}{n}\sum_{i=1}^{n}-\log\left(softmax\left(\tanh\left(x\cdot W+b\right)\right)\square_{y\square_{[i]}}\right)$$

Il faut donc étudier les variations de *L* en fonction de Θ. L'expression de L montre que celle-ci est dérivable comme fonction composée de fonction elle même dérivable (log, softmax, tanh, et l'application linéaire sont dérivables). Pour minimiser L il faut aller dans le sens opposé de son gradient, la matrice de ses dérivées partielles.
Comment varie L quand varie chacune des coordonnées Θ (les vecteurs de poids et biais de chaque classe)? C'est ce que nous dit le gradient.

Voici la méthode pour le calculer.

D'abord pour calculer le gradient de la perte.

$$\frac{\partial L}{\partial h_j{}^{out}} = \begin{cases} \frac{-1}{softmax(z^{out})_y} & \text{si j = y} \\ 0 \text{ sinon} \end{cases}$$

$$\frac{\partial L}{\partial z_j^{out}} = \frac{\partial L}{\partial h_j^{out}} * \frac{\partial h_j^{out}}{\partial z_j^{out}} \quad \text{et} \quad \frac{\partial h_j^{out}}{\partial z_j^{out}} = h_j^{out}\left(1 - h_j^{out}\right) \text{ si j = y}$$

$$\begin{cases} 1 - softmax(z^{out})_y \text{ si j = y} \\ 0 \text{ sinon} \end{cases}$$

Ensuite pour calculer le gradient des valeurs d'activation d'une couche k

$\forall k \neq out$

$$\frac{\partial L}{\partial h_i^k} = \sum_{j=1}^{n^{k+1}} \frac{\partial L}{\partial z_j^{k+1}} * \frac{\partial z_j^{k+1}}{\partial h_i^k}$$

Dérivée partielle au neurone z_j^{k+1}

On a $z_j^{k+1} = h^k . W_{[i,j]}^{k+1} + b_j$

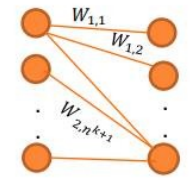
Or $\frac{\partial z_j^{k+1}}{\partial h_i^k} = W_{ij}$

car h_i^k apparait que une seule fois c'est donc égal au poids de coordonnée i,j dans la matrice W^{k+1}

$$= W_{[i,*]}^{k+1} . \frac{\nabla L}{\partial z^{k+1}}$$

Ensemble des dérivées de la couche z^{k+1}

$$\frac{\nabla L}{\partial h^k} \; de \; forme \; n^k * 1 = W^{(k+1)} . \frac{\nabla L}{\partial z^{k+1}}$$



h^k w^{k+1} z^{k+1}

Ensuite pour calculer le gradient des valeurs de préactivation d'une couche k

$$\frac{\partial L}{\partial j_i^k} = \frac{\partial L}{\partial h_i^k} * \frac{\partial h_i^k}{\partial z_i^k}$$

= dérivée partielle au neurone h_i^k * la dérivé de la fonction d'activation en z_i^k

Ici pas de somme car entre la couche de préactivation et la couche d'activation, chaque neurone à un fils.

On a donc

$$\frac{\nabla L}{\partial z^k} = \begin{cases} \frac{\partial L}{\partial h_1^k} * activation.dérivé\left(z_1^k\right) \\ . \\ . \\ \frac{\partial L}{\partial h_{n^k}^k} * activation.dérivé\left(z_{n^k}^k\right) \end{cases}$$

Enfin la mise à jour se fait comme suit

Toujours après avoir calculé $\frac{\nabla L}{\partial z^{k'}}$

On met à jour les paramètres $W^{k'}$ et $b^{k'}$

$$\frac{\partial L}{\partial W_{ij}^{k'}} = \frac{\partial L}{\partial z_j^{k'}} * \frac{\partial z_j^{k'}}{\partial W_{ij}^{k'}} \quad \searrow \quad \left(h_i^{k'-1}\right)$$

Ainsi $\frac{\partial L}{\partial z_j^{k'}} * \left(h^{k'-1}\right)^T$

Et $\frac{\nabla L}{\partial W} = \left(\frac{\nabla L}{\partial z^{k'}} . h^{k'-1}\right)^T$

$$\frac{\nabla L}{\partial b} = \left(\frac{\nabla L}{\partial z^{k'}}\right)^T$$

Car dans la combinaison linéaire, le facteur de b est 1

Cette méthode d'apprentissage s'appelle la descente de gradient.

Passons maintenant à l'aspect technique du projet. La partie qui suit va détailler notre implémentation.

II) L'implémentation

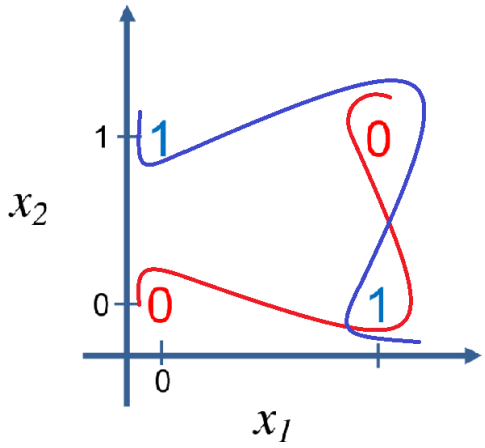
1) Un bon terrain de jeu: le problème du XOR

Pour comprendre le principe d'un réseau de neurones, nous avons dans premier temps établi des classes quelconques pour lesquels nous avons utilisé le XOR (sur 2 entrées) se traduisant par une table de vérité.

Pour s'assurer du bon fonctionnement de notre réseau de neurones, nous l'avons entraîné sur le problème du xor. Dans cet exercice, [x1,x2] est le vecteur d'input du réseau. Ces deux coordonnées sont des booléens. Notre classifieur doit décider si l'input est vrai ou faux. La table de vérité ci-dessous montre les conditions de vérité du XOR:

x1	x2	x1⊕x2
0	0	0
0	1	1
1	0	1
1	1	0

La figure ci dessous montre le caractère non linéairement séparable des données pour XOR, et justifie d'utiliser un réseau de neurone fonctionnel pour résoudre le problème.



Nous nous sommes servi du XOR pour valider notre système de passe-avant et passe arrière.

2) Notre code

a) Les formules

Notre code est composé de deux modules principaux: Network et Tagger. Le premier construit le réseau de neurones , le second permet d'exécuter la tâche du pos tagging..

Détaillons dans un premier l'architecture de notre réseau de neurone.
Network comprend les fonctions d'activation softmax(), relu() et tanh()

$$Softmax(U)_i = e^{U_i} / \sum_{j=1}^{c} e^{U_j}$$

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

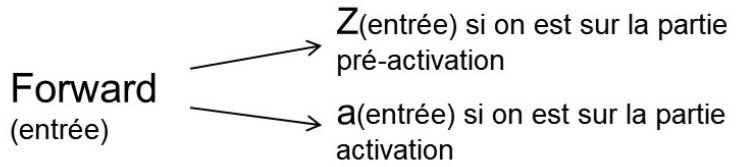
$$ReLU(z) = \max(0; z)$$

Avec Z∈ ℝ et U étant des combinaisons linéaire

Il comprend aussi la fonction de perte NLL, détaillée plus haut.

Pour l'architecture du réseau lui-même on utilise un classe "Layer", permettant de construire une couche dans notre réseau de neurones.

Cette classe comprend la fonction de la passe avant qui fait appel deux appels: un appel à la fonction de préactivation z (combinaison linéaire) suivi d'un appel à la fonction d'activation a (non linéaire).



La fonction rétropropagation est aussi présente dans la classe "Layer" elle fonctionne de la même façon que la passe avant mais elle nécessite le gradient. Si la couche est de pré-activation, le gradient est le produit terme à terme entre le gradient de la couche suivante et la dérivée de la couche suivante appliqué au vecteur de la couche actuelle. Sinon c'est le produit matriciel entre le gradient de la couche suivante et la matrice de poids de la couche suivante. **(cf section II,2,c.)**

Network contient aussi une classe qui crée une couche d'embeddings, elle dispose elle aussi d'une fonction de passe avant et de rétro-propagation. Étant donné qu'il n'y a pas de fonction d'activation ou de pré-activation sur la couche d'embeddings, ces 2 dernières fonctions retournent respectivement le vecteur concaténé des embeddings et le gradient de la première couche cachée par rapport aux embeddings.

Pour finir Network contient la classe SDG qui réalise la descente de gradient stochastique.

<u>En résumé:</u> la classe Network construit le réseau de neurone. Elle permet également de sauvegarder les hyperparamètres pour lesquels nous avons obtenu un bon résultat. On peut ainsi charger ces hyperparamètres directement lorsque l'on change de machine.

La seconde classe, Tagger, s'occupe de la tâche de tagging en elle-même: lecture des corpus au format conll, stockage des données extraites de ces derniers, construction des vecteurs d'input, et différents tests d'optimisation des hyperparamètres. Les principales fonctions permettant l'évaluation pour notre tagger, sont les fonctions "eval_***" qui permettent d'obtenir la perte ou l'accuracy de notre modèle pour une certaine combinaison d'hyperparamètres.

L'algorithme final commence donc par la conversion du vecteur en vecteur d'embeddings concaténés. Il se sert d'un dictionnaire qui associe à chaque indice son embedding. Il continue avec la passe avant: la fonction forward de notre classe network fait appel aux fonctions forwards des couches d'embeddings et des couches cachées, de proche en proche. A l'issu de cette passe avant, la perte est calculée. Ensuite la fonction SDG fait le

chemin inverse en faisant appel aux fonctions de backprop des couches respectives, de l'output à l'input et met à jour nos paramètres.

Enfin concernant l'initialisation de nos paramètres, les embeddings ont été initialisé aléatoirement avec la fonction np.random, ou bien la possibilité d'utiliser des embeddings déjà pré entrainés. Au sein de nos couches cachées, nos biais ont été initialisé à zéro et nos matrices de poids ont été initialisé avec une fonction dite de Xavier:

$$\mathrm{W}_{i,j} \sim U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right).$$

b) Problèmes et choix

Lors de l'utilisation au départ du XOR pour pouvoir comprendre le principe de la passe avant et de la backpropagation. Nous avons eu quelques problème avec le gradient qui tombait toujours à 0. Ceci était dû à la multitude de couches que l'on avait établies. En modifiant le nombre de couche à une, nous n'avions plus de problème de gradient. En effet, nous pensions que plus nous ajouterions meilleurs seraient nos résultats. Au contraire un trop grand nombre de couches faisait tomber le gradient à zéro, empêchant la mise à jour des poids, et par conséquent tout apprentissage.

Cependant certains hyperparamètres ne fonctionnaient qu'une fois sur deux sans doute à cause de l'initialisation des poids. Au final, avec un learning rate de 0.5, 1 couche cachée et 50 neurones, notre XOR était opérationnel.

Ensuite concernant le tagging, pour traiter les mots inconnus, nous avons mis en place un mot 'unk' pour identifier les mots non présents dans le train. Cependant cette méthode ne permettait pas l'apprentissage de l'embedding associé. Nous avons alors considéré les mots du train ayant une seule occurrence comme des inconnus, en faisant l'hypothèse qu'ils auraient une distribution proche de celles des mots inconnu.

En ce qui concerne l'apprentissage, nous avons opté pour l'utilisation d'early stop, en effet nous avions en premier temps mis le nombre d'epoch en hyperparamètre ce qui engendrait des itérations inutiles ou alors de l'overfitting (un bon résultat sur train mais aucune capacité de généralisation sur le dev, ou le test). Ainsi avec l'early stop, nous nous arrêtons quand la perte sur le dev diminue ne diminue plus au bout d'un certain moment, et nous récupérons les paramètres de l'epoch avec la meilleur perte. L'utilisation de la perte comme critère , au lieu de la précision, s'explique du fait que la perte sur une prédiction est plus informative que simplement bien/mal prédit.

Dans le tagging, il y'a un eu un problème d'implémentation qui ramenait nos gradient à zéro. Nous avons alors travaillé sur la théorie du "vanishing gradient" et les neurones saturés. Nous avons alors compris l'importance de l'initialisation des hyperparamètres dans l'évolution des valeurs au sein du réseau. Nous avons utilisé l'initialisation de Xavier. Mais notre problème était simplement un oubli de "+" dans le code. Nous avons tout de même garder l'initialisation de Xavier et nos modèles apprenaient enfin.

c) Les hyperparamètres

Pour pouvoir tester notre tagger, nous avons d'abord réalisé des apprentissages sur une seule phrase. L'idée étant que si un modèle n'apprend pas sur une phrase, il ne pourra pas apprendre sur un plus gros corpus non plus. Ces apprentissages étant trop nombreux, 3 nombres de couches cachées, 8 tailles de couche cachées et 8 learning rate, soit 192 evaluations, nous les avons regroupés sur un notebook (voir fichier joint dans l'archive)[1]. Puis nous avons sélectionné les combinaisons d'hyperparamètres qui atteignent une précision d'au moins 90% en moins de 20 epochs sur l'unique phrase . Cependant tous les modèles retenus possédaient au moins deux couches cachées.

Pour essayer d'avoir des modèles à une couche, nous avons modifié notre procédure en prenant 10% du train et une limite de 500 epochs. Conscient de la longueur de ces apprentissages, nous avons aussi rajouter une procédure d'early stop. Malheureusement aucune combinaison d'hyperparamètres avec une couche cachées ne nous a permis d'avoir une perte qui converge sous 2, et les précisions ne dépassant jamais 25%[2].

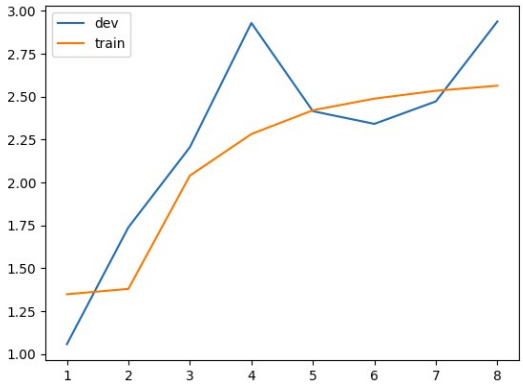
d) Expériences et résultats

Une fois la liste d'hyperparamètres obtenue, nous avons entraîné des modèles avec les hyperparamètres de cette liste sur le corpus train en entier, et avec early stop par rapport à la perte sur le dev. Voici un tableau récapitulatif des meilleures et pires pertes et accuracies sur le dev, lors des apprentissages sur le train. Le meilleur résultat étant en vert et le moins bon en rouge. Le tableau est trié dans l'ordre croissant par rapport à la perte sur le corpus dev. Chaque apprentissage ayant un temps d'exécution entre 2h et 4h.

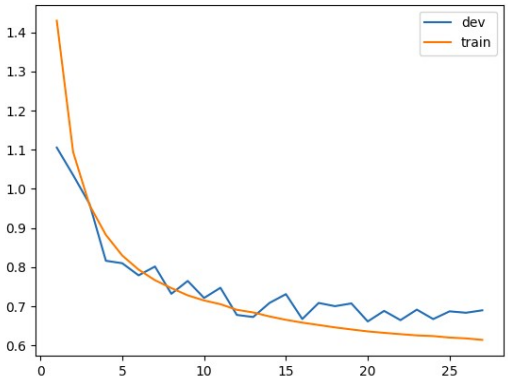
1 Etant donné la multitude des résultats, nous vous renvoyons vers la première partie du notebook pour le détail des graphiques et des valeurs obtenues.
2 Cf. Seconde partie du notebook pour les résultats avec 10% du train

Nombre de Couche Cachée	Nombre de Neurones	Learning Rate	Perte sur dev	Accuracies sur dev
3	300	0,01	0,66142	81,7
3	200	0,01	0,73332	79,3
3	100	0,01	1,01174	73,89
3	90	0,01	1,06037	72,45
3	80	0,01	1,10174	71,25
2	300	0,1	1,01794	69,93
3	70	0,01	1,31707	69,16
2	300	0,01	1,18285	68,74
3	300	0,1	1,05833	67,34
3	60	0,01	1,31617	66,99
2	200	0,01	1,32077	66,72
2	200	0,1	1,21711	64,13
2	100	0,01	1,43479	62,34
2	90	0,01	1,44	61,57
2	100	0,1	1,48999	59,08
3	200	0,1	1,37351	58,4
2	90	0,1	1,53	56,38
2	80	0,01	1,61	53,31
2	80	0,1	1,64	52,24
2	70	0,1	1,703	52,09
2	60	0,1	1,7	50,45
3	100	0,1	2,06065	34,66
3	70	0,1	2,27583	31,05
3	80	0,1	2,51082	27,62
3	90	0,1	2,35141	26,59
3	50	0,1	2,43398	23,05
2	300	1	2,72154	19,66
2	90	1	2,63	18,68
2	70	1	2,9	18,67
3	60	0,1	2,51463	18,62
2	50	1	2,902	16,19
2	80	1	2,71	15,6
2	60	1	2,46	10,8
2	100	1	2,74532	10,75
2	200	1	2,56196	10,75

Ces différents tests nous ont permis de voir les différents cas d'évolution de la perte en fonction du learning rate. Un exemple avec 2 modèles à 3 couches et chacune de 300 neurones



Learning rate de 0.1, trop grand



Learning rate de 0.01, converge bien

Sans pré-entrainement des embeddings, on atteint 81,34% d'accuracies sur le test, et 76,92% de précision sur les mots inconnus.
Et nous avons gardé le même modèle pour apprentissage avec des embeddings pré-entrainé et early stop sur dev encore. Notre modèle a atteint alors 90,62% d'accuracy sur le corpus test avec 79,48% pour les mots inconnus, ainsi qu'une perte de 0,37113.

Nous avons donc avec le meilleur modèle:

Pré-entrainement	Nombre de Couche Cachée	Nombre de Neurones	Learning Rate	Perte sur test	Accuracies sur test	Accuracies UNK
Non	3	300	0,01	0,70042	81,34	76,92
Oui	3	300	0,01	0,41181	90,78	57,91

Nous n'avons pas eu le temps de bien évaluer la robustesse de notre modèle, nous n'avons pu faire que 2 nouveaux entraînements pour chacun et les résultats sont similaires.
En revanche, nous constatons que la précision sur les mots inconnus est moyenne. Cela est peut-être dû à notre initialisation des embeddings ou peut-être de la trop grande proportion de mot inconnu dans le train. En effet seul 40% de mots apparaissent plus d'une fois. Une solution aurait pu être de faire des embeddings différents pour les mots inconnus en fonction de leur morphologie (présence de majuscule, suffixes, etc)

Pour finir voici d'autres idées d'amélioration que nous n'avons pas eu le temps de concrétiser:

- Utilisation de nouveaux features: suffixes, majuscule ou non en début de mot, tag du mot précédent.
- Test sur d'autres words embeddings pré-entrainés.
- Test sur d'autres langues.

IV- Manuel d'utilisateur

Pour exécuter le tagger sur le corpus:

- Placer les fichiers Tagger.py et Network.py dans le même dossier.
- Y ajouter les corpus train, dev et test souhaités.
- Le classifieur charge des hyperparamètres par défaut mais voici l'ordre dans lequel les initialiser le cas échéant. Entre parenthèses :(type, valeur par défaut, clef à entrer):
  - nombre de couches (entier, 3, '-nc')
  - nombre de neurones par couche (entier, 300,'-nn')
  - learning rate (réel, 0.01,'-lr')
  - nombre d'époques maximum (entier, 30,'-e')
  - nombre d'époque sans baisse de perte avant de procéder au early stop (entier, 5,'-p')
  - corpus train (conllu,'-train')
  - corpus test (conllu,'-test')
  - corpus train (conllu,'-dev'')
  - fichier des word2vec (aucun par défaut, '-vec')
  - nombre de features des word2vec (entier, 50,'-f')

- utilisation du gridsearch (booléen, False,’-gs’)
- affichage des résultats à chaque époque (booléen, faux,’-s’)
- sauvegarde du fichier (booléen, faux,’-sf’’).
- charger un modèle de fichier pkl pour le teste (fichier pkl, model.pkl, ‘-lo’)

Webographie

1. Le site deeplearningbook.org
2. Neural Network Methods in Natural Language Processing, Goldberg, 2017.