2m11.3198.4

Université de Montréal

Segmentation hiérarchique du domaine sémantique pour l'accélération d'un modèle de langage

11592163 V.040

par
Frédéric Morin

Département d'informatique et de recherche opérationnelle
Faculté des arts et sciences

Mémoire présenté à la faculté des études supérieures
en vue de l'obtention du grade de
Maîtrise es Science
Informatique

Avril 2004

Copyright © Frédéric Morin MMIV

QA
76
U54
2004
V.040

Université de Montréal

Direction des bibliothèques

AVIS

L'auteur a autorisé l'Université de Montréal à reproduire et diffuser, en totalité ou en partie, par quelque moyen que ce soit et sur quelque support que ce soit, et exclusivement à des fins non lucratives d'enseignement et de recherche, des copies de ce mémoire ou de cette thèse.

L'auteur et les coauteurs le cas échéant conservent la propriété du droit d'auteur et des droits moraux qui protègent ce document. Ni la thèse ou le mémoire, ni des extraits substantiels de ce document, ne doivent être imprimés ou autrement reproduits sans l'autorisation de l'auteur.

Afin de se conformer à la Loi canadienne sur la protection des renseignements personnels, quelques formulaires secondaires, coordonnées ou signatures intégrées au texte ont pu être enlevés de ce document. Bien que cela ait pu affecter la pagination, il n'y a aucun contenu manquant.

NOTICE

The author of this thesis or dissertation has granted a nonexclusive license allowing Université de Montréal to reproduce and publish the document, in part or in whole, and in any format, solely for noncommercial educational and research purposes.

The author and co-authors if applicable retain copyright ownership and moral rights in this document. Neither the whole thesis or dissertation, nor substantial extracts from it, may be printed or otherwise reproduced without the author's permission.

In compliance with the Canadian Privacy Act some supporting forms, contact information or signatures may have been removed from the document. While this may affect the document page count, it does not represent any loss of content from the document.

Université de Montréal
Faculté des études supérieures

Ce mémoire intitulé :

Segmentation hiérarchique du domaine sémantique pour l'accélération d'un modèle de langage

présenté par :

Frédéric Morin

a été évalué par un jury composé des personnes suivantes :

Douglas Eck
(président-rapporteur)

Yoshua Bengio
(directeur de recherche)

Guy Lapalme
(membre du jury)

Mémoire accepté le :
19 août 2004

Sommaire

L'application des réseaux de neurones à des problèmes de modélisation statistique du langage a déjà permis d'obtenir des résultats supérieurs par rapport à ceux de techniques habituelles. Cependant la modélisation du language par réseaux de neurones souffre du désavantage d'être fort gourmante en temps de calcul. Ceci s'explique par la grande quantité de sortie nécessaire, généralement une pour chaque mot du vocabulaire considéré.

On présente une technique générale applicable aux réseaux de neurones et permettant d'améliorer le temps de calcul d'un facteur $\mathcal{O}(\frac{|\mathcal{V}|}{log(|\mathcal{V}|)})$ ($|\mathcal{V}|$ est la taille du vocabulaire) tout en préservant largement les performances désirables des réseaux de neurones. En pratique cette méthode produit une accélération d'un facteur 320 sur le temps d'entraînement et d'un facteur 340 sur le temps de test. On présente également la version parallèle de cette technique amenant à une accélération encore accrue.

Mots clés : Apprentissage machine, réseaux de neurones artificiels, modélisation statistique du langage, clusterisation hiérarchique.

Summary

The application of neural networks to statistical language modeling has already given results superior to those obtained by traditional methods. Unfortunately, the computing resources required by neural networks has proven an important challenge to their general use in language modeling. This is explained by the large number of outputs required, typically one for each word of the considered vocabulary.

We present a technique applied to neural networks that improves the computing time by a factor of $\mathcal{O}(\frac{|\mathcal{V}|}{log(|\mathcal{V}|)})$ ($|\mathcal{V}|$ is the size of the vocabulary) while preserving the performances of neural networks. In practice the method improves training time by a factor of 320 and test time by a factor of 340. We also present the parallel version of the technique leading to even better speedup in computing time.

Keywords : Machine learning, artificial neural networks, statistical language modeling, hierarchical clustering.

Notations

Les notations suivantes sont utilisées :

a	Dénote une quantité scalaire
ω_i	Dénote la i^e quantité d'une séquence $\{\omega_1, \omega_2, \ldots, \omega_n\}$
$\mathbf{v}$	Dénote un vecteur
$\mathbf{v}_i$	Dénote la i^e composante du vecteur $\mathbf{v}$
$\mathbf{v}^i$	Dénote le i^e vecteur d'une séquence $\{\mathbf{v}^1, \mathbf{v}^2, \ldots, \mathbf{v}^n\}$
$\mathbf{v}^T$	Dénote la transposée du vecteur $\mathbf{v}$.
$\mathbf{M}$	Dénote une matrice
$\mathbf{M}_{i,j}$	Dénote la composante de la i^e rangée et j^e colonne de la matrice $\mathbf{M}$.
$\mathbf{M}^i$	Dénote le vecteur correspondant à la i^e rangée de la matrice $\mathbf{M}$.
$\mathbf{M}^T$	Dénote la transposée de la matrice $\mathbf{M}$.
$\mathbf{M} = \left[\begin{smallmatrix}\mathbf{M_1} \\ \mathbf{M_2}\end{smallmatrix}\right]$	Dénote la décomposition d'une matrice $\mathbf{M}$ de largeur W en deux sous-matrices aussi de largeures W.
$\mathbf{M} = [\mathbf{M_1}\ \mathbf{M_2}]$	Dénote la décomposition d'une matrice $\mathbf{M}$ de hauteur L en deux sous-matrices aussi de hauteurs L.
$\mathbf{a} \oplus \mathbf{b}$	Dénote la concaténation de deux vecteurs.

De façon générale, un exposant non-numérique dénote un terme d'une série tandis qu'un indice dénote un élément d'un vecteur ou d'une matrice. Pour des quantités scalaires, on réfère cependant aux éléments d'une séquence par un indice afin d'alléger la présentation. Les quantités en caractères gras représentent des vecteurs ou des matrices, on distingue ceux-ci par des lettres minuscules ou majuscules, respectivement. La dimension d'une matrice $\mathbf{M}$ est présentée ainsi : $(nombre_de_rangees) * (nombre_de_colonnes)$, où le nombre de rangée est toujours donné en premier lieu.

Table des matières

Sommaire i

Summary ii

Notations iii

Table des matières iv

Table des figures v

Liste des tableaux vi

1 Introduction 1
1.1 Paradigme et notations 2
1.2 Mesure de performance 3
1.3 Objectif de la présente recherche 5
1.4 Survol 5

2 Algorithmes existants 6
2.1 Trigrammes simples 6
2.1.1 Méthodes de lissage 7
2.2 Réseaux de neurones 9
2.2.1 Apprentissage machine 9
2.2.2 Fonctionnement des réseaux de neurones 10
2.3 Modélisation par réseaux de neurones 16
2.3.1 Ensembles de données 16
2.3.2 Espace des caractéristiques 17
2.3.3 NPLM 18
2.3.4 Parallélisation du NPLM 22
2.3.5 NPLM avec échantillonnage 25
2.4 WordNet 28

3 Modélisation par réseaux hiérarchiques 30
3.1 Décomposition sémantique 30
3.2 Espace des caractéristiques 32
3.3 Construction - clusterisation WordNet 32
3.3.1 *TF-IDF* 34
3.3.2 Mesures de similarité entre mots 35
3.3.3 Mesure de similarité entre *synsets* 36
3.3.4 Algorithme *K-Mean* (pour $k = 2$) 36
3.4 Détails de l'algorithme 37
3.4.1 Propagation avant 37
3.4.2 Propagation arrière 39
3.5 Parallélisation 41
3.6 Approximation 43
3.7 Mixture 44
3.7.1 Intégration à l'entraînement 44

4 Expérimentations et résultats 45
4.1 Bases de données 45
4.2 Nomenclature 46
4.3 Résultats comparatifs 47
4.3.1 Performance computationnelle 47
4.3.2 Généralisation 49
4.3.3 Efficacité 49
4.4 Résultats *APNews* 52
4.5 Discussion 52
4.5.1 Directions futures 53

Conclusion 57

A Produits vecteur-matrice 58

Bibliographie 59

Table des figures

2.1 Topologie d'un réseau de neurones 12

3.1 Enlèvement des parents multiples. 33
3.2 Binarisation de l'arbre hiérarchique. 34
3.3 Topologie d'un réseau de neurones hiérarchique 39

4.1 Comparaisons de l'erreur (NLL) en fonction du temps d'entraînement (s). . 51
4.2 Propriétés des noeuds 54
4.3 Senses de w_4 55

Liste des tableaux

4.1 Dimensions des ensembles de données pour la base de données *Brown*. 46
4.2 Dimensions des ensembles de données pour la base de données *AP96*. 46
4.3 Temps d'entraînement pour les algorithmes à base de réseau de neurones . . 47
4.4 Temps d'entraînement pour l'algorithme parallélisé *SymbCondHier-approx* . 48
4.5 Temps de test (ms) et (speedup) par exemple pour les algorithmes à base de réseau de neurones 48
4.6 Erreur de généralisation (NLL) sur les ensembles de validation et de test pour $|\mathcal{V}| = 10000$ 49
4.7 Temps de calcul afin d'atteindre l'optimal pour les algorithmes à base de réseau de neurones 50
4.8 Temps de calcul afin d'atteindre l'optimal pour l'algorithme *SymbCondHier-approx* parallèle 50
4.9 Erreur de généralisation sur les ensembles de validation et de test pour $|\mathcal{V}| = 10000$ 52

À mes parents pour leur amour, leur amitié et leur fierté

Chapitre 1

Introduction

La modélisation du langage (écrit) est un problème difficile dont les solutions présentent un grand intérêt pour nombre de champs de recherche. Si l'homme est capable de comprendre la machine, quoique de façon lente et pénible, la machine ne parvient pas encore à faire la réciproque malgré la puissance de calcul disponible. Cela est certainement dû, d'une part, à la complexité inhérente de l'interfaçage humain-machine. Mais, d'autre part, l'inefficacité et les limitations des algorithmes actuels sont aussi en cause. La présente recherche n'étudiera pas le problème de la communication humain-machine comme tel, mais se limitera au sous-problème de construire une modélisation de la langue écrite[1]. La construction de modèles de langues performants permettront, on l'espère, de réduire le fossé communicationnel qui existe entre l'homme et la machine.

Un des objectifs en modélisation du langage est de construire un modèle probabiliste à partir d'une séquence donnée de mots provenant de la langue naturelle et d'inférer la probabilité d'une séquence jamais vue. En d'autres termes, on désire construire un système permettant d'assigner une probabilité à une séquence de mots en utilisant comme connaissance préalable une séquence exemple. La modélisation s'arrête cependant à la structure syntaxique des mots d'une phrase, c'est-à-dire à donner une probabilité aux mots ayant une structure syntaxique (localement) correcte. Ainsi, aucun effort n'est fait pour valider ou pondérer la structure sémantique des mots, c'est-à-dire à donner une probabilité sur le sens d'une phrase.

Malgré ces limitations, un tel système peut ensuite être utilisé par d'autres applications. Par exemple, pour aider à la reconnaissance de la parole, à la reconnaissance textuelle automatisée, à la traduction automatisée ou à la correction d'erreurs. L'utilisation d'un modèle de langue permet, dans le cas de la reconnaissance de la parole, de mesurer la validitité des mots reconnus en leur attribuant une probabilité jointe. Un modèle de langue permet aussi de comparer la probabilité de mots alternatifs ou bien d'identifier les erreurs (probabilité jointe faible). L'idée est d'utiliser le modèle de langue en tant qu'approximation de la distribution probabiliste de séquences de mots et d'inférer la probabilité de séquences traitée par

[1] On définit la langue écrite informellement comme étant une suite de mots w_i d'une certaine longueur, chaque mot étant identifié par son indice i. w_i est un entier qui réfère à un mot du vocabulaire (voir plus loin).

un autre algorithme.

Tel que mentionné précédemment, la modélisation du langage est un problème particulièrement difficile. Cette difficulté intrinsèque provient du fait de la grande dimensionnalité de l'espace probabiliste qu'on considère. Si on suppose un vocabulaire $\mathcal{V}$[2] contenant un nombre $|\mathcal{V}|$ de mots distincts et qu'on étudie des séquences de mots de longueur L, on se retrouve avec la tâche de distribuer une masse de probabilité dans un espace de dimension $|\mathcal{V}|^L$. De façon à bien couvrir cet espace, c'est-à-dire à assigner des probabilités réalistes aux différentes occurrences de séquences possibles, il est nécessaire d'avoir une séquence exemple de dimension exponentielle dans le nombre de mots du vocabulaire. Lorsqu'une telle séquence exemple n'est pas disponible, on se retrouve avec une sous-couverture de certaines régions de l'espace. L'espace probabiliste ne peut donc être une représentation suffisamment fidèle du *vrai* espace des séquences de mots puisque de vastes régions ne sont pas, ou trop peu, couvertes. On cherche donc naturellement à abstraire une structure linguistique entre les mots de façon à permettre de distribuer de la masse de probabilité aux régions qui ne sont pas couvertes par la séquence exemple. Par exemple, en considérant des ensembles de mots synonymes, il est possible d'assigner de la probabilité globalement à un ensemble de synonymes au lieu de procéder sur une base de mots individuels. Ainsi, même si un mot w particulier n'a pas été rencontré dans la séquence exemple, mais que des synonymes l'aient été, il sera possible d'assigner une probabilité pour un séquence contenant ce mot w.

On peut donc dire que l'extraction des structures du langage est implicite (et nécessaire) à la construction d'un espace probabiliste réaliste. Cependant, de par le fait de la dimension finie de la séquence exemple qu'on peut utiliser pour construire cet espace, le modèle de langage souffrira de la nature contextuelle de la dite séquence. En d'autres mots, la séquence exemple ne peut être qu'un sous-ensemble plus ou moins restreint des possibilités offertes par le langage et ne permettra toujours pas une modélisation fidèle pour toutes les séquences de mots possibles. Il reste que malgré l'imprécision inhérente des modèles de langage qu'on peut construire de façon pratique, leur utilité est indéniable et la recherche de meilleurs modèles reste un défi stimulant.

1.1 Paradigme et notations

Soit

$$\{w_1, w_2, \ldots, w_L\}, v_{w_i} \in \mathcal{V}$$

une séquence ordonnée d'indices w_i faisant référence aux mots d'un vocabulaire $\mathcal{V}$[3]. On appellera *corpus* la séquence disponible à la modélisation de l'espace de probabilité que l'on veut construire. On subdivise en général cette séquence en trois sous-séquences, appellées *ensemble d'entraînement*, *ensemble de validation* et *ensemble de test*. L'ensemble d'entraînement est la séquence exemple mentionnée précédemment et est utilisé à la construction du modèle.

[2]Un vocabulaire $\mathcal{V}$ est un ensemble de mots v_j et $|\mathcal{V}|$ est la cardinalité de cet ensemble, c'est-à-dire le nombre de mots considérés. De façon formelle, $\mathcal{V} = \{v_1, v_2, \ldots, v_{|\mathcal{V}|}\}$.

[3]Afin d'alléger la présentation, on dira directement $w_i \in \mathcal{V}$

L'ensemble de validation est utilisé pour réguler la construction du modèle. L'ensemble de test n'est pas utilisé pour la construction comme tel mais sert à quantifier la performance du modèle sur une séquence *typique* de mots (assumée statistiquement indépendante des séquences d'entraînement et de validation) du langage que l'on veut modéliser. On appelle aussi la performance d'un modèle, l'*erreur de généralisation*. C'est une approximation (un estimateur non-biaisé) de l'espérance sur la fonction de performance[4].

On désire calculer la probabilité suivante :

$$P(w_1, w_2, \ldots, w_L) = P(w_1)P(w_2|w_1)\ldots P(w_L|w_1\ldots w_{L-1}) \tag{1.1}$$

pour une séquence arbitraire (de longueur donnée L) de mots pris d'un langage fixé (choisi implicitement par le corpus) et défini sur un vocabulaire $\mathcal{V}$. Étant donné la longueur potentiellement grande d'une séquence de mots et la grande quantité de combinaisons possibles, l'objectif sera de construire une approximation de cette probabilité. La performance du modèle construit est déterminée par la mesure de *perplexité*, définie ci-après.

1.2 Mesure de performance

La théorie de l'information, développée par Shannon [20], établit les fondements mathématiques de la communication symbolique ou numérique. Une des statistiques fondamentales de cette théorie est l'*entropie* qui sert à mesurer l'incertitude d'une source de communication[5]. Dans l'application qu'on étudie ici, la source "génère" une séquence de caractères (pris du vocabulaire $\mathcal{V}$) selon la fonction de probabilité f_X. Pour une séquence de longueur L, l'entropie (dénotée $H(X)$) se calcule ainsi :

$$\begin{aligned} H(X) &= -E_{f_X}\left[log_2\ f_X(w)\right] \\ &= -\sum_{i=1}^{L} f_X(w_i)\ log_2\ f_X(w_i), \qquad w_i \in \mathcal{V} \end{aligned} \tag{1.2}$$

On veut comparer la source f_X avec un modèle Q_X. Pour cela on peut utiliser l'entropie croisée :

$$H_C(X) = -E_{f_X}\left[log_2\ Q_X(w)\right]$$

que l'on peut estimer en remplaçant f_X par la distribution empirique, ce qui donne :

$$\hat{H}_C(X) = -\frac{1}{L}\sum_{i=1}^{L} log_2\ Q_X(w_i)$$

où $\hat{f}_X = \frac{1}{L}$ est la distribution observée dans la séquence. L'entropie croisée possède la caractéristique importante qu'elle est toujours supérieure ou égale à l'entropie de la source :

$$H_C(X) \geq H(X)$$

[4]Dans ce mémoire on utilisera la mesure de perplexité, définie à la section 1.2

[5]Une source de communication génère un flot de caractères d'un alphabet $\mathcal{V}$ selon une densité de probabilités f_X. Une source d'information est donc implicitement associée à une variable aléatoire X.

avec égalité lorsque $Q_X = f_X$. Tel que mentionné précédemment, l'entropie mesure l'incertitude d'une source, c'est-à-dire le degré d'incertitude qu'un observateur a par rapport aux caractères d'une séquence. Par exemple, si tout les caractères sont équiprobables ($Q_X(w_i) = \frac{1}{|\mathcal{V}|}, \forall w_i \in \mathcal{V}$) alors l'entropie est maximale et l'incertitude sur l'identité du prochain caractère l'est aussi. De même, si les caractères ont tous une probabilité nulle à l'exception d'un seul ($Q_X(w_1) = 1, Q_X(w_i) = 0$ pour $i \neq 1$) alors l'entropie est nulle ($log\ 1 = 0$) et le prochain caractère est connu avec absolue certitude.

Une mesure de performance qui dérive naturellement de l'entropie croisée est la perplexité. Cette mesure est régulièrement utilisée pour mesurer les performances de modèles de langage. L'équation suivante montre comment la perplexité est obtenue en terme d'entropie croisée :

$$\begin{aligned} PERP(X) &= 2^{\hat{H}_C(X)} \\ &= 2^{-\frac{1}{L}\sum_{i=1}^{L} log_2\ Q_X(w_i)} \\ &= \sqrt[L]{\frac{1}{\prod_{i=1}^{L}\ Q_X(w_i)}} \end{aligned}$$

La perplexité est ainsi l'inverse de la moyenne géométrique des probabilités. La perplexité mesure le degré de difficulté à prédire le prochain caractère. Par exemple, si tous les caractères sont équiprobables alors la perplexité est la suivante :

$$PERP(X) = \sqrt[L]{\frac{1}{\prod_{i=1}^{L}\ 1/|\mathcal{V}|}} = |\mathcal{V}|.$$

Dans cet example, la perplexité s'interprète comme le nombre des possibilités parmi lesquelles il faut choisir le prochain caractère. En général, la probabilité des caractères n'est pas uniforme mais l'interprétation de l'exemple précédent reste intuitivement utile. Ainsi, pour mesurer un modèle de langage, une faible perplexité indiquera la réussite du modèle à attribuer des probabilités, diminuant le nombre des possibilités parmi lesquelles choisir. Malgré tout, la perplexité sera toujours une quantité positive et non-nulle. Elle ne tendra non plus jamais vers zéro, indiquant l'incertitude inhérente dans l'utilisation du langage.

La mesure de performance que l'on utilisera pour rapporter les résultats présentés dans ce document est la perplexité. Cependant, au lieu d'utiliser la base 2, on utilisera la base naturelle e. Ainsi, la perplexité sera donc calculée de la façon suivante :

$$\begin{aligned} PERP(X) &= e^{\hat{H}_C(X)} \\ &= e^{-\frac{1}{L}\sum_{i=1}^{L} ln\ Q_X(w_i)} \end{aligned}$$

où $ln(\cdot)$ est la fonction logarithme naturel et où l'entropie a été définie implicitement avec la base e.[6]

[6]La base 2 est habituellement utilisée en raison de la propriété que l'*information* est exprimée en *bits*.

1.3 Objectif de la présente recherche

L'objectif de la recherche présentée dans ce mémoire est de développer les méthodes de modélisation du langage par réseaux de neurones [1], [2], [3]. Ces méthodes sont capables de bien modéliser l'espace de probabilité qu'on a présenté précédemment, mais souffrent de leur gourmandise en temps de calcul. On désire donc trouver des méthodes, basées sur les réseaux de neurones, qui permettent d'améliorer le temps de calcul nécessaire mais tout en conservant leurs performances de généralisation.

1.4 Survol

Le **chapitre 2** présente quelques algorithmes de modélisation statistique du langage existants. On fait un survol des méthodes classiques communément appellées *trigrammes* et des différentes améliorations que l'on emploie habituellement, notamment, le lissage. On introduit ensuite quelques notions d'apprentissage machine ainsi qu'une description générale des réseaux de neurones. Les réseaux de neurones seront présentés en termes d'une fonction paramétrisée capable d'approximer une fonction F (inconnue) à l'aide d'une série d'exemples. On présente finalement deux méthodes de modélisation du langage par réseaux de neurones ainsi que la version parallèle d'une de ces méthodes. Cela permettra d'introduire la notation utilisée au chapitre 3 ainsi que les détails des calculs effectués.

Le **chapitre 3** présente la méthode nouvelle développée au cours de la présente recherche. Elle se base sur les méthodes de réseaux de neurones présentées au chapitre 2 et sur l'outil linguistique *WordNet*. On décrit la décomposition sémantique hiérarchique qui rend possible l'accélération importante des calculs. On aborde également la question de la parallélisation de l'algorithme et on montre comment la décomposition hiérarchique y apporte une solution simple.

Le **chapitre 4** présente les résultats du modèle de langue proposé dans ce document et le compare aux autres algorithmes de modélisation utilisés. Les résultats comparatifs sont effectués sur une base de donnée étalon bien connue, *Brown*. On explore également les améliorations futures à apporter au modèle présenté dans ce mémoire.

Chapitre 2

Algorithmes existants

Ce chapitre présente différents algorithmes utilisés en modélisation du langage. L'algorithme le plus commun et de loin le plus utilisé est la modélisation par *trigrammes*. La section 2.1 donne les détails de cet algorithme. Le reste du chapitre est dédié aux méthode de modélisation du langage par réseaux de neurones déjà existantes. On débute, à la section 2.2, par une description générale des réseaux de neurones et des notions d'apprentissage machine requise pour le reste du document. On décrit en outre l'algorithme de rétropropagation de l'erreur généralement employé pour l'entraînement des réseaux de neurones. La section 2.3 présente deux types de réseaux de neurones employés pour la modélisation du langage. La description du premier algorithme, *NPLM*, permettra d'établir les bases nécessaires à la description du nouvel algorithme présenté au chapitre 3. Le second algorithme présente une amélioration de *NPLM* permettant une amélioration du temps de calcul lors de l'entraînement. Tous ces algorithmes seront ultérieurement comparés à la méthode présentée au prochain chapitre.

2.1 Trigrammes simples

Cette section présente la famille de techniques la plus couramment employée en modélisation du langage. Les techniques basées sur les *trigrammes* (ou plus généralement les *n-grammes*) sont relativement simples et peu coûteuses à mettre en oeuvre. Elles performent également relativement bien sur un large éventail de données de langage. On présente ici cette technique dans l'optique de s'en servir comme étalon de comparaison avec les méthodes basées sur les réseaux de neurones développés au cours de la présente recherche.

On désire toujours construire une approximation à la probabilité de l'équation 1.1 :

$$P(w_1, w_2, \ldots, w_L) = P(w_1)P(w_2|w_1)\ldots P(w_L|w_1\ldots w_{L-1})$$

L'approximation du trigramme se base sur l'hypothèse simplificatrice suivante :

$$P(w_i|w_1\cdots w_{i-1}) \approx P(w_i|w_{i-2}w_{i-1})$$

C'est-à-dire qu'on approxime les probabilités conditionnelles en se limitant à des probabilités sur des triplets de mots. La probabilité $P(w_i|w_{i-2}w_{i-1})$ est ainsi estimée à partir des fréquences des triplets présents dans l'ensemble d'entraînement (séquence exemple) de la façon suivante :

$$\begin{aligned} P(w_i|w_{i-2}w_{i-1}) &= \frac{P(w_{i-2}w_{i-1}w_i)}{P(w_{i-2}w_{i-1})} \\ &\approx \frac{C(w_{i-2}w_{i-1}w_i)}{C(w_{i-2}w_{i-1})} \end{aligned} \tag{2.1}$$

où $C(\cdot)$ est la fonction qui compte les occurences de triplets et de paires dans la séquence exemple.

En pratique l'approximation du trigramme souffre d'une déficience majeure. Elle est très dépendente de l'ensemble d'entraînement utilisé. Ainsi, des ensembles différents produiront une estimation différente de la distribution de probabilité que l'on cherche à modéliser. Cela s'explique par le fait que beaucoup de triplets de mots ne se retrouveront pas dans l'ensemble d'entraînement (et auront une probabilité estimée nulle) malgré le fait qu'ils soient valides. Pour contourner ce problème, on fait appel aux méthodes de lissage qui permettront, comme le nom l'indique, d'adoucir l'approximation en redistribuant de la masse de probabilité dans des régions non couvertes par l'ensemble d'entraînement. Cela permettra de réduire la variabilité de la modélisation par rapport à l'ensemble d'entraînement.

2.1.1 Méthodes de lissage

Il existe une grande quantité de méthodes de lissage. Le problème que ces méthodes cherchent à résoudre est le fait que l'ensemble d'entraînement est insuffisant pour couvrir la totalité des possibilités de triplets de mots. Par contre, le même ensemble est beaucoup plus apte à estimer les probabilités pour des paires de mots (que l'on dénomme *bigrammes*) et également pour des mots uniques (singletons ou *unigrammes*). L'essence de ces méthodes est alors de combiner l'information (déficiente et bruitée) des *trigrammes* avec l'information plus *stable* des bigrammes et des unigrammes.

Interpolation linéaire

La méthode la plus simple est l'interpolation linéaire entre le trigramme, le bigramme et l'unigramme (et la distribution uniforme) :

$$P_{interpolation}(w_i|w_{i-2}w_{i-1}) = \lambda_1 P(w_i|w_{i-2}w_{i-1}) + \lambda_2 P(w_i|w_{i-1}) + \lambda_3 P(w_i) + \lambda_4 \frac{1}{|\mathcal{V}|}$$

où $\sum_{i=1}^{4} \lambda_i = 1$. Le dernier terme correspond à l'assignation d'une masse de probabilité uniforme sur tous les mots. L'estimation des probabilités se fait sur l'ensemble d'entraînement et la détermination des paramètres λ_i se fait sur l'ensemble de validation.

Lissage de Katz

Le lissage de Katz [11] se base sur la formule de Good-Turing [8]. L'idée maîtresse est la suivante : si une séquence précise de mots survient une seule fois dans l'ensemble d'entraînement (une fois parmi plusieurs millions de mots), elle est fort probablement surestimée puisque d'autres séquences de mêmes probabilités auraient pu apparaître à sa place. La même observation est aussi valide, à un moindre degré, pour les séquences apparaissant deux fois, et ainsi de suite. Il s'agit donc de corriger à la baisse les estimés des décomptes de séquences de mots.

Soit n_r la quantité représentant le nombre de n-grammes qui se retrouvent r fois dans l'ensemble d'entraînement. Good a démontré que l'on peut corriger la quantité n_r par le facteur suivant :

$$r^*(r) = (r+1)\frac{n_{r+1}}{n_r}$$

En utilisant n_r à la place de la fonction $C(\cdot)$ dans le numérateur de l'équation 2.1, on obtient une nouvelle approximation des probabilités conditionnelles. Sommant sur toutes les probabilités conditionnelles, on s'aperçoit qu'il reste un résidu de masse de probabilité. Ce résidu est ensuite distribué uniformément sur les événements rares qui ne surviennent pas dans l'ensemble d'entraînement.

Le lissage de Katz est une fonction de sélection binaire. Dans le cas où le n-gramme a été rencontré dans l'ensemble d'entraînement, on utilise la probabilité corrigée présentée précédemment. Dans l'autre cas, on utilise l'information du $(n-1)$-gramme normalisé par une constante :

$$P_{Katz}(w_i|w_{i-2}w_{i-1}) = \begin{cases} \frac{r^*(C(w_{i-2}w_{i-1}w_i))}{C(w_{i-2}w_{i-1})} & \text{si } C(w_{i-2}w_{i-1}w_i) > 0 \\ \alpha(w_{i-2}w_{i-1}) \times P_{Katz}(w_i|w_{i-1}) & \text{sinon} \end{cases}$$

où $\alpha(\cdot)$ est un facteur de normalisation permettant aux probabilités de sommer à un.

2.2 Réseaux de neurones

Les méthodes à base de trigramme sont des modèles simples, relativement performants et peu coûteux. Cependant, leur capacité à modéliser des dépendences complexes est limitée. Les réseaux de neurones, par contre, sont capables de modéliser des fonctions complexes et bruitées.

Les réseaux de neurones sont un des algorithmes disponibles parmi l'éventail des techniques offertes par l'apprentissage machine. C'est une technique relativement simple et qui fonctionne bien pour une grande diversité de problèmes. Les réseaux de neurones représentent l'approche *connexioniste* de l'intelligence artificielle. Un réseau de neurones est plus facilement visualisable par sa topologie, c'est-à-dire la façon de connecter les composantes ensembles, présentée à la figure 2.1.

Cette section présente tout d'abord une introduction au domaine de l'apprentissage machine nécessaire pour la compréhension des concepts qui sont ultérieurement utilisés. L'apprentissage machine est le champ d'étude des algorithmes d'apprentissage en général. Les réseaux de neurones sont *un* des algorithmes d'apprentissage possibles parmi une vaste panoplie. La section 2.2.2 donne une description générale du fonctionnement des réseaux de neurones. L'application des réseaux de neurones à la modélisation du langage comme tel est donnée à la section 2.3.

2.2.1 Apprentissage machine

On se limite dans cette section à présenter, outre le paradigme de l'apprentissage machine nécessaire à la mise en oeuvre des réseaux de neurones, leur topologie et leur règle d'entraînement. Les notions présentées seront réutilisées lors de la présentation des algorithmes de modélisation du langage par réseaux de neurones.

Ensemble de données

Lorsqu'on parle d'apprentissage machine, on réfère au processus de construction d'un algorithme de décision ou d'approximation à partir de données. Ces données peuvent être étiquetées ou non. On parle alors d'apprentissage supervisé ou non supervisé, respectivement. Dans le cas de l'apprentissage non supervisé, on dispose d'une série d'exemples correspondant à des entrées d'une fonction :

$$X = \{\mathbf{d}^1, \mathbf{d}^2, \ldots, \mathbf{d}^n\} \tag{2.2}$$

Dans le cas de l'apprentissage supervisé, les données disponibles sont des instances d'entrées d'une fonction ainsi qu'une sortie (ou réponse) désirée (étiquetée). On dispose alors d'une série d'exemples :

$$X = \{(\mathbf{d}^1, \mathbf{t}^1), (\mathbf{d}^2, \mathbf{t}^2), \ldots, (\mathbf{d}^n, \mathbf{t}^n)\} \tag{2.3}$$

où les $(\mathbf{d}^i, \mathbf{t}^i)$ sont les couples (*entrée, réponse désirée*).

On verra plus loin que la modélisation du langage est un problème d'apprentissage non supervisé mais pour lequel on peut utiliser des algorithmes supervisés (comme les réseaux

de neurones) en considérant les données de l'équation 2.2 comme des données du type de l'équation 2.3.

Règle d'apprentissage

L'objectif est de construire une fonction F prenant en entrée des vecteurs du même type que les $\mathbf{d}^i$ et de produire des sorties qui apportent de l'information ($\mathbf{p}$) sur la valeur possible des $\mathbf{t}^i$. La fonction F est une fonction paramétrisée :

$$\mathbf{p} = F(\mathbf{d}, \omega)$$

où ω réfère aux *paramètres*[1]. On dit que $\mathbf{p}$ est la prédiction sur l'entrée $\mathbf{d}$.

La règle d'apprentissage est l'algorithme qui permettra de modifier la valeur des paramètres ω de façon à améliorer les prédictions de F. En réalité, on cherche à construire l'approximation d'une fonction généralement inconnue sauf pour l'ensemble de données 2.2 ou 2.3. On appelle le processus de modification des paramètres, le processus d'entraînement. Le processus d'entraînement est l'application d'un algorithme de correction des poids ω et est lui-même régi par les hyperparamètres $\mathcal{H}$[2].

Fonction d'erreur

Étant donné la fonction F ainsi qu'une instance de ses paramètres ω, on choisit une fonction E déterminant une pénalité (ou erreur) sur la réponse fournie par la fonction F sur une instance $\mathbf{d}^i$. L'objectif sera alors de minimiser la somme des erreurs de la fonction F sur l'ensemble complet des données. Une technique simple consiste à minimiser E graduellement en corrigeant les paramètres ω selon le gradient de la fonction d'erreur[3] par rapport à ces paramètres. Cette technique est habituellement appellées *descente stochastique de gradient*. Il faut noter que la relation entre $\mathbf{d}$ et $\mathbf{t}$ est généralement non-déterministe, c'est-à-dire que $\mathbf{d}^i = \mathbf{d}^j \not\Rightarrow \mathbf{t}^i = \mathbf{t}^j$.

2.2.2 Fonctionnement des réseaux de neurones

Un réseau de neurones s'identifie naturellement à la fonction paramétrisée $F(\mathbf{d}, \omega)$ présentée précédemment. Il s'agit de trouver un algorithme permettant de modifier ses paramètres ω afin d'obtenir une bonne approximation de la fonction désirée. En théorie, les réseaux de neurones possèdent des propriétés intéressantes. Notamment, en vertu de ce qu'on appelle généralement le théorème d'approximation universelle, la propriété de pouvoir approximer

[1] Aussi appelés *poids*.

[2] On distingue les paramètres des hyperparamètres comme suit : les paramètres sont généralement choisis automatiquement par un algorithme alors que les hyperparamètres sont typiquement choisi préalablement de façon (plus ou moins) intuitive. Par exemple, le nombre d'unités cachées est un hyperparamètre, tout comme le nombre d'entrées et de sorties.

[3] E est, comme F, une fonction dépendente des paramètres ω et ainsi dérivable par rapport à ceux-ci.

toutes fonctions continues aussi précisément que désiré. En pratique, il n'existe pas d'algorithmes permettant de modifier les paramètres pour obtenir n'importe quelle fonction. Il existe cependant un algorithme simple et relativement efficace qui, même s'il ne permet pas de remplir les promesses du théorème d'approximation, fonctionne généralement bien. C'est l'algorithme de descente stochastique de gradient, mentionné plus tôt, où le gradient est obtenu par rétropropagation de l'erreur. On présente cet algorithme dans les prochaines sections. En résumé, on procède successivement sur chacun des exemples $(\mathbf{d}^i, \mathbf{t}^i)$ de l'ensemble d'entraînement à une étape de propagation avant et à une étape de propagation arrière. Ces étapes successives permettent de calculer le gradient de la fonction d'erreur E dans l'espace des paramètres ω.

Propagation avant

L'étape de propagation avant permet de calculer la prédiction $\mathbf{p}$ du réseau de neurones sur un exemple $\mathbf{d}$ présenté en entrée et également de calculer le coût E associé. L'entrée $\mathbf{d}$ présentée au réseau de neurones procède de gauche à droite, de l'entrée à la sortie. Au cours du passage à travers le réseau, les données sont pondérées une première fois par les poids de la première couche de paramètres :

$$\mathbf{s}_j = \mathbf{b}_{\mathbf{i};j} + \sum_{k=1}^{n_{inputs}} \mathbf{W}_{\mathbf{i};j,k}\mathbf{d}_k \qquad j = 1, 2, \ldots, n_{hidden}$$

où $\mathbf{b}_\mathbf{i}$ sont les biais de la première couche, de dimension n_{hidden}, et $\mathbf{W}_\mathbf{i}$ est la matrice de la première couche de poids, de dimensions $n_{inputs} * n_{hidden}$. n_{inputs} réfère au nombre d'entrées du réseau, tandis que n_{hidden} dénote le nombre d'unités cachées. Habituellement, le choix de n_{inputs} est déterminé par la nature du problème alors que le nombre d'unités cachées n_{hidden} peut-être choisi. On peut réécrire l'équation précédente de façon plus compacte en utilisant la notation matricielle :

$$\mathbf{s} = \mathbf{d}^T\mathbf{W}_\mathbf{i} + \mathbf{b}_\mathbf{i}$$

On applique ensuite une fonction non linéaire, on utilise habituellement la fonction $tanh(\cdot)$ qui possède des caractéristiques intéressantes (voir [5], [6] et [9]). Il en résulte ce qu'on appelle le vecteur d'*activation* $\mathbf{a}$:

$$\mathbf{a}_j = tanh(\mathbf{s}_j) \qquad j = 1, 2, \ldots, n_{hidden} \tag{2.4}$$

ou sous la forme matricielle :

$$\mathbf{a} = tanh(\mathbf{s})$$

Le résultat est ensuite pondéré par la seconde couche de paramètres pour obtenir la sortie brute :

$$\mathbf{o}_k = \mathbf{b}_{\mathbf{h};k} + \sum_{l=1}^{n_{hidden}} \mathbf{W}_{\mathbf{h};k,l}\mathbf{a}_l \qquad k = 1, 2, \ldots, n_{outputs} \tag{2.5}$$

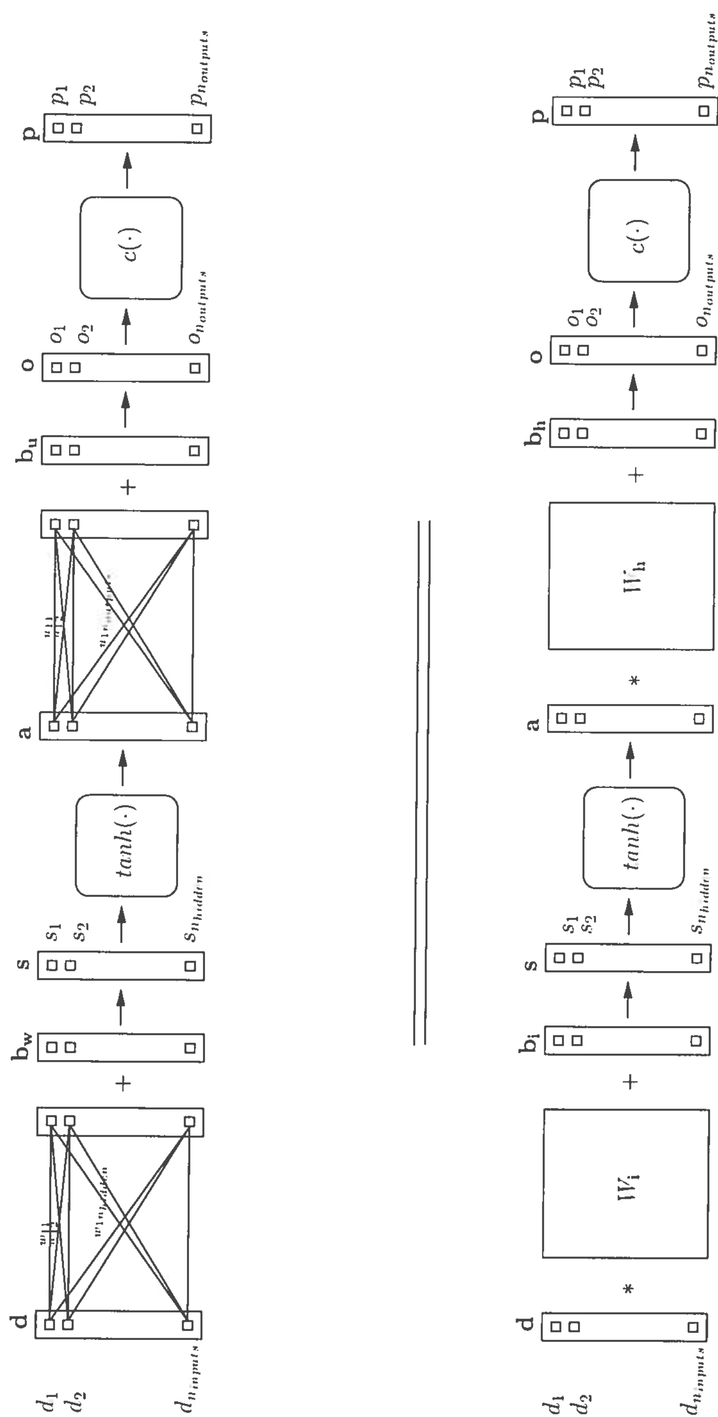

FIG. 2.1 – Topologie d'un réseau de neurones

où $\mathbf{b_h}$ sont les biais de le seconde couche, de dimension $n_{outputs}$, et $\mathbf{W_h}$ est la matrice de la seconde couche, de dimensions $n_{hidden} * n_{outputs}$. $n_{outputs}$ est le nombre de sorties du réseau. Comme pour n_{inputs}, le nombre de sorties $n_{outputs}$ est déterminé par la nature du problème. La forme matricielle correspondante est la suivante :

$$\mathbf{o} = \mathbf{a}^T \mathbf{W_h} + \mathbf{b_h}$$

Ensuite, la sortie brute $\mathbf{o}$ est soumise à la fonction de transfert $c(\cdot)$ choisie pour le problème que l'on désire considérer. Il est possible d'omettre cette fonction de transfert. Cependant, il est parfois utile de transformer la sortie brute. Par exemple, si l'on désire modéliser une distribution de probabilité, il est essentiel que la somme des sorties soit 1. Pour ce faire, on peut utiliser la fonction $softmax(\cdot)$:

$$\begin{aligned} \mathbf{p} &= softmax(\mathbf{o}) \\ \mathbf{p}_k &= \frac{e^{\mathbf{o}_k}}{\sum_{l=1}^{n_{outputs}} e^{\mathbf{o}_l}} \qquad i = 1, \ldots, n_{outputs} \end{aligned} \tag{2.6}$$

De cette façon, on a bien que $\sum_j \mathbf{p}_j = 1$. Cela permet en outre d'interpréter les sorties du réseau comme étant les probabilités *a posteriori* de différentes catégories (représentant, par exemple, les mots d'un vocabulaire) étant donnée une entrée.

On écrit les équations précédentes de façon plus compacte ainsi :

$$\mathbf{p} = softmax\big((tanh(\mathbf{d}^T \mathbf{W_i} + \mathbf{b_i}))^T \mathbf{W_h} + \mathbf{b_h}\big) \tag{2.7}$$

La figure 2.1 illustre la topologie du réseau et l'organisation des différentes composantes discutées précédemment.

Enfin, la fonction d'erreur est la fonction qui permet de caractériser les performances du réseaux de neurones et d'attribuer des modifications aux paramètres. La fonction d'erreur la plus simple et la plus souvent employée est la fonction de perte quadratique :

$$E_{quadratique}(\mathbf{d}^i, \omega) = \|\mathbf{t}^i - F(\mathbf{d}^i, \omega)\|^2$$

Le rôle de la fonction d'erreur est de contraindre le réseau à produire les sorties désirées. Ce rôle est rempli en pénalisant de façon appropriée l'écart entre la sortie du réseau et la sortie désirée.

La fonction d'erreur que l'on utilise pour cette recherche est la fonction E_{NLL}[4] qui s'écrit :

$$E_{NLL}(\mathbf{d}^i, \omega) = -ln(\mathbf{p}_{cible})$$

où *cible* est l'indice de la sortie désirée. Cette fonction d'erreur est intimement reliée à la perplexité présentée au chapitre précédent. Ainsi la minimisation de la fonction d'erreur E_{NLL} permet, de façon indirecte, de minimiser la perplexité. En effet, si on prend la moyenne

[4] NLL = Negative Log-likelihood

de l'erreur E_{NLL} sur l'ensemble de données X, cette moyenne est alors reliée à la perplexité de l'ensemble de données X de la façon suivante :

$$\frac{1}{n}\sum_{i=0}^{n} -ln(\mathbf{p}^{i}_{cible}) = ln(PERP(X))$$

Or, la fonction $ln(\cdot)$ est une fonction monotone croissante et cette moyenne trouve donc son minimum au même point que la perplexité.

Règle d'apprentissage

La règle d'apprentissage du réseau de neurones est relativement simple et intuitive. Elle procède itérativement mais individuellement sur chacun des exemples de l'ensemble d'entraînement. L'idée est qu'il faut corriger chacun des paramètres ω_i dans la direction inverse du gradient de l'erreur (calculée pour un exemple donné) :

$$\Delta\omega_i = -\eta\left(\frac{\partial E}{\partial \omega_i}\right) \tag{2.8}$$

La mise à jour effective des paramètres s'effectue par l'application numérique de la règle de dérivation en chaîne sur la fonction d'erreur E. La constante η est un hyperparamètre (appellée le taux d'apprentissage, *learning rate* en anglais) permettant de réguler la "vitesse" d'apprentissage. Elle détermine l'amplitude du "pas" à parcourir dans la direction inverse du gradient. En pratique, la constante η peut être modifiée en fonction du nombre d'exemples présentés[5]. À chaque itération[6], tous les paramètres sont mis à jour avec la valeur des paramètres de l'itération précédente :

$$\Delta\omega_i(n) = -\eta(n)\left(\frac{\partial}{\partial \omega_i}E(\mathbf{d}^n, \omega(n-1), \mathcal{H})\right)$$

Les paramètres avant la première itération sont initialisés aléatoirement.

Propagation arrière

En pratique, il est possible d'effectuer le calcul des corrections en procédant de la sortie vers l'entrée. Il n'est donc pas nécessaire de calculer la dérivée partielle $\partial E/\partial \omega_i$ (pour chaque paramètre i) à partir de l'expression explicite de $\partial E/\partial \omega_i$, mais certains calculs peuvent être réutilisés en utilisant les propriétés de la dérivation en chaîne. C'est à cause de cette méthode de calcul, qui procède des sorties aux entrées, qu'on appelle cette règle *rétropropagation de l'erreur*. Pour illustré le processus de propagation arrière, voici la séquence des calculs effectués. On calcule tout d'abord le gradient de l'erreur E par rapport à la sortie $\mathbf{o}$: $\partial E/\partial \mathbf{o}_j$

[5] Ou nombre d'itérations.

[6] Une itération correspond à un exemple.

pour chaque composante j de la sortie. On calcul ensuite le gradient sur les poids et les biais de la seconde couche :

$$\begin{aligned}\left(\frac{\partial E}{\partial \mathbf{W}_{\mathbf{h};k,l}}\right) &= \left(\frac{\partial E}{\partial \mathbf{o}_k}\right)\left(\frac{\partial \mathbf{o}_k}{\partial \mathbf{W}_{\mathbf{h};k,l}}\right) \\ \left(\frac{\partial E}{\partial \mathbf{b}_{\mathbf{h};k}}\right) &= \left(\frac{\partial E}{\partial \mathbf{o}_k}\right)\left(\frac{\partial \mathbf{o}_k}{\partial \mathbf{b}_{\mathbf{h};k}}\right)\end{aligned} \tag{2.9}$$

Avant d'appliquer l'équation 2.8 aux poids et biais de la seconde couche, on doit auparavant calculer la gradient de l'erreur sur l'activation $\mathbf{a}$:

$$\left(\frac{\partial E}{\partial \mathbf{a}_j}\right) = \sum_{k=1}^{n_{outputs}} \left(\frac{\partial E}{\partial \mathbf{o}_k}\right)\left(\frac{\partial \mathbf{o}_k}{\partial \mathbf{a}_j}\right)$$

On peut maintenant appliquer l'équation 2.8 modifiant les valeurs des poids et biais de la seconde couche. Il est important de noter que les calculs précédents réutilisent tous le gradient $\partial E/\partial \mathbf{o}$ initialement calculé.

On propage ensuite le gradient au travers de la fonction d'activation $tanh(\cdot)$:

$$\left(\frac{\partial E}{\partial \mathbf{s}_j}\right) = \left(\frac{\partial E}{\partial \mathbf{a}_j}\right)(1 - \mathbf{a}_j^2) \tag{2.10}$$

où le terme $(1 - \mathbf{a}_j^2)$ est l'expression de la dérivée de $tanh(\cdot)$ pour l'élément j. On peut maintenant calculer les gradients sur les poids et biais de la première couche :

$$\begin{aligned}\left(\frac{\partial E}{\partial \mathbf{W}_{\mathbf{i};j,k}}\right) &= \left(\frac{\partial E}{\partial \mathbf{s}_j}\right)\left(\frac{\partial \mathbf{s}_j}{\partial \mathbf{W}_{\mathbf{i};j,k}}\right) \\ \left(\frac{\partial E}{\partial \mathbf{b}_{\mathbf{i};j}}\right) &= \left(\frac{\partial E}{\partial \mathbf{s}_j}\right)\left(\frac{\partial \mathbf{s}_j}{\partial \mathbf{b}_{\mathbf{i};j}}\right)\end{aligned} \tag{2.11}$$

Une fois les gradients calculés, il ne reste plus qu'à appliquer l'équation 2.8 de nouveau pour modifier les valeurs des poids et biais de la première couche. Il faut noter que pour les deux derniers calculs, seul le gradient $\partial E/\partial \mathbf{s}$ a été utilisé et qu'on ne fait plus référence aux poids et biais de la seconde couche.

Entraînement

On appelle l'application de la règle d'apprentissage sur les exemples de l'ensemble d'entraînement, une *époque*. L'entraînement se poursuit habituellement pendant un certain nombre d'itérations sur l'ensemble d'entraînement (appellées époques). Il est possible (et même souhaitable) de modifier graduellement à la baisse la valeur de la constante η durant l'entraînement. L'heuristique choisie dépend du problème mais l'intuition est la suivante. Si η est trop grande alors il devient impossible de descendre à l'intérieur d'un minimum local et d'y rester. À l'inverse, si η est trop petite alors la descente est trop lente. Alors il est judicieux de débuter l'entraînement avec une valeur de η suffisamment grande pour aller rapidement et de décroître cette valeur régulièrement de façon à pouvoir ultimement atteindre un bon minimum local.

2.3 Modélisation par réseaux de neurones

Les sections suivantes présentent successivement les étapes menant au modèle de langage par réseau de neurones présenté dans [2]. On définit tout d'abord les ensembles de données de façon à exprimer le problème de modélisation en terme d'apprentissage. On décrit ensuite l'espace des caractéristiques qui permet l'entraînement d'un modèle de langage à base de réseau de neurones. On présente deux implémentations existantes de tels modèles : *NPLM* et *NPLM par échantillonnage*. On procèdera également à la description de la version parallèle de l'algorithme de *NPLM* (aussi dans [2]). Il est important de souligner que la plupart des détails d'implémentation de l'algorithme de *NPLM* et de sa version parallèle sont nécessaire à la compréhension de l'algorithme présenté au chapitre 3.

La section précédente a fait la présentation des notions qui sont nécessaires pour construire des réseaux de neurones. Ce chapitre donne les détails relatifs à leur utilisation dans le contexte de la modélisation du langage, c'est-à-dire lorsqu'on désire modéliser la probabilité suivante :

$$P(w_1, w_2, \ldots, w_L) = P(w_1)P(w_2|w_1)\ldots P(w_l|w_1, \ldots, w_{L-1})$$

où L est le dimension de l'ensemble des mots disponibles. On procède initialement de façon similaire au trigramme en utilisant une approximation définie par l'hypothèse simplificatrice suivante :

$$P(w_i|w_1 \ldots w_i - 1) \approx P(w_i|w_{i-l+1}, w_{i-l+2}, \ldots, w_{i-1}) \tag{2.12}$$

où l est la dimension maximale de la fenêtre utilisée pour calculer l'approximation.

2.3.1 Ensembles de données

La modélisation du langage, telle que présentée dans l'introduction, est un problème d'apprentissage non supervisé. On dispose ainsi d'une séquence de L mots telle que définie à la section 1.1 :

$$X = \{w_1, w_2, \ldots, w_L\}, w_i \in \mathcal{V}$$

Pour simplifier, on suppose ici que X est l'ensemble d'entraînement.

Pour obtenir, formellement, la série des exemples d'entraînement, il suffit d'imaginer une fenêtre d'une certaine longueur l se déplaçant dans la séquence par incréments unitaires. On obtient alors, à l'incrément i, les mots :

$$\{w_{i-l+1}, \ldots, w_{i-1}, w_i\}$$

On défini les vecteurs *entrées* $\mathbf{d}^i$ ainsi :

$$\mathbf{d}^i = \{w_{i-l+1}, \ldots, w_{i-2}, w_{i-1}\}$$

et les vecteurs *réponses désirées* $\mathbf{t}^i$ de la façon suivante :

$$\mathbf{t}^i_k = \begin{cases} 1 & \text{si } k = w_i \\ 0 & \text{sinon.} \end{cases}$$

Les vecteurs $\mathbf{t}^i$ sont des vecteurs de dimension $|\mathcal{V}|$, c'est-à-dire le nombre de mots que l'on considère. La dernière expression signifie simplement que la réponse désirée est un vecteur dont les composantes sont nulles à l'exception d'une seule. La position k de la composante non-nulle est l'entier représentant le mot k du vocabulaire *ainsi que* le mot identifié par w_i.

On obtient donc une série de n exemples (avec $n = L - l + 1$).

$$\{(\mathbf{d}^1, \mathbf{t}^1), (\mathbf{d}^2, \mathbf{t}^2), \ldots, (\mathbf{d}^n, \mathbf{t}^n)\}$$

Cette série permet donc d'utiliser un algorithme d'apprentissage supervisé pour procéder à la modélisation du langage en utilisant une séquence de mot pour définir des exemples de types (*entrée, réponse désirée*).

2.3.2 Espace des caractéristiques

Au lieu de présenter directement des (identificateurs de) mots en entrée au réseau de neurones, on associera plutôt à chaque mot un vecteur de caractéristiques. Cela fait en sorte de pouvoir associer le code discret d'un mot w_j à des vecteurs de réels. Soit $\mathbf{C}^{w_j}$ le vecteur correspondant au mot w_j, chaque vecteur étant de dimension m (typiquement $m = 30$). Soit $C(\cdot)$ la fonction qui prend la donnée d'entrée $\mathbf{d}^i$, en extrait les mots du contexte (dénoté h_t) et retourne la concaténation des vecteurs caractéristiques correspondants :

$$\begin{aligned} C(\mathbf{d}^i) = C(h_t) = \mathbf{C}_t &= C(w_{i-l+1}, w_{i-l+2}, \ldots, w_i) \\ &= \mathbf{C}^{w_{i-l+1}} \oplus \mathbf{C}^{w_{i-l+2}} \oplus \cdots \oplus \mathbf{C}^{w_i} \end{aligned} \tag{2.13}$$

où l est la dimension de la fenêtre se déplaçant et définissant le contexte utilisé pour déterminer la probabilité du mot w_i[7].

L'utilisation de vecteurs caractéristique est l'élément clé qui permet ici la modélisation du langage en utilisant des réseaux de neurones. On procède au passage des mots de l'entrée à leurs vecteurs dans l'espace caractéristique pour plusieurs raisons.

Premièrement, la représentation des mots dans l'espace caractéristique permet d'utiliser l'idée générale des ensembles de synonymes présentée dans l'introduction. Il ne s'agit pas exactement d'utiliser les synonymes, mais plutôt d'identifier des patrons d'usages de certain mot. On parle alors du concept de *substituabilité* où, par exemple, deux mots peuvent être substitués l'un pour l'autre sans changer le sens d'une phrase. En pratique la substituabilité de deux mots n'est pas identifiée explicitement mais implicitement durant l'apprentissage. En effet, on s'attend intuitivement à ce que deux mots *substituables* comportant une fonction linguistique similaire aient des vecteurs caractéristiques prochent. Ainsi, deux mots substituables génèreront des entrées similaires pour le réseau de neurones et produiront par le fait même des sorties proches. Cela permet de se détacher du concept de mot pour plutôt appliquer le processus de décision sur la fonction du mot. Donc, l'apprentissage se faisant non sur des mots mais sur des ensembles plus abstraits permet, comme pour l'idée générale

[7]Les exemples de l'ensemble d'entraînement sont définis par une fenêtre de dimension fixe se déplacant dans le corpus et choisissant les mots à présenter en entrée ainsi que le mot désiré w_i (pour l'exemple i).

des ensembles de synonymes, de résoudre en partie le problème de la sous-couverture de l'ensemble d'entraînement.

Deuxièmement, les vecteurs caractéristiques $\mathbf{C}^{w_j}$ seront "appris" aux cours du processus d'apprentissage. C'est-à-dire qu'on considère les composantes de ces vecteurs comme faisant partie des paramètres du réseau de neurones. Ainsi, lors de la rétropropagation de l'erreur, on appliquera une correction aux vecteurs ayant été utilisés pour générer l'entrée. C'est parce que les vecteurs caractéristiques sont appris qu'il n'est pas nécessaire d'identifier explicitement les fonctions linguistiques des mots puisqu'elles seront "découvertes" automatiquement lors de l'apprentissage. Les vecteurs caractéristiques sont donc un élément important du modèle de langage qui est appris.

Dernièrement, il est plus facile d'apprendre une fonction "lisse" si les valeurs d'entrée sont continues. Ceci permet ainsi de faciliter le processus de descente de gradient.

Tous les modèles basés sur les réseaux de neurones présentés dans ce document utiliseront cette représentation dans l'espace caractéristique.

Pour terminer, on écrit maintenant le calcul du vecteur d'activation (voir eq. 2.4) ainsi :

$$\begin{aligned} \mathbf{a}_j &= tanh\left(\mathbf{b}_{\mathbf{i};j} + \sum_{k=1}^{n_{inputs}} \mathbf{W}_{\mathbf{i};j,k}\mathbf{e}_k \quad j = 1, 2, \ldots, n_{hidden}\right) \\ \mathbf{a} &= tanh\left(\mathbf{e}^T\mathbf{W}_\mathbf{i} + \mathbf{b}_\mathbf{i}\right) \end{aligned} \tag{2.14}$$

où $\mathbf{e} = C(\mathbf{d}^i)$[8] et $\mathbf{W}_\mathbf{i}$ est de dimensions $n_{inputs} * n_{hidden}$. L'équation 2.14 sert à mettre en évidence le fait que contrairement à l'utilisation usuelle d'un réseau de neurone ou un vecteur $\mathbf{d}$ est utilisé en entrée, on utilise maintenant un vecteur $\mathbf{e}$. Ce vecteur $\mathbf{e}$ est construit selon l'entrée du réseau et désigne, selon l'entrée, un sous-ensemble de tout les vecteurs caractéristiques. Ainsi, comme $\mathbf{W}_\mathbf{i}$, les vecteurs caractéristiques utilisés pour la construction de $\mathbf{e}$ font partie des paramètres qui seront modifiés durant le processus d'apprentissage.

2.3.3 NPLM

Le NPLM[9] [2] est un réseau de neurones capable de modéliser la probabilité de l'équation 2.12. Les deux étapes cruciales menant au NPLM ont été présentées dans les sections précédentes. Ainsi, la construction de la section 2.3.1 menant à un ensemble de données est essentielle à l'application ultérieure de l'algorithme de rétropagation de l'erreur. De plus, l'utilisation de vecteurs caractéristiques est nécessaire pour l'utilisation de réseaux de neurones.

Le modèle de langage NPLM utilise les deux éléments précédents ainsi que les fonctions de transfert $softmax(\cdot)$ et d'erreur E_{NLL} définies à la section 2.2.2. Les détails du calcul pour l'étape de propagation avant sont également tels que présentés antérieurement, c'est-à-dire

[8]Noter que n_{inputs} est redéfini de façon à tenir compte de la *nouvelle* entrée $\mathbf{e}$.

[9]en anglais : *Neural Probabilistic Language Model.*

que les sorties sont calculées ainsi :

$$\mathbf{p} = softmax\big((tanh(\mathbf{e}^T\mathbf{W_i} + \mathbf{b_i}))^T\mathbf{W_h} + \mathbf{b_h}\big) \tag{2.15}$$

On présente donc seulement les calculs nécessaires pour l'étape de propagation arrière.

Pour l'étape de propagation arrière (*bprop*), la mise à jour des poids ($\mathbf{W_h}$, $\mathbf{b_h}$, $\mathbf{W_i}$, $\mathbf{b_i}$ et $\mathbf{e}$[10]), est effectuée en utilisant le gradient de l'erreur E_{NLL} sur les poids. On calcule les gradients de façon récursive en commencant par $\frac{\partial E_{NLL}}{\partial \mathbf{W}_{\mathbf{h};k,l}}$ et $\frac{\partial E_{NLL}}{\partial \mathbf{b}_{\mathbf{h};k}}$, suivi par $\frac{\partial E_{NLL}}{\partial \mathbf{W}_{\mathbf{i};i,j}}$ et $\frac{\partial E_{NLL}}{\partial \mathbf{b}_{\mathbf{i};j}}$ et terminant par $\frac{\partial E_{NLL}}{\partial \mathbf{e}_i}$. Au cours du calcul on garde en mémoire certains calculs intermédiaires de façon à éviter de refaire plusieurs calculs. On présente tout d'abord les expressions du gradient de l'erreur par rapport aux différents paramètres. On abordera ensuite la question du calcul efficace de ces différentes quantités.

On commence par l'expression du gradient pour les sorties brutes :

$$\begin{aligned}
\left(\frac{\partial E_{NLL}}{\partial \mathbf{o}_k}\right) &= \left(\frac{\partial E_{NLL}}{\partial \mathbf{p}_{cible}}\right)\left(\frac{\partial \mathbf{p}_{cible}}{\partial \mathbf{o}_k}\right) \\
&= \left(\frac{-1}{\mathbf{p}_{cible}}\right)\left(\mathbf{p}_{cible}(\mathbb{I}_{\{k=cible\}} - \mathbf{p}_k\right) \\
&= -\left(\mathbb{I}_{\{k=cible\}} - \mathbf{p}_k\right)
\end{aligned}$$

On poursuit pour les poids de la couche cachée par l'application de l'équation 2.9 :

$$\begin{aligned}
\left(\frac{\partial E_{NLL}}{\partial \mathbf{W}_{\mathbf{h};k,l}}\right) &= \left(\frac{\partial E_{NLL}}{\partial \mathbf{p}_{cible}}\right)\left(\frac{\partial \mathbf{p}_{cible}}{\partial \mathbf{o}_k}\right)\left(\frac{\partial \mathbf{o}_k}{\partial \mathbf{W}_{\mathbf{h};k,l}}\right) \\
&= \left(\frac{-1}{\mathbf{p}_{cible}}\right)\Big(\mathbf{p}_{cible}(\mathbb{I}_{\{k=cible\}} - \mathbf{p}_k)\Big)(\mathbf{a}_l) \\
&= -\mathbf{a}_l(\mathbb{I}_{\{k=cible\}} - \mathbf{p}_k) \\
\left(\frac{\partial E_{NLL}}{\partial \mathbf{b}_{\mathbf{h};k}}\right) &= \left(\frac{\partial E_{NLL}}{\partial \mathbf{p}_{cible}}\right)\left(\frac{\partial \mathbf{p}_{cible}}{\partial \mathbf{o}_k}\right)\left(\frac{\partial \mathbf{o}_k}{\partial \mathbf{b}_{\mathbf{h};k}}\right) \\
&= \left(\frac{-1}{\mathbf{p}_{cible}}\right)\Big(\mathbf{p}_{cible}(\mathbb{I}_{\{k=cible\}} - \mathbf{p}_k)\Big) \\
&= -(\mathbb{I}_{\{k=cible\}} - \mathbf{p}_k)
\end{aligned} \tag{2.16}$$

Pour les poids de la couche d'entrée, les équations explicites dérivées de l'équation 2.9

[10]Le vecteur $\mathbf{e}$ est, techniquement, une référence aux vecteurs $\mathbf{C}^{w_k}$ de l'équation 2.13. On dénotera néanmoins les poids des vecteurs $\mathbf{C}^{w_k}$ directement par $\mathbf{e}$.

sont les suivantes :

$$
\begin{aligned}
\left(\frac{\partial E_{NLL}}{\partial \mathbf{W}_{\mathbf{i};j,k}}\right) &= \left(\frac{\partial E_{NLL}}{\partial \mathbf{p}_{cible}}\right) \sum_{l=1}^{n_{outputs}} \left(\frac{\partial \mathbf{p}_{cible}}{\partial \mathbf{o}_l}\right)\left(\frac{\partial \mathbf{o}_l}{\partial \mathbf{a}_j}\right)\left(\frac{\partial \mathbf{a}_j}{\partial \mathbf{s}_j}\right)\left(\frac{\partial \mathbf{s}_j}{\partial \mathbf{W}_{\mathbf{i};j,k}}\right) \\
&= \left(\frac{-1}{\mathbf{p}_{cible}}\right) \sum_{l=1}^{n_{outputs}} \left(\mathbf{p}_{cible}(\mathbb{I}_{\{l=cible\}} - \mathbf{p}_l)\right)\left(\mathbf{W}_{\mathbf{h};l,j}\right)\left(1-\mathbf{a}_j^2\right)\left(\mathbf{e}_k\right) \\
&= -(1-\mathbf{a}_j^2)\mathbf{e}_k \sum_{l=1}^{n_{outputs}} (\mathbb{I}_{\{l=cible\}} - \mathbf{p}_l)\mathbf{W}_{\mathbf{h};l,j} \\
\left(\frac{\partial E_{NLL}}{\partial \mathbf{b}_{\mathbf{i};j}}\right) &= \left(\frac{\partial E_{NLL}}{\partial \mathbf{p}_{cible}}\right) \sum_{l=1}^{n_{outputs}} \left(\frac{\partial \mathbf{p}_{cible}}{\partial \mathbf{o}_l}\right)\left(\frac{\partial \mathbf{o}_l}{\partial \mathbf{a}_j}\right)\left(\frac{\partial \mathbf{a}_j}{\partial \mathbf{s}_j}\right)\left(\frac{\partial \mathbf{s}_j}{\partial \mathbf{b}_{\mathbf{i};j}}\right) \\
&= \left(\frac{-1}{\mathbf{p}_{cible}}\right) \sum_{l=1}^{n_{outputs}} \left(\mathbf{p}_{cible}(\mathbb{I}_{\{l=cible\}} - \mathbf{p}_l)\right)\left(\mathbf{W}_{\mathbf{h};l,j}\right)\left(1-\mathbf{a}_j^2\right) \\
&= -(1-\mathbf{a}_j^2) \sum_{l=1}^{n_{outputs}} (\mathbb{I}_{\{l=cible\}} - \mathbf{p}_l)\mathbf{W}_{\mathbf{h};l,j}
\end{aligned} \tag{2.17}
$$

Les équations explicites pour les poids des vecteurs caractéristiques sont :

$$
\begin{aligned}
\left(\frac{\partial E_{NLL}}{\partial \mathbf{e}_i}\right) &= \left(\frac{\partial E_{NLL}}{\partial \mathbf{p}_{cible}}\right) \sum_{l=1}^{n_{outputs}} \left\{\left(\frac{\partial \mathbf{p}_{cible}}{\partial \mathbf{o}_l}\right) \sum_{k=1}^{n_{hidden}} \left(\frac{\partial \mathbf{o}_l}{\partial \mathbf{a}_k}\right)\left(\frac{\partial \mathbf{a}_k}{\partial \mathbf{s}_k}\right)\left(\frac{\partial \mathbf{s}_k}{\partial \mathbf{e}_i}\right)\right\} \\
&= \left(\frac{-1}{\mathbf{p}_{cible}}\right) \sum_{l=1}^{n_{outputs}} \left(\mathbf{p}_{cible}(\mathbb{I}_{\{l=cible\}} - \mathbf{p}_l) \sum_{k=1}^{n_{hidden}} \left(\mathbf{W}_{\mathbf{h};l,k}\right)\left(1-\mathbf{a}_k^2\right)\left(\mathbf{W}_{\mathbf{i};k,i}\right)\right) \\
&= -\sum_{l=1}^{n_{outputs}} \left((\mathbb{I}_{\{l=cible\}} - \mathbf{p}_l) \sum_{k=1}^{n_{hidden}} \mathbf{W}_{\mathbf{h};l,k}(1-\mathbf{a}_k^2)\mathbf{W}_{\mathbf{i};k,i}\right) \\
&= -\sum_{k=1}^{n_{hidden}} (1-\mathbf{a}_k^2)\mathbf{W}_{\mathbf{i};k,i} \sum_{l=1}^{n_{outputs}} (\mathbb{I}_{\{l=cible\}} - \mathbf{p}_l)\mathbf{W}_{\mathbf{h};l,k}
\end{aligned} \tag{2.18}
$$

On substitue finalement les expressions des dérivées partielles dans l'équation 2.8.

Calcul efficace

Tel que mentionné, il n'est pas efficace d'utiliser les équations explicites des dérivées partielles pour faire les calculs. Le choix judicieux de quantités intermédiaires permet d'éviter de recalculer les quantités et ainsi d'économiser beaucoup de temps.

On garde tout d'abord en mémoire le calcul suivant dans un vecteur $\mathbf{x}$ (de dimension

$n_{outputs}$) :

$$\begin{aligned}
\mathbf{x}_k &= \left(\frac{\partial E_{NLL}}{\partial \mathbf{p}_{cible}}\right)\left(\frac{\partial \mathbf{p}_{cible}}{\partial \mathbf{o}_k}\right) \\
&= \left(\frac{-1}{\mathbf{p}_{cible}}\right)\left(\mathbf{p}_{cible}(\mathbb{I}_{\{k=cible\}} - \mathbf{p}_k)\right) \\
&= (\mathbf{p}_k - \mathbb{I}_{\{k=cible\}})
\end{aligned}$$

Avant de modifier les poids $\mathbf{W_h}$ et $\mathbf{b_h}$, on calcul tout d'abord le gradient $\mathbf{y}$ (de dimension n_{hidden}) :

$$\begin{aligned}
\mathbf{y}_j &= \left(\frac{\partial E_{NLL}}{\partial \mathbf{p}_{cible}}\right)\sum_{k=1}^{n_{outputs}}\left(\frac{\partial \mathbf{p}_{cible}}{\partial \mathbf{o}_k}\right)\left(\frac{\partial \mathbf{o}_k}{\partial \mathbf{a}_j}\right)\left(\frac{\partial \mathbf{a}_j}{\partial \mathbf{s}_j}\right) \\
&= (1 - \mathbf{a}_j^2)\sum_{k=1}^{n_{outputs}} \mathbf{x}_k \mathbf{W}_{\mathbf{h};k,j}
\end{aligned}$$

Les équations 2.16 pour les poids $\mathbf{W_h}$ et $\mathbf{b_h}$ se calculent maintenant ainsi :

$$\begin{aligned}
\left(\frac{\partial E_{NLL}}{\partial \mathbf{W}_{\mathbf{h};k,l}}\right) &= \mathbf{x}_k \mathbf{a}_l \\
\left(\frac{\partial E_{NLL}}{\partial \mathbf{b}_{\mathbf{h};k}}\right) &= \mathbf{x}_k
\end{aligned}$$

Ces expressions s'expriment naturellement sous forme vectorielle :

$$\begin{aligned}
\left(\frac{\partial E_{NLL}}{\partial \mathbf{W_h}}\right) &= \mathbf{x}\ \mathbf{a}^T \\
\left(\frac{\partial E_{NLL}}{\partial \mathbf{b_h}}\right) &= \mathbf{x}
\end{aligned}$$

Avant de calculer les corrections pour les poids $\mathbf{W_i}$ et $\mathbf{b_i}$, on fait les calculs pour l'équation 2.18 :

$$\begin{aligned}
\left(\frac{\partial E_{NLL}}{\partial \mathbf{e}_i}\right) &= \sum_{j=1}^{n_{hidden}} \mathbf{y}_j \mathbf{W}_{\mathbf{i};j,i} \\
\left(\frac{\partial E_{NLL}}{\partial \mathbf{e}}\right) &= \mathbf{y}^T \mathbf{W_i}
\end{aligned}$$

Finalement, on calcule les équations 2.17 :

$$\begin{aligned}
\left(\frac{\partial E_{NLL}}{\partial \mathbf{W}_{\mathbf{i};j,k}}\right) &= \mathbf{y}_j \mathbf{e}_k \\
\left(\frac{\partial E_{NLL}}{\partial \mathbf{W_i}}\right) &= \mathbf{y}\ \mathbf{e}^T
\end{aligned}$$

et

$$\left(\frac{\partial E_{NLL}}{\partial \mathbf{b_{i;}}_j}\right) = \mathbf{y}_j$$
$$\left(\frac{\partial E_{NLL}}{\partial \mathbf{b_i}}\right) = \mathbf{y}$$

L'ordre des calculs présentés ci-haut permet de procéder des sorties du réseau vers les entrées de façon récursive.

2.3.4 Parallélisation du NPLM

La parallélisation d'un algorithme nécessite l'analyse des parties d'un calcul qui peuvent être exécutées indépendamment. Il existe peu de problèmes algorithmiques qui peuvent être décomposés en une série de sous-problèmes et calculés indépendemment. En général, les différents sous-problèmes ont des dépendances qui doivent être communiquées de façon à ce que chaque unité de calcul parallèle fasse le bon calcul. Il s'agit souvent de reproduire une partie des calculs sur chacun des processeurs afin de minimiser la quantité de données à communiquer. La quantité de calculs à reproduire et la quantité de communication à effectuer sont comparées par rapport à leur côut en temps et un compromis minimisant le temps total est trouvé.

Dans la formulation des réseaux de neurones présentée, le calcul des quantités au travers d'un réseau est essentiellement le calcul de produits matrice-vecteur. Il est possible de décomposer ces produits en blocs, tel qu'expliqué à l'annexe A. Ces produits décomposés peuvent ainsi être calculés sur différentes machines permettant, potentiellement, un gain sur la vitesse de calcul. Cependant, il est nécessaire de communiquer certaines quantités entre les différentes machines prenant part au calcul. C'est la communication de ces quantités qui est en générale problématique et qu'il s'agit d'amortir et/ou de minimiser.

Il existe deux possibilités simples pour paralléliser un réseau de neurones. Elles consistent à décomposer les produits vecteur-matrice selon soit le nombre d'unités cachées n_{hidden} (partition des matrices et vecteur $\mathbf{W_i}$ et $\mathbf{W_h}$ et $\mathbf{b_i}$) ou le nombre des sorties $n_{outputs}$[11] (partition des matrice et vecteur $\mathbf{W_h}$ et $\mathbf{b_h}$).

Exemple

Pour fin d'exemple, on considère le cas où on effectue la décomposition selon le nombre d'unités cachées. Soit N le nombre de machines disponibles et supposons pour simplifier que la quantité n_{hidden} est un multiple de N. Soit $\mathbf{V_i}^l$ et $\mathbf{V_h}^l$ les sous-matrices de $\mathbf{W_i}$ et $\mathbf{W_h}$ pour $0 \leq l < N$. Ces matrices auront pour dimensions respectives $n_{inputs} * (n_{hidden}/N)$ et $(n_{hidden}/N) * n_{outputs}$. De même, soit $\mathbf{q_i}^l$ les sous-vecteurs de $\mathbf{b_i}$, de dimensions (n_{hidden}/N).

[11] Par exemple, en décomposant selon n_{hidden}, on divise les matrices $\mathbf{W_i}$ et $\mathbf{W_h}$ selon leur dimension n_{hidden} et on calcule les produits partiels.

Les sous-matrices et sous-vecteurs s'organisent comme suit :

$$\mathbf{W_i} = [\mathbf{V_i}^1\ \mathbf{V_i}^2\ \ldots\ \mathbf{V_i}^{n_{hidden}/N}]$$
$$\mathbf{b_i} = [\mathbf{q_i}^1\ \mathbf{q_i}^2 \ldots\ \mathbf{q_i}^{n_{hidden}/N}]$$

$$\mathbf{W_h} = \begin{bmatrix} \mathbf{V_h}^1 \\ \mathbf{V_h}^2 \\ \vdots \\ \mathbf{V_h}^{n_{hidden}/N} \end{bmatrix}$$

Il est essentiel de noter que les matrices et vecteurs $\mathbf{W_i}$, $\mathbf{b_i}$ et $\mathbf{W_h}$ n'existe ici que conceptuellement et qu'on ne retrouve que les matrices et vecteurs partiels sur chacun des processeurs. En d'autres mots, la machine l ne possède en mémoire que les quantités $\mathbf{V_i}^l$, $\mathbf{q_i}^l$ et $\mathbf{V_h}^l$.

Propagation avant

Pour l'étape de propagation avant, on calcule tout d'abord des activations partielles sur chacune des machines. Le calcul se fait de la même façon qu'auparavant, mais en utilisant les matrice et vecteur partiels $\mathbf{V_i}^l$ et $\mathbf{q_i}^l$:

$$\mathbf{v}^l = tanh(\mathbf{e}^T\mathbf{V_i} + \mathbf{b_i})$$

On poursuit ensuite par le calcul des sorties brutes partielles en utilisant les matrices $\mathbf{V_h}^l$.

$$\mathbf{o}^l = (\mathbf{v}^l)^T\mathbf{V_h}^l + \mathbf{b_h}$$

Jusqu'à ce point, tous les calculs ont été effectués de façon indépendante et simultanée, sans aucune communication entre les machines. Pour continuer l'étape de propagation avant, chaque machine doit maintenant communiquer à toutes les autres le résultat de son calcul des sorties brutes partielles $\mathbf{o}^l$. On somme ensuite indépendemment sur chacune des machines ces résultats partiels pour obtenir la sortie brute usuelle :

$$\mathbf{o} = \sum_{l=1}^{N} \mathbf{o}^l$$

La sortie finale peut maintenant être obtenue, sur chacune des machines, à partir des sorties brutes. Chaque machine ayant une copie identique de la sortie $\mathbf{p}$:

$$\mathbf{p} = softmax(\mathbf{o})$$

On calcule finalement la fonction de coût qui sera utilisée par l'étape de propagation arrière. L'étape de propagation avant est maintenant terminée et toutes les machines possèdent les mêmes valeurs de sorties.

Propagation arrière

Pour l'étape de propagation arrière, on calcule les corrections sur les poids $\mathbf{V_i}^l$, $\mathbf{b_i}$, $\mathbf{V_h}^l$ et $\mathbf{b_h}$ de manière identique à la section 2.3.3. C'est-à-dire qu'on calcul indépendemment sur chaque machine un vecteur $\mathbf{x}^l$ permettant de corriger les poids $\mathbf{V_h}^l$ et $\mathbf{b_h}$:

$$\mathbf{x}_k^l = (\mathbf{p}_k - \mathbb{I}_{\{i=cible\}})$$

On calcule maintenant un vecteur gradient partiel $\mathbf{y}^l$ sur chaque machine :

$$\mathbf{y}_j^l = (1 - (\mathbf{v}_j^l)^2) \sum_{k=1}^{n_{outputs}} \mathbf{x}_k^l \mathbf{V}_{\mathbf{h};k,j}$$

et on calcule les corrections sur les poids $\mathbf{V_i}^l$ et $\mathbf{b_i}^l$:

$$\left(\frac{\partial E_{NLL}}{\partial \mathbf{V_h}^l}\right) = \mathbf{x}^l(\mathbf{v}^l)^T$$
$$\left(\frac{\partial E_{NLL}}{\partial \mathbf{b_h}^l}\right) = \mathbf{x}^l$$

Le gradient partiel $\mathbf{y}^l$ est ensuite communiqué à chacune des autres machines et la somme calculée :

$$\mathbf{y} = \sum_{l=1}^{n_{hidden}/N} \mathbf{y}^l$$

La rétropropagation de l'erreur se poursuit avec le calcul d'un gradient partiel $\mathbf{u}^l$ sur le vecteur $\mathbf{e}$:

$$\mathbf{u}^l = \left(\frac{\partial E_{NLL}}{\partial \mathbf{e}}\right) = (\mathbf{y}^l)^T \mathbf{V_i}^l$$

On calcule ensuite les corrections pour les poids $\mathbf{V_i}^l$ et $\mathbf{b_i}^l$:

$$\left(\frac{\partial E_{NLL}}{\partial \mathbf{V_i}^l}\right) = \mathbf{y}^l\ \mathbf{e}^T$$
$$\left(\frac{\partial E_{NLL}}{\partial \mathbf{b_i}^l}\right) = \mathbf{y}^l$$

Le gradient partiel $\mathbf{u}^l$ est communiqué entre les machines et sommé localement :

$$\mathbf{u} = \sum_{l=1}^{n_{hidden}/N} \mathbf{u}^l$$

et les corrections finales sur les poids $\mathbf{e}$ sont enfin calculées :

$$\left(\frac{\partial E_{NLL}}{\partial \mathbf{e}}\right) = \mathbf{u}$$

Il est important de noter qu'à la fin de ce calcul, toutes les machines possèdent une copie identique des vecteurs caractéristiques.

Le cas où on décompose selon le nombre de sorties $|\mathcal{V}|$ est essentiellement similaire à l'exemple présenté ici, à l'exception du moment où les communications sont effectués. En pratique pour les réseaux présentés jusqu'ici, la décomposition selon le nombre des sorties est plus performante. Cela s'explique par la moins grande quantité de communications à effectuer (essentiellement à l'étape de propagation avant). On a plutôt présenté la décomposition selon le nombre d'unités cachées n_{hidden} puisque c'est celle qui se révèle la plus efficace pour le type de réseaux que l'on présente au prochain chapitre.

En pratique, aucune de ces deux méthodes ne fonctionne de façon satisfaisante sur un *cluster* avec les réseaux présentés jusqu'à maintenant étant donné la quantité importante de communications à effectuer. Cependant sur des machines à mémoire partagée, la parallélisation est relativement efficace. Il faut également noter la nécessité de maintenir en synchronisation les vecteurs caractéristiques sur chacune des machines. En effet, si on ne synchronise pas ces vecteurs, les sorties brutes partielles ne sont pas le résultat d'une décomposition du problème mais ne sont que des calculs partiels avec des entrées *différentes*.

2.3.5 NPLM avec échantillonnage

Soit θ un des paramètres du réseau de neurone ($\theta \in \omega$) présenté à la section précédente. On peut réécrire le gradient de l'erreur E par rapport au paramètre θ de la façon suivante :

$$\begin{aligned}\left(\frac{\partial E}{\partial \theta}\right) &= -\frac{\partial log(\mathbf{p}_{cible})}{\partial \theta} \\ &= \left(\frac{-1}{\mathbf{p}_{cible}}\right)\left\{\mathbf{p}_{cible}\left(\frac{\partial \mathbf{o}_{cible}}{\partial \theta}\right) - \frac{\mathbf{p}_{cible}}{\sum_{j=1}^{n_{outputs}} e^{\mathbf{o}_j}} \sum_{j=1}^{n_{outputs}} e^{\mathbf{o}_j}\left(\frac{\partial \mathbf{o}_j}{\partial \theta}\right)\right\} \\ &= -\left(\frac{\partial \mathbf{o}_{cible}}{\partial \theta}\right) + \sum_{j=1}^{n_{outputs}} \mathbf{p}_j\left(\frac{\partial \mathbf{o}_j}{\partial \theta}\right)\end{aligned} \tag{2.19}$$

La sommation de l'équation précédente est le terme qui représente la plus grande partie du travail computationnel à effectuer puisqu'elle nécessite le calcul de la fonction de partition $Z = \sum_{j=1}^{n_{outputs}} e^{\mathbf{o}_j}$ et par le fait même le calcul de toutes les sorties $\mathbf{o}_j$. Si on remarque que cette sommation représente une espérance sur la distribution de probabilité *effective*[12] du réseau de neurones (dénotée par P), on peut réécrire l'équation originale ainsi :

$$\left(\frac{\partial E}{\partial \theta}\right) = -\left(\frac{\partial \mathbf{o}_{cible}}{\partial \theta}\right) + E_P\left[\left(\frac{\partial \mathbf{o}_j}{\partial \theta}\right)\right]$$

[12]La distribution de probabilité donnée par le réseau de neurones et l'instance des valeurs de ses paramètres ω.

Si on veut calculer exactement cette espérance, on doit tirer un point pour chaque sortie ($i = 1, 2, \ldots, n_{outputs}$). Cela se fait en calculant chacune des sorties j de la façon suivante :

$$\mathbf{o}_j = \mathbf{a}^T \mathbf{W_h}^j + \mathbf{b_h}_j$$

où $\mathbf{a}$ est l'activation et $\mathbf{W_h}^j$ est une sous-matrice de $\mathbf{W_h}$ contenant la rangée des poids nécessaires au calcul de $\mathbf{o}_j$. Il s'agit en effet du même calcul que celui présenté à la section précédente pour le *NPLM*. Le calcul des $\mathbf{p}_j$ est aussi nécessaire mais est obtenu directement des $\mathbf{o}_j$ de façon peu coûteuse.

L'objectif est donc d'évaluer l'espérance de façon approximative au lieu de la calculer complètement. On désire donc procéder en ne calculant qu'un sous-ensemble des sorties $\mathbf{o}_j$. Or cela implique qu'on doive maintenant avoir une approximation des $\mathbf{p}_j$ (eq. 2.19) puisque ces quantités nécessitent normalement le calcul de toutes les sorties $\mathbf{o}_j$. La méthode retenue dans [3] et [4] est d'utiliser une méthode d'échantillonnage pondéré (en anglais : *importance sampling* ; voir [16] et [17]) pour estimer les $\mathbf{p}_j$ et de sélectionner le nombre de points tirés de façon adaptative. On présente une courte introduction à l'échantillonnage pondéré à la prochaine section.

Échantillonnage pondéré

Soit P une fonction de probabilité définie sur un ensemble fini $\mathcal{X}$. L'espérance d'une fonction $h(x)$ évaluée sur P est donnée par :

$$E_P[h(X)] = \sum_{x \in \mathcal{X}} h(x)P(x)$$

Soit ensuite Q une autre fonction de probabilité et de même support que P. On dit que c'est la distribution *instrumentale*.

L'objectif est ici d'introduire une distribution Q pour laquelle il est plus facile de tirer des échantillons que la distribution originale P. On réécrit l'équation précédente en terme d'une espérance évaluée sur Q de la façon suivante :

$$\begin{aligned} E_P[h(X)] &= \sum_{x \in \mathcal{X}} h(x) \frac{P(x)}{Q(x)} Q(x) \\ &= E_Q\left[h(X)\frac{P(X)}{Q(X)}\right] \end{aligned}$$

En tirant m points $(x_1, x_2, \ldots, x_m)$ de la distribution Q, on obtient un estimateur de $E_P[h(X)]$:

$$E_P[h(X)] \approx \frac{1}{m} \sum_{i=1}^{m} h(x_i) \frac{P(x_i)}{Q(x_i)}$$

Malheureusement il est encore nécessaire d'évaluer la fonction de partition Z lors du calcul des $P(x_i)$.

Variante (échantillonnage pondéré biaisé)

Soit $P(x) = \frac{1}{Z} f(x)$ où $Z = \sum_{x \in \mathcal{X}} f(x)$ est la fonction de partition. On peut réécrire Z sous la forme d'une espérance sous la distribution uniforme U :

$$Z = E_U[Nf(x)] = NE_U[f(x)], \quad \text{où } N = |\mathcal{X}|$$

Si on utilise l'échantillonnage pondéré, on obtient l'équation suivante :

$$Z = NE_U[f(x)] = E_Q\left[\frac{f(X)}{Q(X)}\right]$$

Soit $\{x_i\}_{i=1}^m$, les points tirés de Q. On estime alors Z par :

$$Z \approx \frac{1}{m} \sum_{i=1}^{m} \frac{f(x_i)}{Q(x_i)}$$

et $E_P[h(X)]$ par

$$\begin{aligned} E_P[h(X)] &\approx \frac{1}{m} \sum_{i=1}^{m} h(x_i) \frac{P(x_i)}{Q(x_i)} \\ &= \frac{1}{mZ} \sum_{i=1}^{m} h(x_i) \frac{f(x_i)}{Q(x_i)} \\ &\approx \frac{1}{\sum_{j=1}^{m} \frac{f(x_j)}{Q(x_j)}} \sum_{i=1}^{m} h(x_i) \frac{f(x_i)}{Q(x_i)} \end{aligned}$$

2.4 WordNet

WordNet est une base de données lexicale compilée manuellement par une équipe de linguistes, sous la direction de George A. Miller, de l'université Princeton. La base de données consiste en un certain nombre d'ensembles de mots (noms, verbes, adjectifs et adverbes) appelés *synsets* et regroupant des entités lexicalement proches. Les synsets sont des ensemble de mots synonymes (*syn*onyme-*sets*) qui regroupent ainsi des mots de sens similaires et permettant une organisation sémantique des mots du vocabulaire (anglais). On dit alors qu'un synset définit implicitement (par ses mots) un concept linguistique. En d'autres mots, les sens (*synsets*) sont identifiés et définis par les mots qu'ils contiennent.

Par exemple, le navigateur *WordNet* produit les entrées suivantes pour le mot *demonstration* :

```
The noun demonstration has 5 senses (first 4 from tagged texts)

1. (8) presentation, presentment, demonstration
    -- (a show or display;
        the act of presenting something to sight or view;
        "the presentation of new data";
        "he gave the customer a demonstration")
2. (5) demonstration
    -- (a show of military force or preparedness;
        "he confused the enemy with feints and demonstrations")
3. (3) demonstration, manifestation
    -- (a public display of group feelings (usually of a political nature);
        "there were violent demonstrations against the war")
4. (2) demonstration, monstrance
    -- (proof by a process of argument or a series of proposition proving an
        asserted conclusion)
5. demonstration, demo
    -- (a visual presentation showing how something works;
        "the lecture was accompanied by dramatic demonstrations";
        "the lecturer shot off a pistol as a demonstration of the startle
         response")
```

Il existe donc, pour ce mot, 5 sens distincts définis par différents ensembles de mots (contenant bien sûr *demonstration*) et détaillés par le contenu entre parenthèses. Ainsi le premier sens, défini par l'ensemble de mot $\{presentation, presentment, demonstration\}$ réfère à un concept distinct de celui définit par l'ensemble $\{demonstration, manifestation\}$.

WordNet permet d'utiliser ces ensembles de mots afin de générer une *ontologie*, c'est-à-dire un arbre de dépendances entre les différents *synsets*. Une ontologie est créée à partir du choix de point de vue que l'on choisit. On peut construire l'arbre selon différentes relations : hypernymie (ou inversement hyponymie), méronymie ou synonymie (inversement antonymie). Par exemple, la mise en relation des *synsets* par la relation d'hypernymie (la

relation dite "IS-A") donne ainsi un arbre débutant à un noeud racine, qui définit *le* sens le plus général, et allant vers des concepts de plus en plus précis et restreints. De plus, les *synset*s de plus haut niveau sont *englobants* dans ce sens que pour un *synset* donné, tous les *synset*s de plus bas niveau qui y sont reliés sont des instances (plus précises) du sens défini par le *synset* original. En d'autres mots, les *synset*s de plus bas niveau représentent des concepts qui sont des sous-ensembles des *synset*s de plus haut niveau auxquels ils sont attachés. Par exemple, voici comment s'organise la hiérarchie des sens pour le dernier sens du mot *demonstration*, débutant par la racine :

```
ROOT
    -- (root concept)
NOUN
    -- (noun concept)
abstraction
    -- (a general concept formed by extracting common features from specific
        examples)
relation
    -- (an abstraction belonging to or characteristic of two entities or
        parts together)
social relation
    -- (a relation between living organisms (especially between people))
communication
    -- (something that is communicated by or to or between people or groups)
visual communication
    -- (communication that relies on vision)
demonstration, demo
    -- (a visual presentation showing how something works;
        "the lecture was accompanied by dramatic demonstrations";
        "the lecturer shot off a pistol as a demonstration of the startle
         response")
```

Tout les mots sont représentés sous le noeud *ROOT*. Le noeud *NOUN*, enfant de *ROOT*, présente déjà un sous-ensemble des mots restreint aux noms du langage. Chaque niveau de l'arbre représente un sous-ensemble toujours plus restreint et ainsi permet de décrire des sens de plus en plus précis.

Comme on vient de le voir par l'exemple, une ontologie représente de l'information utile sur la structure du langage. Dans le prochain chapitre on présente une méthode permettant d'utiliser l'information contenue dans une ontologie construite à partir de la relation d'hypernymie afin d'entraîner un réseau de neurones de type *NPLM*.

Chapitre 3

Modélisation par réseaux hiérarchiques

Ce chapitre présente la méthode centrale étudiée dans cette recherche. C'est une méthode parente des autres approches utilisant des réseaux de neurones. Le but premier de cette nouvelle méthode est d'utiliser une décomposition sémantique afin d'accélérer l'entraînement et l'opération de test des *NPLM*s.

L'idée maîtresse est d'utiliser de l'information *a priori* sur la structure du langage pour permettre de segmenter le problème de modélisation. On prend ainsi avantage de l'information linguistique offerte par le logiciel *WordNet* pour effectuer une décomposition sémantique. Cette décomposition déterminera en partie la topologie finale du réseau de neurones et intégrera donc implicitement de l'information linguistique avant l'entraînement comme tel.

3.1 Décomposition sémantique

On désire toujours modéliser la probabilité :

$$P(w_1, w_2, \ldots, w_L) = P(w_1)P(w_2|w_1)\ldots P(w_l|w_1, \ldots, w_{L-1})$$

On utilise une approximation similaire à celle du trigramme en définissant l'hypothèse simplificatrice suivante :

$$P(w_i|w_1 \ldots w_i - 1) \approx P(w_i|w_{i-l+1}, w_{i-l+2}, \ldots, w_{i-1}) \tag{3.1}$$

où l est la dimension de la fenêtre utilisée.

Supposons qu'on divise les mots du vocabulaire en deux ensembles disjoints de mots E_1 et E_2, de dimensions similaires. On pourrait utiliser un réseau de neurones pour modéliser les probabilités suivantes :

$$P(E_j|w_{i-k+1}, w_{i-k+2}, \ldots, w_{i-1}), \qquad j = 1, 2$$

Si on continue à diviser ces ensembles récursivement jusqu'à ce que chaque ensemble terminal ne contiennent qu'un seul mot, on peut construire un arbre binaire incorporant une partition

hiérarchique des mots. Un noeud donné de l'arbre représentera un ensemble de mots et chacun de ses noeuds enfants sera un sous-ensemble disjoint de l'ensemble original. Muni d'un tel arbre, on peut également construire un réseau de neurones permettant de calculer la probabilité d'un noeud enfant étant donné les mots d'un contexte. On peut réécrire l'éq. 3.1 ainsi :

$$P(w_i|w_{i-k+1}, w_{i-k+2}, \ldots, w_{i-1}) = \prod_{j \in \mathcal{P}_i} P(n_j|w_{i-k+1}, w_{i-k+2}, \ldots, w_{i-1}, parent(n_j))$$
$$P(w_i|h_t) = \prod_{j \in \mathcal{P}_i} P(n_j|h_t, parent(n_j))$$

où n_j sont des indices référants aux différents noeuds de l'arbre, $parent(\cdot)$ est une fonction retournant le noeud parent d'un noeud, h_t est un raccourci de notation pour le contexte $\{w_{i-k+1}, w_{i-k+2}, \ldots, w_{i-1}\}$ et $\mathcal{P}_i$ est le chemin (la liste des noeuds) allant de la racine au noeud feuille contenant le mot w_i. On prendra comme convention que $P(n_0|h_t, parent(n_0)) = P(n_0|h_t) = 1$ et on négligera ainsi d'inclure le noeud racine dans les chemins $\mathcal{P}_i$.

L'utilisation d'une décomposition sémantique permet d'accélérer les calculs de façon importante. Pour voir ceci, on note tout d'abord que pour le *NPLM*, l'utilisation de la fonction $softmax(\cdot)$ requiert le calcul de $|\mathcal{V}|$ sorties pour déterminer les probabilités des mots étant donné un contexte $h_t = \{w_{i-k+1}, w_{i-k+2}, \ldots, w_{i-1}\}$. Si on utilise une décomposition avec b noeuds enfants par noeud parents et qu'on suppose que l'arbre est balancé alors on doit calculer b sorties un total de $log_b(|\mathcal{V}|)$ fois, où $log_b(|\mathcal{V}|)$ est la profondeur d'un arbre balancé contenant $|\mathcal{V}|$ noeuds. L'accélération est ainsi dans $\mathcal{O}(\frac{|\mathcal{V}|}{log_b(|\mathcal{V}|)})$ par rapport à un *NPLM*. L'accélération est alors maximale pour b le plus petit possible, c'est-à-dire $b = 2$.

L'objectif est donc de construire des partitions de l'ensemble original des mots du vocabulaire qui aient un sens sémantique, c'est-à-dire qui incorpore de l'information sur le sens des mots. On pourrait imaginer qu'il s'agisse d'un arbre de décision et que chaque noeud de l'arbre corresponde à une question binaire sur la nature sémantique des mots. Les noeuds proches de la racine répondant à des questions relativement générales et les noeuds proches des feuilles répondant à des questions de plus en plus discriminantes. Il n'est évidemment pas pratique de procéder manuellement et de définir les questions partitionnant les mots. On aimerait idéalement procéder automatiquement en concevant des questions d'ordre distributionnelles, c'est-à-dire des questions partionnant un ensemble de mots en deux de façon optimale selon des critères statistiques.

Finalement, si on permet de partitionner les ensembles de façon non-disjointe, il est possible qu'un mot se retrouve dans plus d'un noeud terminal. Il existe alors plus d'un chemin menant à un mot et on peut donc réécrire l'équation précédente ainsi :

$$\begin{aligned} P(w_i|h_t) &= \sum_k \prod_{j \in \mathcal{P}_{i,k}} P(n_j|h_t, parent(n_j)) \\ &= \sum_k P(\mathcal{P}_{i,k}) \end{aligned} \tag{3.2}$$

où $\mathcal{P}_{i,k}$ réfère au $k^{\text{ème}}$ chemin menant au mot w_i et $P(\mathcal{P}_{i,k})$ réfère à la probabilité de ce chemin. Cela permet de raffiner la décomposition sémantique en permettant aux mots d'avoir plus

d'un sens. Il faut noter que dans le cas ou plusieurs chemins mènent au même mot, il existe une superposition des chemins dans l'arbre et que plusieurs termes $P(n_j|h_t, parent(n_j))$ sont répétés mais n'ont besoin d'être calculés qu'une seule fois.

3.2 Espace des caractéristiques

La méthode de réseau de neurones hiérarchique utilise la même notion d'espace des caractéristiques que celle décrite à la section 2.3.2 pour les réseaux de neurones ordinaires. La seule différence est qu'en plus du passage des mots de l'entrée aux vecteurs caractéristiques on doit également procéder au passage des noeuds n_j à des vecteurs caractéristiques $\mathbf{n}_j$ étant donné que ces derniers font maintenant partie de l'entrée du réseau de neurones. L'entrée du réseau devient donc la concaténation des vecteurs caractéristiques des deux types :

$$\mathbf{e} = \mathbf{C}_t \oplus \mathbf{n}_j \tag{3.3}$$

3.3 Construction - clusterisation WordNet

La construction d'un arbre binaire incorporant une décomposition susceptible d'être utilisable et performante est loin d'être évidente. Une façon simple de faire est de considérer l'arbre produit par une ontologie *WordNet* selon la relation d'hypernymie. Cet arbre introduit naturellement une décomposition hiérarchique des sens des mots du vocabulaire. Cependant cet arbre n'est pas binaire. En effet, la majorité des noeuds possède plus de deux enfants et certains autres possèdent plus d'un parent. On présente donc un algorithme permettant la modification de l'arbre original en un arbre binaire.

Premièrement, comme le montre la figure 3.1, il faut faire un choix d'un *synset* parent dans le cas où un *synset* donné en possède plus d'un. Cette étape est effectuée manuellement, selon les connaissances linguistiques de l'auteur. Il s'agit de déterminer par les sens donnés par *WordNet* lequel des parents englobe ou justifie le mieux le sens de l'enfant. Par exemple, le *synset* suivant :

```
prosecutor, public prosecutor, prosecuting officer, prosecuting attorney
    -- (a government official who conducts criminal prosecutions on behalf
        of the state)
```

possède les parents suivants :

```
official, functionary
    -- (a worker who holds or is invested with an office)
lawyer, attorney
    -- (a professional person authorized to practice law; conducts lawsuits
        or gives legal advice)
```

Le second parent semble le plus approprié puisqu'il fait toujours référence au domain judiciaire, qui est l'élément important dans cet exemple. Il n'est pas évident de choisir lequel des parents est le plus approprié. Le prochain exemple est plus difficile :

```
person, individual, someone, somebody, mortal, human, soul
    -- (a human being; "there was too much for one person to do")
```

possède les parents suivants :

```
organism, being
    -- (a living thing that has (or can develop) the ability to act or
        function independently)
causal agent, cause, causal agency
    -- (any entity that causes events to happen)
```

Dans cet exemple, le premier parent semble être le sens souhaitable parce qu'un être humain est toujours un organisme tandis que ce n'est pas nécessairement le cas pour le deuxième parent. Les exemples montrent qu'il n'est pas toujours facile de choisir le parent le plus logique.

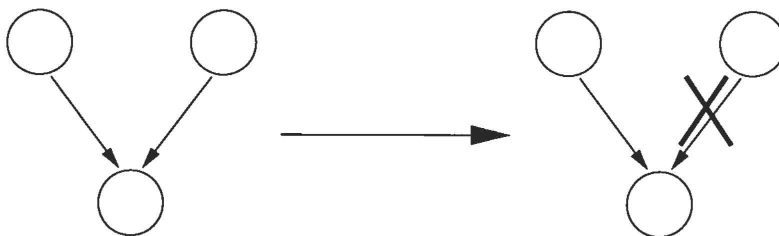

FIG. 3.1 – Enlèvement des parents multiples.

Deuxièmement, on choisit les sens des mots que l'on veut préserver dans l'arbre. Pour chaque mots du vocabulaire $\mathcal{V}$ il existe plusieurs sens pour chaque mot. Pour chaque sens répertorié dans la base de donnée *WordNet*, on comptabilise des fréquences et on garde seulement, par exemple, les 5 sens les plus fréquents de chaque mot. *WordNet* offre des fréquences pour un certain nombre de sens de la base de donnée, mais pas pour tous. On utilise ces fréquences "*WordNet*" avec les fréquences obtenues directement sur l'ensemble d'entraînement utilisé[1] pour calculer une fréquence pondérée permettant d'ordonner l'importance de chacun des sens des mots :

$$freq(s_i, w_j) = \alpha f(s_i, w_j) + (1 - \alpha) f(w_j)$$

où $f(s, w)$ est la fréquence du sens s du mot w retournée par *WordNet* et $f(w)$ est la fréquence du mot w calculée à partir de l'ensemble d'entraînement. Dans le cas où *WordNet*

[1]Dans ce cas, on n'a que l'information sur la fréquence des mots et non sur la fréquence des sens de chaque mot.

n'offre pas d'information pour un sens donné d'un mot, on prend $f(s,w) = 0$. En pratique on utilise $\alpha = 0.8$

Finalement, il faut s'assurer que chaque *synset* possède *au plus* deux enfants. C'est cette dernière étape qui requiert une clusterisation. La clusterisation consiste ici à identifier les *synsets* possédant plus de deux enfants et à regrouper les enfants en deux *synsets* composites. Ces *synsets* composites englobent les enfants originaux, de façon à ce que chaque *synset* ait au plus deux enfants. Chaque *synset* composite a comme enfant les *synsets* correspondant à une partie des enfants originaux. On répète la procédure récursivement sur tout l'arbre en partant de la racine. L'algorithme de clusterisation *K-Means* est présenté à la section 3.3.4.

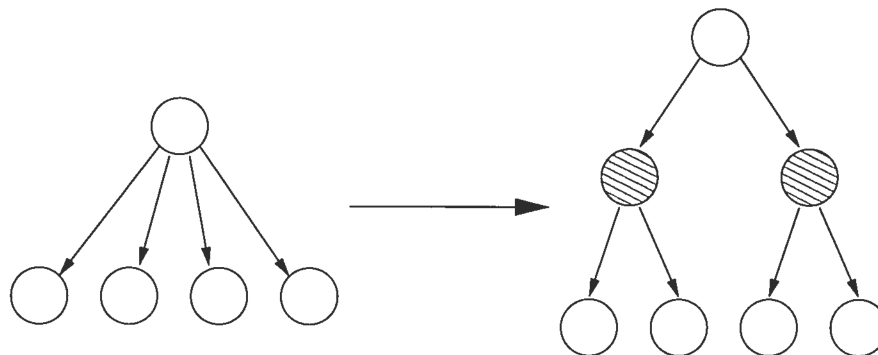

FIG. 3.2 – Binarisation de l'arbre hiérarchique.

Avant de décrire les détails de l'algorithme de clusterisation, on présente dans les prochaines sections les mesures de similarités utilisées pour mesurer la proximité entre deux mots du vocabulaire et entre deux synsets de l'ontologie.

3.3.1 *TF-IDF*

TF-IDF[2][18][19][13] est typiquement employée pour construire un *vecteur caractéristique* décrivant le contenu informationnel d'une série de documents. Cette méthode nécessite préalablement de diviser un texte (ou corpus) en un nombre n de documents. Elle construit ensuite un vecteur de dimension m (m sera la dimension du vocabulaire, i.e. : le nombre total de mots distincts considérés) pour chacun des n documents. Pour chaque document i on recueille ensuite les fréquences $\mathbf{F}_{i,j}$ (le nombre d'occurrences) du mot j et le nombre de documents contenant le mot j, $\mathbf{d}_j$. Le vecteur *TF-IDF* pour le document i aura finalement la forme suivante :

$$\mathbf{d}_i = (\mathbf{F}_{i,1} log(n/\mathbf{d}_1), \mathbf{F}_{i,2} log(n/\mathbf{d}_2), \ldots, \mathbf{F}_{i,m} log(n/\mathbf{d}_m))$$

La pondération par le factor *log* permet de donner plus d'importance aux mots *discriminants*, c'est-à-dire aux mots qui ont tendance à n'apparaître que dans des contextes limités.

[2] *Term Frequency - Inverse Document Frequency. Term Frequency* réfère à la fréquence des mots (termes) alors que *Document Frequency* dénote le nombre de documents vérifiant la présence d'un mot donné.

TF-IDF est utile pour faire de la clusterisation de documents. Or, l'objectif est ici de clusteriser par rapport aux mots. Pour ce faire, nous modifions légèrement la mesure de façon à produire un *vecteur caractéristique* (de dimension n, n étant le nombre de documents) pour chaque mot j d'un vocabulaire de m mots. Le vecteur pour le mot j aura la forme suivante :

$$\begin{aligned}\mathbf{m}_j &= (\mathbf{D}_{1,j}, \mathbf{D}_{2,j}, \ldots, \mathbf{D}_{n,j}) \\ &= (\mathbf{F}_{1,j} log(N_j/\mathbf{F}_{1,j}), \mathbf{F}_{2,j} log(N_j/\mathbf{F}_{2,j}), \ldots, \mathbf{F}_{n,j} log(N_j/\mathbf{F}_{n,j}))\end{aligned}$$

où N_j est la fréquence totale du mot j, c'est-à-dire la somme des fréquences de chacun des documents :

$$N_j = \sum_{i=1}^{n} \mathbf{F}_{i,j}$$

Chaque document défini une dimension du vecteur :

$$\mathbf{D}_{i,j} = \mathbf{F}_{i,j} log(N_j/\mathbf{F}_{i,j})$$

De plus, le facteur *log* permet de garder la comportement original de *TF-IDF* en pondérant à la hausse les dimensions qui ont une fréquence *relative*[3] plus élevée. Intuitivement, on s'attends à ce qu'un document traitant d'un sujet précis expriment non seulement un mot de façon fréquente, mais également que d'autres mots reliés au même sujet soient aussi utilisés avec une plus grande fréquence. Il est important de noter que les fréquences individuelles $\mathbf{F}_{i,j}$ ne sont pas particulièrement significatives. En effet, comme on le verra à la prochaine section, la mesure de similarité entre les mots se calcule sur des vecteurs normalisés.

3.3.2 Mesures de similarité entre mots

Le vecteur *TF-IDF* qui est calculé pour chacun des mots du vocabulaire peut maintenant être utilisé pour mesurer la similarité entre chacun des mots. La similarité est définie comme étant le produit scalaire (ou interne) entre deux vecteurs *TF-IDF* normalisés :

$$sim(\mathbf{m}_j, \mathbf{m}_k) = \frac{\mathbf{m}_j \cdot \mathbf{m}_k}{||\mathbf{m}_j|| \, ||\mathbf{m}_k||} \tag{3.4}$$

Une autre mesure de similarité, basée sur la distance euclidienne, est aussi envisageable. Soit

$$dist(\mathbf{m}_j, \mathbf{m}_k) = \left|\left| \frac{\mathbf{m}_j}{||\mathbf{m}_j||} - \frac{\mathbf{m}_k}{||\mathbf{m}_k||} \right|\right|$$

la distance normalisée entre deux mots. On peut définir la similarité en terme de la distance ainsi :

$$sim(\mathbf{m}_j, \mathbf{m}_k) = \frac{1}{dist(\mathbf{m}_j, \mathbf{m}_k)}$$

Dans ce mémoire, on utilisera exclusivement la similarité définie par l'équation 3.4.

[3] $\mathbf{F}_{i,j}/N_j$ est une fréquence relative, c'est-à-dire la proportion de tout les mots j exprimée dans le document i.

3.3.3 Mesure de similarité entre *synsets*

L'algorithme de clusterisation (présenté plus bas) ne fait pas la clusterisation sur des mots individuels mais sur des ensembles de mots (*synsets*). Pour ce faire, il sera nécessaire de pouvoir calculer la similarité entre deux ensembles de mots. On calcule donc un vecteur caractéristique pour chacun des *synsets* à partir des vecteurs caractéristiques de chacun des mots présents dans le *synset*. Le vecteur calculé pour chaque *synset* aura la même dimension que les vecteurs de mots (i.e. : une dimension de n).

$$\mathbf{v}_k = (\mathbf{V}_{1,k}, \mathbf{V}_{2,k}, \ldots, \mathbf{V}_{n,k})$$

où $\mathbf{V}_{i,k}$ est défini comme suit :

$$\mathbf{V}_{i,k} = median_{j \in synset_k} \left\{ \frac{\mathbf{m}_{i,j}}{\|\mathbf{m}_j\|} \right\}$$

En d'autres mots, chaque composante i du vecteur $\mathbf{v}_k$ est la médiane des composantes i des vecteurs pour les mots associés au *synset* k. La mesure de similarité entre deux *synsets* s'exprime naturellement comme suit :

$$sim(\mathbf{v}_k, \mathbf{v}_l) = \mathbf{v}_k \cdot \mathbf{v}_l$$

L'objectif est ici de trouver un *centre* pour chacun des synsets dans l'espace caractéristique. La similarité entre *synsets* s'explique alors intuitivement comme la projection d'un vecteur caractéristique sur l'autre.

3.3.4 Algorithme *K-Mean* (pour $k = 2$)

On désire modifier un arbre dont les noeuds sont des ensembles de mots. Chaque noeud possède une mesure permettant d'estimer la similarité avec un autre noeud. L'objectif est de regrouper les noeuds proches de façon à obtenir comme résultat final un arbre binaire. L'algorithme *K-Mean* est utilisé sur les *synsets* possédant plus de deux enfants.

1 **begin**
2 Initialiser μ comme la moyenne des vecteurs des synsets : $\mu = \frac{1}{n}\sum_n \mathbf{v}_n$
3 Générer μ^1 et μ^2 à partir de μ en ajoutant du bruit : $\mu^i = \mu + \epsilon$
4 **faire** Classer les *synsets* enfants au groupe de moyenne μ^1 ou μ^2
(synset $k \in$ groupe 1 si $sim(\mathbf{v}_k, \mu^1) > sim(\mathbf{v}_k, \mu^2)$)
5 Recalculer μ^1 et μ^2 selon l'appartenance des *synsets*
6 **tant que** μ^1 ou μ^2 changent
7 Créer enfants composites des *synsets* regroupés autour de μ^1 et μ^2
8 Remplacer les enfants par les enfants composites
9 **end**

Algorithme 1 : Clusterisation *K-Mean*

En utilisant la mesure de similarité entre *synsets* présentée précédemment, on regroupe les enfants sous deux *synsets* composites. En d'autres mots, pour tous les *synsets* possédant plus de deux enfants, on remplace ses enfants par deux *synsets* composites. Les enfants de ces nouveaux *synsets* seront les enfants originaux. La figure 3.2 illustre le processus de clusterisation.

L'algorithme *K-Mean* procède itérativement en mettant à jour deux centres μ^1 et μ^2 et en utilisant la similarité entre chaque noeud enfant et les centres pour déterminer à quel centre chaque noeud appartient. Une fois l'algorithme terminé et l'appartenance des enfants à un des deux centres déterminé, l'arbre est mis à jour en remplaçant les noeuds enfants par deux noeuds composites. Chaque noeud composite aura comme enfant les noeuds d'un des deux centres.

3.4 Détails de l'algorithme

Dans cette section, on présente les équations pour le *NPLM hiérarchique*. Elles sont semblables à celles pour le *NPLM* présentées à la section 2.3.3, sauf pour quelques modifications.

Pour simplifier la présentation, on assumera que deux noeuds ayant le même parent seront dénotés par des indices successifs, par exemple, n_j et n_{j+1}, de telle sorte que :

$$parent(n_j) = parent(n_{j+1})$$

On note tout d'abord le fait qu'étant donné qu'il s'agit d'un arbre binaire et que les sorties des réseaux de neurones ont seulement 2 sorties, on aura par la définition de la fonction $softmax$ que $\mathbf{p}_i = 1 - \mathbf{p}_{i+1}$. Dans ce cas, on remarque (voir eq. 2.6) que :

$$\left(\frac{\partial \mathbf{p}_i}{\partial \mathbf{o}_i}\right) = \mathbf{p}_i(1-\mathbf{p}_i) = (1-\mathbf{p}_{i+1})\mathbf{p}_{i+1} = \left(\frac{\partial \mathbf{p}_i}{\partial \mathbf{o}_{i+1}}\right)$$

Pour alléger la notation, on abrégera certaines expressions. Tout d'abord, on écrira $P(w_i|h_t)$ ainsi : $P(w_i)$, prenant pour acquis que le contexte h_t est implicitement sous-entendu. Ensuite, on remplacera par $\mathbf{p}_j$ l'expression $P(n_j|h_t, \text{parent}(n_j))$. Finalement, on utilisera la notation $P(\mathcal{P}_{i,k})$ à la place de l'expression $\prod_{j\in\mathcal{P}_{i,k}} P(n_j|h_t, \text{parent}(n_j))$ pour indiquer explicitement qu'il s'agit de la probabilité *cumulée* le long d'un chemin de l'arbre.

3.4.1 Propagation avant

On désire tout d'abord calculer la probabilité suivante :

$$P(n_j|h_t, parent(n_j)) \tag{3.5}$$

C'est-à-dire la probabilité d'un noeud sachant son parent et un contexte h_t. L'entrée $\mathbf{e}$ (voir eq. 3.3) du réseau de neurones est donc maintenant non seulement fonction du contexte h_t mais également du noeud parent de n_j pour lequel on veut calculer la probabilité *a posteriori* $P(n_j|h_t, parent(n_j))$.

Tel que mentionné à la section 3.2, similairement au fait que chaque mots w_i est associé à un vecteur caractéristique $\mathbf{C}^{w_i}$, chaque noeud $n_{parent(n_j)}$ est également associé à un vecteur caractéristique $\mathbf{n}_{parent(n_j)}$ de longueur $l(\mathbf{n}_{parent(n_j)}) = l(code_noeuds)$[4]. L'entrée $\mathbf{e}$ est alors calculée comme étant la concaténation de $C(h_t) = \mathbf{C}_t$ avec $\mathbf{n}_{parent(n_j)}$.

On calcule la probabilité de l'équation 3.5 de la façon suivante :

$$\mathbf{p}(parent(n_j)) = softmax((tanh(\mathbf{e}^T\mathbf{W_i} + \mathbf{b_i}))^T\mathbf{W_h}(parent(n_j)) + \mathbf{b_h}(parent(n_j))) \quad (3.6)$$

où $\mathbf{p}(parent(n_j))$ est un vecteur de dimension 2 et $\mathbf{W_h}(parent(n_j))$ et $\mathbf{b_h}(parent(n_j))$ sont des sous-matrice et sous-vecteur de $\mathbf{W_h}$ et $\mathbf{b_h}$, obtenus en fonction du noeud n_j pour lequel on désire la probabilité[5] $\mathbf{p}_j$. $\mathbf{W_h}$ est une matrice de dimension $n_{hidden} * (nombre\ total\ de\ noeuds)$ alors que $\mathbf{b_h}$ est un vecteur de dimension $(nombre\ total\ de\ noeuds)$. Il existe donc une colonne de $\mathbf{W_h}$ ($\mathbf{W_h}(n_k)$) et un biais ($\mathbf{b_h}(n_k)$) pour chaque noeud k.

Le calcul de $\mathbf{e}^T\mathbf{W_i}$ peut maintenant être décomposé en deux parties indépendantes. Tout d'abord, on décompose la matrice $\mathbf{W_i}$ en deux sous-matrices :

$$\mathbf{W_i} = \begin{bmatrix} \mathbf{W_i}^* \\ \mathbf{W_i}^c \end{bmatrix}$$

où $\mathbf{W}_i^*$ est de mêmes dimensions que la matrice $\mathbf{W_i}$ de la section 2.3.2, c'est-à-dire $n_{inputs} * n_{hidden}$, et la matrice $\mathbf{W}_i^c$ est de dimensions $l(code_noeuds) * n_{hidden}$.

$\mathbf{W_i}^*$ est donc la matrice qui est multipliée par la concaténation $\mathbf{C}_t$ des vecteurs caractéristiques correspondants aux mots du contexte h_t. $\mathbf{W_i}^c$ est la matrice qui est multipliée par le vecteur caractéristique $\mathbf{n}_{parent(n_j)}$ du noeud $parent(n_j)$. Le calcul de la préactivation $\mathbf{s}$ s'écrit maintenant ainsi :

$$\begin{aligned} \mathbf{s} &= \mathbf{e}^T\mathbf{W_i} + \mathbf{b_i} \\ &= \mathbf{C}_t^T\mathbf{W_i}^* + \mathbf{n}_{parent(n_j)}^T\mathbf{W_i}^c + \mathbf{b_i} \\ &= (\mathbf{C}_t^T\mathbf{W_i}^* + \mathbf{b_i}) + \mathbf{n}_{parent(n_j)}^T\mathbf{W_i}^c \end{aligned} \quad (3.7)$$

Le calcul de l'équation 3.6 est donc similaire au calcul de la probabilité pour un *NPLM* (voir éq. 2.15). La figure 3.3 illustre la topologie d'un réseau de neurones hiérarchique. Elle diffère de celle présentée à la section 2.2.2 par le calcul d'une probabilité sur des noeuds au lieu de sur des mots.

La quantité $P(w_i|h_t)$ implique donc le calcul des probabilités $P(n_j|h_t, parent(n_j))$, pour chaque noeud n_j apparaissant dans l'équation 3.2. Cependant, lors du calcul de $\mathbf{s}$, on prend avantage de la décomposition de l'équation 3.7 et on note que la quantité $\mathbf{C}_t^T\mathbf{W_i}^* + \mathbf{b_i}$ n'a besoin d'être calculée qu'une seule fois pour tout les noeuds. Le calcul de $\mathbf{n}_{parent(n_j)}^T\mathbf{W_i}^c$ doit cependant être effectué pour chaque $j \in \mathcal{P}_{i,k}$, c'est-à-dire pour chaque noeud sur le(s) chemin(s) allant de la racine au mot désiré.

[4] la longueur des vecteurs caractéristiques des noeuds, $l(code_noeuds)$, est typiquement 30.

[5] On dénote les deux composantes de $\mathbf{p}(parent(n_j))$ par $\mathbf{p}_j$ et $\mathbf{p}_{j+1}$.

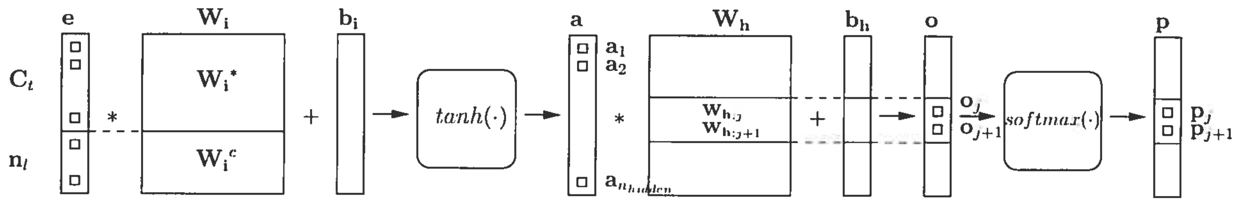

FIG. 3.3 – Topologie d'un réseau de neurones hiérarchique

3.4.2 Propagation arrière

Prenant en considération le résultat de la décomposition sémantique (eq. 3.2), l'expression de l'erreur E_{NLL} de l'équation 2.2.2 devient

$$\begin{aligned} E_{NLL} &= -log(P(w_i|h_t)) \\ &= -log\Big(\sum_k \prod_{j \in \mathcal{P}_{i,k}} P(n_j|h_t, \text{parent}(n_j))\Big) \\ &= -log\Big(\sum_k \prod_{j \in \mathcal{P}_{i,k}} \mathbf{p}_j\Big) \\ &= -log\Big(\sum_k P(\mathcal{P}_{i,k})\Big) \end{aligned} \tag{3.8}$$

où w_i est le mot cible. Il faut prendre note qu'une probabilité $\mathbf{p}_j$ peut apparaître plus d'une fois dans l'expression précédente, soit sous sa forme directe ou soit sous sa forme complément (i.e. $(1 - \mathbf{p}_j)$).

Soit $\mathcal{I}(j)$ les chemins $\mathcal{P}_{i,k}$ contenant le noeud j (i.e. dont le terme $\mathbf{p}_j$ apparaît directement dans l'expression de $P(\mathcal{P}_{i,k})$). Soit $\tilde{\mathcal{I}}(j)$ les chemins $\mathcal{P}_{i,k}$ contenant le noeud frère de j (i.e. dont le terme $(1-\mathbf{p}_j)$ apparaît directement dans l'expression de $P(\mathcal{P}_{i,k})$. On peut maintenant écrire les dérivées partielles de la sortie aux sorties brutes :

$$\begin{aligned}
\left(\frac{\partial E_{NLL}}{\partial \mathbf{o}_j}\right) &= \left(\frac{\partial E}{\partial \mathbf{p}_i}\right)\left(\frac{\partial \mathbf{p}_i}{\partial \mathbf{o}_j}\right) \\
&= \frac{-1}{P(w_i)}\left(\frac{\partial P(w_i)}{\partial \mathbf{p}_i}\right)\left(\frac{\partial \mathbf{p}_i}{\partial \mathbf{o}_j}\right) \\
&= \frac{-1}{P(w_i)}\left(\frac{\partial}{\partial \mathbf{p}_i}\left\{\sum_{l\in\mathcal{I}(k)} P(\mathcal{P}_{i,l}) + \sum_{l\in\bar{\mathcal{I}}(k)} P(\mathcal{P}_{i,l})\right\}\right)\left(\frac{\partial \mathbf{p}_i}{\partial \mathbf{o}_j}\right) \\
&= \frac{-1}{P(w_i)}\left(\sum_{l\in\mathcal{I}(k)} \frac{\partial P(\mathcal{P}_{i,l})}{\partial \mathbf{p}_i} + \sum_{l\in\bar{\mathcal{I}}(k)} \frac{\partial P(\mathcal{P}_{i,l})}{\partial \mathbf{p}_i}\right)\left(\mathbf{p}_i(\mathbb{I}_{\{i=j\}} - \mathbf{p}_j)\right) \\
&= \frac{-1}{P(w_i)}\left(\sum_{l\in\mathcal{I}(k)} \frac{P(\mathcal{P}_{i,l})}{\mathbf{p}_i} - \sum_{l\in\bar{\mathcal{I}}(k)} \frac{P(\mathcal{P}_{i,l})}{1-\mathbf{p}_i}\right)(\mathbf{p}_i(\mathbb{I}_{\{i=j\}} - \mathbf{p}_j)) \\
&= \frac{-(\mathbb{I}_{\{i=j\}} - \mathbf{p}_j)}{P(w_i)}\left(\sum_{l\in\mathcal{I}(k)} P(\mathcal{P}_{i,l}) - \frac{\mathbf{p}_i}{1-\mathbf{p}_j}\sum_{l\in\bar{\mathcal{I}}(k)} P(\mathcal{P}_{i,l})\right)
\end{aligned} \tag{3.9}$$

On peut directement remplacer cette dernière expression dans l'équation 2.16 originale. Par un travail similaire, on obtient les expressions se substituant dans les équations 2.17 et 2.18. On note au passage le cas "simple" où il n'existe qu'un seul chemin, $P(w_i) = P(\mathcal{P}_{i,1})$:

$$\left(\frac{\partial E_{NLL}}{\partial \mathbf{o}_j}\right) = -(\mathbb{I}_{\{i=j\}} - \mathbf{p}_j)$$

Calcul efficace

Le calcul efficace de la rétropropagation de l'erreur s'effectue de façon similaire à celui pour le *NLPM* présenté à la section 2.3.3. Pour chaque noeud de l'arbre, on calcul tout d'abord un vecteur gradient $\mathbf{x}$ ainsi :

$$\mathbf{x}_j = \left(\frac{\partial E_{NLL}}{\partial \mathbf{o}_j}\right) \tag{3.10}$$

où j dénote chacune des 2 sorties pour chaque noeud. On calcul ensuite un gradient $\mathbf{y}$

$$\mathbf{y}_k = \sum_j \left(\frac{\partial E_{NLL}}{\partial \mathbf{o}_j}\right)\left(\frac{\partial \mathbf{o}_j}{\partial \mathbf{a}_k}\right)\left(\frac{\partial \mathbf{a}_k}{\partial \mathbf{s}_k}\right) \tag{3.11}$$

On poursuit par le calcul des corrections pour les poids de $\mathbf{W_h}$ et $\mathbf{b_h}$ correspondants au noeud n_j (c'est-à-dire la sous-matrice $\mathbf{W_h}(parent(n_j))$ et le sous-vecteur $\mathbf{b_h}(parent(n_j))$) :

$$\begin{aligned}
\left(\frac{\partial E_{NLL}}{\partial \mathbf{W_h}(parent(n_j))}\right) &= \mathbf{x}\mathbf{a}^T \\
\left(\frac{\partial E_{NLL}}{\partial \mathbf{b_h}(parent(n_j))}\right) &= \mathbf{x}
\end{aligned} \tag{3.12}$$

On doit maintenant calculer les corrections sur l'entrée $\mathbf{e} = \mathbf{C}_t \oplus \mathbf{n}_j$:

$$\begin{aligned}\left(\frac{\partial E_{NLL}}{\partial \mathbf{e}}\right) &= \mathbf{y}^T \mathbf{W}_\mathrm{i} \\ &= \mathbf{y}^T \begin{bmatrix}\mathbf{W}_\mathrm{i}{}^* \\ \mathbf{W}_\mathrm{i}{}^c\end{bmatrix} \\ &= \mathbf{y}^T \mathbf{W}_\mathrm{i}{}^* + \mathbf{y}^T \mathbf{W}_\mathrm{i}{}^c\end{aligned}$$

En pratique on appliquera la correction immédiatement sur les poids des vecteurs caractéristiques de noeuds :

$$\left(\frac{\partial E_{NLL}}{\partial \mathbf{n}_j}\right) = \mathbf{y}^T \mathbf{W}_\mathrm{i}{}^c \tag{3.13}$$

où $\mathbf{n}_j$ est le vecteur caractéristique du noeud n_j (voir eq. 3.3). Cependant, les corrections sur les poids des vecteurs caractéristiques de mots seront appliquées en dernier lieu. On accumulera donc dans un vecteur $\mathbf{z}$ la somme des vecteurs $\mathbf{y}$ de chaque noeud. De la même façon, on appliquera les corrections sur la matrice $\mathbf{W}_\mathrm{i}{}^c$ immédiatement, mais on attendra pour les corrections de $\mathbf{W}_\mathrm{i}{}^*$ et $\mathbf{b}_\mathrm{i}$:

$$\left(\frac{\partial E_{NLL}}{\partial \mathbf{W}_\mathrm{i}{}^c}\right) = \mathbf{y}\ \mathbf{n}_j^T \tag{3.14}$$

Finalement, lorsque les corrections précédentes ont été effectuées pour chaque noeud, on utilise le vecteur $\mathbf{z}$ des gradients accumulés pour finaliser l'étape de propagation arrière. On calcul tout d'abord les corrections sur les vecteurs caractéristiques de mots :

$$\left(\frac{\partial E_{NLL}}{\partial \mathbf{C}_t}\right) = \mathbf{z}^T \mathbf{W}_\mathrm{i}{}^* \tag{3.15}$$

et on termine par les corrections pour la matrice de poids $\mathbf{W}_\mathrm{i}{}^*$:

$$\begin{aligned}\left(\frac{\partial E_{NLL}}{\partial \mathbf{W}_\mathrm{i}{}^*}\right) &= \mathbf{z}\ \mathbf{C}_t^T \\ \left(\frac{\partial E_{NLL}}{\partial \mathbf{b}_\mathrm{i}{}^*}\right) &= \mathbf{z}\end{aligned} \tag{3.16}$$

3.5 Parallélisation

Une méthode de parallélisation simple et efficace consiste à distribuer les noeuds des chemins parmis les M machines disponibles selon leurs profondeurs. Soit $prof(n_j)$ la *profondeur* du noeud n_j dans l'arbre binaire. Par exemple la profondeur du noeud racine est 0 et la profondeur de ses enfants immédiats est 1. Soit $m \in \{0, \ldots, M-1\}$ une des machines. Ainsi la machine m fera les calculs sur l'ensemble des noeuds suivants :

$$\mathcal{N}_m = \{j \mid prof(n_j)\ mod\ M = m\} \tag{3.17}$$

À titre d'exemple, si on suppose qu'il y a un total de $M = 2$ machines et que le calcul de la propabilité du mot w_i sachant h_t implique un seul chemin de profondeur 4, alors selon l'équation 3.2, on doit calculer :

$$p(w_i|h_t) = P(n_1|h_t, n_0)P(n_2|h_t, n_1)P(n_3|h_t, n_2)P(n_4|h_t)$$

Selon l'équation 3.17, la machine $m = 0$ calculera les quantités $P(n_1|h_t, n_0)$ et $P(n_3|h_t, n_2)$ tandis que la machine $m = 1$ calculera les quantités $P(n_2|h_t, n_1)$ et $P(n_4|h_t, n_1)$.

L'étape de propagation avant est relativement simple. On calcule l'équation 3.6 pour chaque noeud assigné à une machine donnée. À la fin de ces calculs, chaque machine possède les probabilités pour un sous-ensemble des noeuds. Il est finalement nécessaire de communiquer à chacune des machines les probabilités des noeuds dont elle n'a pas effectué les calculs. Dans l'exemple, la machine $m = 0$ communiquera le produit $P(n_1|h_t, n_0)P(n_3|h_t, n_2)$ à la machine $m = 1$ et cette dernière communiquera le produit $P(n_2|h_t, n_1)P(n_4|h_t, n_1)$. En dernier lieu, les probabilités sont multipliée et chaque machine possède finalement la valeur finale de $p(w_i|h_t)$.

L'étape de propagation arrière est plus complexe. Chaque machine calcule les quantités des équations 3.10, 3.11, 3.12 et 3.13 pour chaque noeud n_j qui lui sont assignés. Il est important de souligner que les corrections effectués sur les poids des équations précédentes sont indépendentes du reste du calcul. En pratique, cela signifie que chaque machine possède une partition disjointe de ces poids et qu'il n'est jamais nécessaire de les avoir regroupés sur une seule machine. En d'autres mots, la machine m, sachant l'ensemble des noeuds pour lesquels elle doit faire le calcul (à différentes profondeurs de l'arbre), n'a besoin que d'un sous-ensemble des poids de $\mathbf{W_h}$, $\mathbf{b_h}$ ainsi que des vecteurs caractéristiques de noeuds $\mathbf{n}_j$. De plus, ces poids ne sont utilisés par aucune autre machine.

Le reste du calcul ne peut s'effectuer qu'après avoir communiqué la somme des vecteurs $\mathbf{y}$ des noeuds de chaque machine. C'est-à-dire que chaque machine échange la somme des vecteurs $\mathbf{y}$ qu'elle a calculé. Chaque machine calcule finalement la somme totale, on la dénotera $\mathbf{y}^*$, de tout ces vecteurs. Il est maintenant possible, pour chaque machine, d'effectuer les calculs des équations 3.15 et 3.16[6]. Cependant, le calcul de l'équation 3.14 n'est pas possible étant donné que les quantités $\mathbf{n}_j$ ne sont pas présentent sur toutes les machines. Il est donc nécessaire de communiquer directement les quantités $\frac{\partial E_{NLL}}{\partial \mathbf{W_i}^c}$ de chaque noeud. On procède donc de façon similaire à la méthode employée pour le vecteur $\mathbf{y}$ et on échange les sommes locale de ces quantités entre chacune des machines. En sommant les contributions de chacune des machines dans une matrice $\mathbf{Z}$, on peut finalement calculer l'équation 3.14 ainsi :

$$\left(\frac{\partial E_{NLL}}{\partial \mathbf{W_i}^c}\right) = \mathbf{Z}$$

La rétropropagation de l'erreur nécessite la communication, pour chaque exemple, d'une vecteur $\mathbf{y}$ et d'une matrice $\mathbf{Z}$ de dimensions respectives n_{hidden} et $l(code_noeud) * n_{hidden}$. La

[6]Il faut remarquer que par la définition de $\mathbf{z}$ de la section précédente, $\mathbf{z}$ est bien la somme des $\mathbf{y}$ de chaque noeud, c'est-à-dire $\mathbf{y}^*$.

communication de la matrice $\mathbf{Z}$ est potentiellement coûteuse étant donné sa dimension. La section 3.6 donne une façon de diminuer le temps de communication nécessaire.

En pratique la profondeur maximale de l'arbre sera de l'ordre de quelques dizaines (≈ 40) alors que le nombre de machines disponibles m sera au plus de 8 ou 16. Même dans le cas où l'arbre est parfaitement balancé, il existe cependant un déséquilibre sur la charge de calcul à faire[7] pour chacune des machines tel que défini par l'équation 3.17. En utilisant les valeurs données précédemment (40 comme profondeur maximale et $m = 16$), le nombre de noeuds calculés pour chaque machine est d'au moins $\lfloor 40/16 \rfloor = 2$. Cela signifie que 8 des machines devront faire les calculs pour un noeud de plus que les autres machines. Cependant ce déséquilibre n'est pas trop important en première analyse et est en réalité inférieur à cause des calculs locaux réutilisés[8] et des communications qui sont éliminées par le choix du partage des noeuds.

En pratique, si on utilisait un réseau de neurones *NPLM*, on paralléliserait (voir section 2.3.4) selon le nombre de sorties, étant donné le grand nombre de sorties. Cela signifie que pour chaque exemple, on devrait communiqué un vecteur gradient $\mathbf{y}$ de dimension n_{hidden}. Or, si on fait abstraction de la matrice $\mathbf{Z}$, la méthode présentée dans cette section est aussi efficace que la méthode pour le *NPLM*. La prochaine section montre comment on peut diminuer le côut relié à la communication de $\mathbf{Z}$.

3.6 Approximation

Lorsqu'on utilise l'algorithme sous sa forme parallèle, il n'est pas efficace de communiquer et d'aggréger à chaque itération la partie du gradient qui sert à la correction des poids de ${\mathbf{W}_i}^c$, c'est-à-dire $\mathbf{Z}$. La dimension importante de cette matrice rend prohibitif le temps requis pour la communication. Cependant, le coût computationnel, même en mode séquentiel, peut aussi être relativement important.

En mode parallèle, une façon efficace de procéder est d'accumuler localement les $\mathbf{Z}$ sur chaque machine m pendant un certain nombre d'exemples et de les échanger périodiquement au lieu de le faire pour chaque exemple. Cela permet d'amortir le coût de la communication sur plusieurs exemples. En mode séquentiel, il suffit d'accumuler les gradients $\mathbf{y}$ et d'appliquer la correction sur les poids ${\mathbf{W}_i}^c$ périodiquement. Dans les deux cas cependant, cela introduit une modification potentiellement importante par rapport au fonctionnement habituel de l'algorithme de rétropropagation.

Pour tout les calculs parallel effectués, l'approximation décrite dans cette section à été utilisée en communiquant les matrices $\mathbf{Z}$ à intervals de 4000 exemples. En mode séquentiel, l'approximation consiste à accumuler les gradients de $\mathbf{W}_i$ dans la matrice $\mathbf{Z}$ et d'appliquer les corrections à intervals de 4000 exemples. Comme on le verra au prochain chapitre, l'approximation n'affecte pas à outrance les performances en généralisation de l'algorithme.

---

[7]Le calcul de la probabilité pour chacun des noeuds dans le chemin de l'arbre.

[8]Le calcul de $\mathbf{C}_t^T {\mathbf{W}_i}^*$ peut-être réutilisé tel que mentionné précédemment.

3.7 Mixture

Cette section présente une amélioration à l'algorithme de réseau hiérarchique. La méthode est relativement simple, apporte une amélioration des performances de généralisation appréciable et ne vient pas pénaliser le temps de calcul de l'algorithme original. C'est donc, d'une certaine façon, une amélioration "gratuite".

L'objectif est toujours de modéliser la probabilité $P(w_i|h_t)$. On a déjà présenté quelques méthodes permettant de modéliser cette probabilité. L'idée est ici de *combiner* deux méthodes afin d'améliorer les performances. On combinera deux modèles qui seront, dans le cas présenté ici, le modèle par trigrammes et le modèle par réseau de neurones hiérarchique. L'expression pour la probabilité que l'on désire modéliser devient la suivante :

$$P(w_i|h_t) = \alpha P_{NN}(w_i|h_t) + (1-\alpha)P_T(w_i|h_t)$$

où P_{NN} et P_T réfère aux probabilités telles que modélisées, respectivement, par réseau de neurones hiérarchique et par trigrammes. Pour le modèle présenté ici, on assume que le modèle pour P_T est déjà entraîné (il restera statique) et qu'on désire procéder à l'entraînement du réseau de neurones.

3.7.1 Intégration à l'entraînement

Il est également possible d'utiliser une mixture qui modifie la fonction d'erreur employée lors de l'entraînement. La mixture des deux modèles permettra alors de modifier légèrement l'expression du gradient de l'erreur sur les paramètres du réseau de neurones. Ainsi, l'expression de l'erreur E pour un exemple donné s'exprime maintenant comme suit :

$$E = -log\{\alpha P_{NN}(w_i|h_t) + (1-\alpha)P_T(w_i|h_t)\}$$

Pour faire le pont avec les expressions de la section 2.3.3, on écrira plutôt :

$$E = -log\{\alpha \mathbf{p}_{cible} + (1-\alpha)P_T(w_i|h_t)\}$$

On procédera comme auparavant afin d'obtenir, par dérivation en chaîne, l'expression $\frac{\partial E}{\partial \omega_i}$. Par les équations présentées à la section 2.3.3, on possède déjà les expressions pour $\frac{\partial \mathbf{p}_{cible}}{\partial \omega_i}$. On procède rapidement en développant ainsi :

$$\begin{aligned}\left(\frac{\partial E}{\partial \omega_i}\right) &= \left(\frac{\partial E}{\partial \mathbf{p}_{cible}}\right)\left(\frac{\partial \mathbf{p}_{cible}}{\partial \omega_i}\right) \\ &= \left(\frac{-\alpha}{P(w_i|h_t)}\right)\left(\frac{\partial \mathbf{p}_{cible}}{\partial \omega_i}\right)\end{aligned}$$

Dans l'implémentation de l'algorithme de réseaux de neurones, il ne s'agit que de modifier la quantité du gradient de l'erreur par rapport aux paramètres par le facteur indiqué dans l'équation précédente.

Chapitre 4

Expérimentations et résultats

Ce chapitre présente les différentes expérimentations ayant trait à l'algorithme de *NPLM* hiérarchique présenté au chapitre 3. On présente des comparaisons entre les autres algorithmes comme les *trigrammes* et les autres types de méthodes basées sur les réseaux de neurones. Les résultats comparatifs ont été effectués à deux niveaux. On mesure tout d'abord la performance au niveau du temps de calcul des différents algorithmes. On évalue ensuite la capacité de généralisation des *NPLM* hiérarchiques par rapport aux autres méthodes. C'est-à-dire qu'on évalue comparativement la qualité des solutions obtenues avec les différentes méthodes. Ces deux types de comparaisons permettront de vérifier l'objectif initial de la recherche qui était d'accélérer les méthodes à base de réseau de neurones tout en maintenant les performances de généralisations. Toutes les expériences comparatives utilisent la base de données *Brown*.

On présente également les résultats de l'entraînement de *NPLM* hiérarchiques sur la base de données *APNews*. Cette base de données étant beaucoup plus volumineuse que *Brown* et étant difficile à utiliser avec les autres méthodes à base de réseau de neurones.

La prochaine section fait tout d'abord la description des bases de données utilisées aux cours des expérimentations présentées dans ce chapitre.

4.1 Bases de données

Deux bases de données ont été utilisées au cours de la présente recherche. La première, *Brown*, est un corpus comprenant 1 105 515 mots. Le vocabulaire utilisé a été défini comme étant les 10 000 mots ayant les plus grandes fréquences. Ce corpus a été segmenté en trois sous-ensembles : les ensembles d'entraînement, de validation et de test. Les dimensions respectives de ces sous-ensembles sont présentées au tableau 4.1. Le corpus *Brown* est l'étalon standard généralement utilisé en modélisation du langage pour comparer les différents algorithmes.

La base de données *APNews* est constituée des articles de l'*Associated Press* parus pour l'année 1996. Pour cette raison, on dénote ce corpus par *AP96*. Ici également, on a utilisé comme vocabulaire les 10 000 mots les plus fréquents apparaissant dans le corpus. *AP96*

Ensemble	Nombre de mots
Entraînement	900 000
Validation	100 000
Test	105 515
Total	1 105 515

TAB. 4.1 – Dimensions des ensembles de données pour la base de données *Brown*.

est constituée de 2 220 177 mots. Ce corpus a également été divisé pour constituer les trois sous-ensembles habituels. Le tableau 4.2 présente les dimensions de ces ensembles.

Ensemble	Nombre de mots
Entraînement	1 776 141
Validation	222 017
Test	222 019
Total	2 220 177

TAB. 4.2 – Dimensions des ensembles de données pour la base de données *AP96*.

4.2 Nomenclature

On réfère aux différents algorithmes par les étiquettes suivantes :

NPLM	Algorithme original à base de réseau de neurones
ContDivNPLM	Algorithme *NPLM* avec échantillonnage
SymbCondHier	Algorithme *NPLM* hiérarchique
SymbCondHier-approx	Algorithme *NPLM* hiérarchique utilisant l'approximation de la section 3.6.
SymbCondHier-mix	Algorithme *NPLM* hiérarchique avec mixture *trigramme*

Pour la version parallèle de l'algorithme *SymbCondHier* on utilise l'approximation de la section 3.6, c'est-à-dire *SymbCondHier-approx*. Cela fera en sorte que les résultats de la version parallèle pour un seul processeur seront plus rapide que ceux de la version séquentielle. Pour l'algorithme *SymbCondHier-mix* on utilise la mixture *simple* avec une valeur de 0.72 pour α[1].

[1]à moins qu'il en soit mentionné autrement

4.3 Résultats comparatifs

Cette section présente les résultats comparatifs entre les différents algorithmes de modélisation du langage. Pour procéder à cette comparaison, on a utilisé la base de données *Brown* sur un vocabulaire de 10 000 mots. Les prochaines sections présentent les différents points de comparaisons entre les algorithmes.

4.3.1 Performance computationnelle

Cette section compare les temps de calcul des différents algorithmes à base de réseau de neurones. On utilise la même base de données (*Brown*) avec la même dimension de vocabulaire ($|\mathcal{V}| = 10\,000$ mots) pour tous les algorithmes. On compare les temps pour l'*apprentissage* (c'est-à-dire l'entraînement) comme tel mais également pour la mesure de performance (l'opération de *test*) sur l'ensemble de validation et de test. Les résultats sont présentés selon deux mesures : le temps pour une époque[2] ainsi que le temps sur un seul exemple.

Comme on le mentionnera à la section 4.3.3, les résultats présentés ne sont pas immédiatement indicatifs de l'efficacité. Par exemple, il est fort possible qu'un algorithme prenne moins de temps pour apprendre pendant une époque mais qu'il prenne plus de temps au *total* pour arriver aux mêmes performances de généralisation. Cependant, ces résultats fournissent une caractérisation des différents réseaux qui sera réutilisée plus tard.

Type de réseau	Temps par époque (s)	Temps par exemple (ms)	Speedup
NPLM	416 300	462.607	1.00
ContDivNPLM	6 062	6.735	68.69
SymbCondHier	1 609	1.788	258.73
SymbCondHier-approx	1 270	1.412	327.63

TAB. 4.3 – Temps d'entraînement pour les algorithmes à base de réseau de neurones

Comme le montre le tableau 4.3, l'accélération du temps d'entraînement (pour un exemple) de l'algorithme hiérarchique (*SymbCondHier*) est substantiel. L'algorithme *ContDivNPLM* performe également très bien. Ainsi, la stratégie du *ContDivNPLM* de calculer seulement un sous-ensemble des sorties (de l'ordre de 100) afin d'évaluer le gradient permet des économies de temps substantielles. Cependant, même comparativement à l'algorithme *ContDivNPLM*, le *SymbCondHier* est 3.7 fois plus rapide et le *SymbCondHier-approx* 4.7 fois. Les résultats viennent en outre confirmer le temps de calcul nécessaire à la correction des poids de la matrice $\mathbf{W_i}^c$ de l'algorithme *SymbCondHier* et de l'économie apportée par l'approximation *SymbCondHier-approx*.

[2]L'application de la règle de rétropropagation sur tous les exemples de l'ensemble d'entraînement.

Le tableau 4.4 présente les temps d'entraînement pour la version parallèle de l'algorithme : *SymbCondHier-approx*. Les résultats montrent à quel point la parallélisation est efficace. Il faut mentionner que la parallélisation de *NPLM* est à toutes fins impraticable. En effet, pour plus de deux processeurs, le temps de communication devient trop élevé et aucune économie de temps n'est possible[3]. Le *SymbCondHier* permet donc d'utiliser un parc de machines de façon efficace, ce qui n'était auparavant pas possible avec l'algorithme *NPLM*.

Nombre de processeurs	Temps par époque (s)	Temps par exemple (ms)	Speedup
1	1 270	1.412	1.00
2	715	0.794	1.78
4	461	0.512	2.76
8	362	0.402	3.51

TAB. 4.4 – Temps d'entraînement pour l'algorithme parallélisé *SymbCondHier-approx*

Le tableau 4.5 présente la comparaison des temps de test pour les différents algorithmes à base de réseau de neurones. Le temps de test est le temps nécessaire pour calculer la fonction de coût sur un ensemble de données. Contrairement à l'entraînement, le calcul pour l'opération de test peut se faire de façon totalement indépendante pour chaque exemple.

	Type de réseau			
Nombre de processeurs	*NPLM*	*ContDivNPLM*	*SymbCondHier*	*SymbCondHier-approx*
1	270.702 (1.00)	221.297 (1.22)	1.438 (188.25)	0.786 (344.40)
2	271.270 (0.99)	111.568 (2.43)	0.806 (335.86)	0.357 (758.27)
4	86.199 (3.14)	55.978 (4.84)	-	0.179 (1512.30)
8	38.741 (6.99)	28.473 (9.51)	0.189 (1432.29)	0.103 (2628.17)

TAB. 4.5 – Temps de test (ms) et (speedup) par exemple pour les algorithmes à base de réseau de neurones

Les résultats montrent que l'accélération pour le *SymbCondHier* est encore très importante, près de 200 fois plus rapide que l'algorithme *NPLM* et près de 350 fois plus rapide pour le *SymbCondHier-approx*. Même comparé à l'algorithme *ContDivNPLM*, l'accélération du *SymbCondHier* est plus de 150 fois plus rapide sur un seul processeur et celle du *SymbCondHier-approx* plus de 280 fois plus rapide. Ce gain par rapport aux temps de test de ces deux algorithmes est phénoménal et s'explique principalement par le fait que le calcul est particulièrement économique dans le cas du *SymbCondHier* (et du *SymbCondHier-approx*). En effet, la fonction de coût employée, ne nécessitant que la valeur $\mathbf{p}_{cible}$ pour chaque exemple,

[3]La version parallèle du NPLM est cependant relativement performante sur des machines à mémoire partagée

permet de limiter le calcul à la seule sortie nécessaire, c'est-à-dire au seul(s) chemin(s) de l'arbre menant au mot cible. Cette économie n'est pas possible pour les autres algorithmes qui nécessitent le calcul de toutes les sorties.

4.3.2 Généralisation

La précédente section à présenté le temps de calcul des algorithmes sur la base du temps de calcul par exemple (ou par époque). Cette section présente les résultats comparatifs dans le cadre de l'erreur de généralisation, c'est-à-dire les mesures de performances sur l'ensemble de test. *A priori* ce sont ces résultats qui sont d'intérêt premier puisqu'ils permettent de déterminer quels sont les algorithmes qui modélisent le mieux l'espace de probabilités que l'on cherche à approximer.

Algorithme	Validation	Test	Nombre d'époques
Trigramme	299.463	268.753	-
NPLM	213.159	195.257	18
ContDivNPLM	209.439	192.571	15
SymbCondHier	241.636	220.767	51
SymbCondHier-approx	241.488	220.960	42
SymbCondHier-mix	221.218	201.916	38

TAB. 4.6 – Erreur de généralisation (NLL) sur les ensembles de validation et de test pour $|\mathcal{V}| = 10000$

Le tableau 4.6 montre en premier lieu à quel point les méthodes à base de réseaux de neurones sont supérieures à l'algorithme par trigramme. Les algorithmes *NPLM* et *ContDivNPLM* sont les plus performants. L'algorithme *SymbCondHier* se compare tout de même bien par rapport à ces deux méthodes. En outre, on remarque que l'approximation *SymbCondHier-approx* performe aussi bien que la version originale. Finalement, les résultats pour le réseau *SymbCondHier-mix* montre comment il est possible d'améliorer légèrement les performances par l'utilisation d'une mixture. Il est important de noter, cependant, que l'utilisation d'une mixture est aussi possible pour les réseaux *NPLM* et *ContDivNPLM*.

4.3.3 Efficacité

Dans cette section, on termine la comparaison entre les différents algorithmes à base de réseau de neurones. On regarde les performances sur l'ensemble de test en fonction du temps d'entraînement nécessaire pour atteindre un certain niveau. En d'autres mots, on compare les *vitesses d'apprentissages*, c'est-à-dire les performances de test en fonction du temps d'apprentissage.

Le tableau 4.7 montre les temps de calculs pour atteindre la convergence des algorithmes. Ces résultats montre l'efficacité des algorithmes *ContDivNPLM* et *SymbCondHier* (et ses

dérivés). Cependant, compte tenu des résultats de la dernière section sur la capacité de généralisation de *SymbCondHier-approx*, cet algorithme est particulièrement avantageux, étant non seulement le plus rapide mais d'un facteur de 1.7 par rapport à *ContDivNPLM*. L'algorithme *SymbCondHier-mix* est pratiquement aussi rapide que *SymbCondHier*, justifiant en pratique l'utilisation d'une mixture avec trigramme. Il est important de mentionner que la mixture de *SymbCondHier-mix* n'utilise pas un *SymbCondHier-approx*. Il est donc possible de profiter de l'amélioration en généralisation de la mixture et de profiter du gain en vitesse de l'approximation.

Type de réseau	Erreur de test	Nombre d'époques	Temps total (m)	Speedup
NPLM	196.0393	16	97 422.65	1.00
ContDivNPLM	192.571	15	1 515.50	62.28
SymbCondHier	220.767	51	1 367.66	71.23
SymbCondHier-approx	220.960	42	890.06	109.45
SymbCondHier-mix	201.916	38	1 427.77	68.23

TAB. 4.7 – Temps de calcul afin d'atteindre l'optimal pour les algorithmes à base de réseau de neurones

Le tableau 4.8 montre tout d'abord que l'algorithme parallélisé est particulièrement stable, atteignant virtuellement le même point de convergence, indépendemment du nombre de processeurs utilisés. Il est donc envisageable de minimiser le temps total de calcul (d'entraîner le plus rapidement possible) au dépend de l'efficacité et de maintenir les mêmes performances que l'algorithme séquentiel.

Nombre de processeurs	Erreur de test	Nombre d'époques	Temps total (m)	Speedup
1	220.960	42	890.06	1.00
2	219.727	42	500.91	1.78
4	219.727	42	325.37	2.74
8	219.727	42	257.20	3.46

TAB. 4.8 – Temps de calcul afin d'atteindre l'optimal pour l'algorithme *SymbCondHier-approx* parallèle

La figure 4.1 compare l'évolution de l'erreur de test en fonction du temps pour les algorithmes *ContDivNPLM* et *SymbCondHier*. On voit ainsi la vitesse d'apprentissage supérieure des modèles *SymbCondHier* par rapport au *ContDivNPLM*. Les courbes *brown_symbcondhier_efficiency* et *brown_symbcondhier_1procs_efficiency* montre en outre la différence de vitesse entre, respectivement, *SymbCondHier* et *SymbCondHier-approx*.

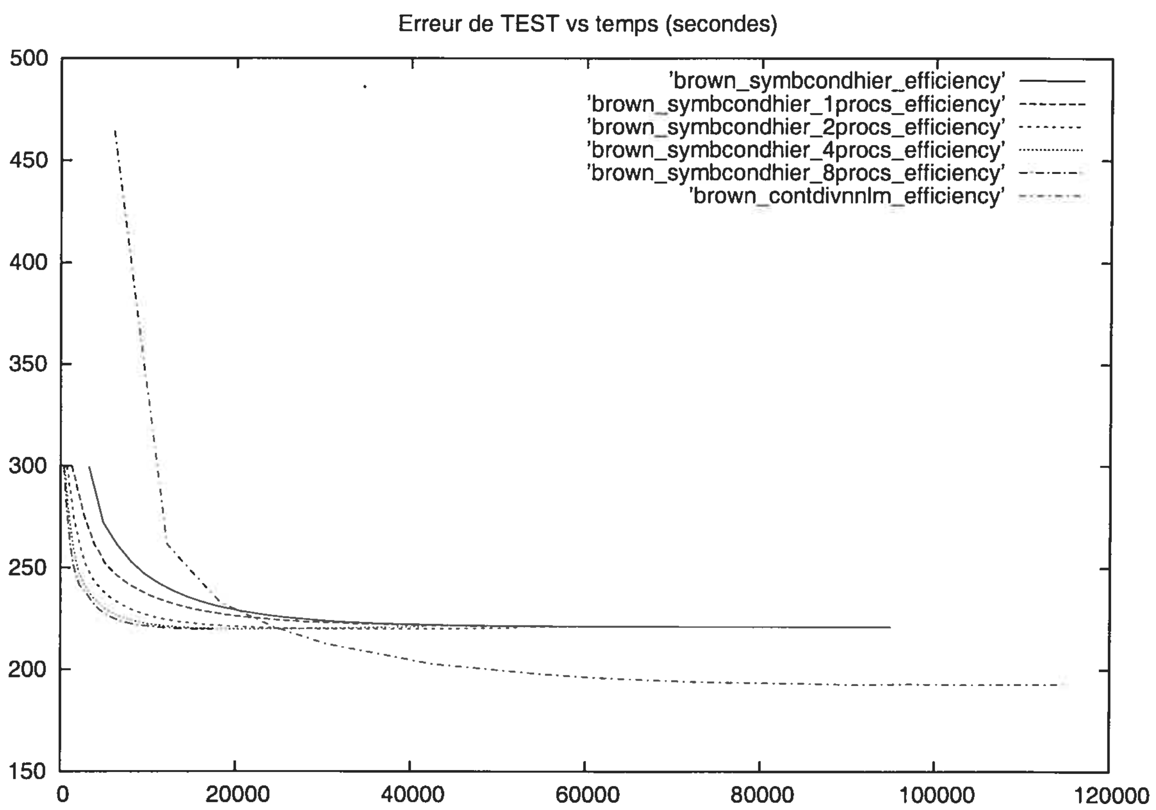

FIG. 4.1 – Comparaisons de l'erreur (NLL) en fonction du temps d'entraînement (s).

4.4 Résultats *APNews*

Cette section présente les résultats obtenus par l'algorithme de *NPLM* hiérarchique sur la base de données *APNews*. Pour l'algorithme *SymbCondHier-mix*, une valeur de 0.5 à été utilisée pour α.

Algorithme	Validation	Test	Nombre d'époques
Trigramme	50.151	47.031	-
SymbCondHier	60.060	56.797	33
SymbCondHier-mix	48.208	46.081	82

TAB. 4.9 – Erreur de généralisation sur les ensembles de validation et de test pour $|\mathcal{V}| = 10000$

Le tableau 4.9 présente les résultats pour la base de données *APNews*. Les performances de l'algorithme *NPLM* hiérarchique sont décevants. En effet, la perplexité est supérieure à celle obtenue pour le modèle trigramme. Compte tenu du coût additionnel en temps de calcul de l'algorithme *NPLM* hiérarchique, il est difficile de justifier l'utilisation d'une méthode à base de réseaux de neurones comme le *NPLM* hiérarchique. Cependant, l'algorithme *NPLM-mix* permet d'obtenir une performance légèrement meilleure que le trigramme. Il faut noter que cet algorithme est une mixture utilisant le modèle trigramme. Cela indique comment l'utilisation de deux modèles différents permet d'obtenir de meilleurs résultats combinés.

4.5 Discussion

Les résultats des sections précédentes montrent tout d'abord comment les méthodes à base de réseaux de neurones sont généralement supérieures à la méthode classique de trigramme. Cependant la performance accrue se fait au dépend du temps de calcul nécessaire pour obtenir une bonne solution. Le modèle originel *NPLM* de la section 2.3.3 souffre particulièrement de cette gourmandise computationnelle. Le modèle *ContDivNPLM* de la section 2.3.5 et le nouveau modèle présenté au chapitre 3 ont été proposé pour résoudre ce problème computationnel. Les résultats des sections précédentes montrent dans quelle mesure ils réussissent.

Le modèle de *NPLM* par échantillonnage est particulièrement performant en termes de généralisation et de rapidité de calcul. Cependant, ce réseau ne résoud qu'une partie du problème computationnel, c'est-à-dire l'entraînement. L'étape de test est presqu'aussi lente que pour le modèle *NPLM*, comme le montre les résultats du tableau 4.5. Le modèle *SymbCondHier* est le plus rapide de toutes les méthodes à base de neurones. Il n'est cependant pas le plus performant, mais les performances sont raisonnables. De plus, ce modèle vient résoudre le problème du temps de calcul de l'étape de test.

Il est important de souligner que les résultats présentés ont été obtenus sur un vocabulaire de dimension $|\mathcal{V}| = 10000$. On considère généralement des vocabulaires de plus grande taille,

typiquement 20000, 30000 ou même 50000. Le modèle *NPLM* est inutilisable dans un tel contexte. Le modèle *NPLM* par échantillonnage sera probablement beaucoup plus rapide que le *NPLM* pour l'entraînement. L'avantage computationnel du *NPLM* hiérarchique vient du fait qu'il exploite une décomposition hiérarchique. Cela permet d'augmenter le nombre de sorties tout en ayant une augmentation du temps de calcul sub-linéaire.

En théorie, l'algorithme *NPLM* hiérarchique se comporte bien, au point de vue du temps de calcul, pour de larges vocabulaires, $|\mathcal{V}| > 50000$. Tout d'abord, on sait que le nombre de noeuds de l'arbre binaire augmente linéairement en fonction du nombre de sorties ($= |\mathcal{V}|$), mais que la profondeur seulemenent logarithmiquement. À titre d'exemple, pour un arbre balancé de $n = 10000$ feuilles, on a un total de $2n - 1 = 19999$ noeuds et une profondeur moyenne de $log_2(10000) \approx 13$. Pour un vocabulaire de 50000 mots, on a soudainement un total de 99999 noeuds et une profondeur moyenne de ≈ 16. La profondeur de l'arbre est indicatrice du temps de calcul nécessaire lors de l'entraînement puisqu'il vaut faire les calculs pour les noeuds dans le chemin menant à un mot. Pour un vocabulaire de 50000 mots, on donc peut s'attendre à une augmentation du temps de calcul d'un facteur inférieur à $16/13 \approx 1.23$, ce qui est exceptionnel.

4.5.1 Directions futures

Cette section explore différentes possibilités qui pourraient permettrent d'améliorer le modèle de *NPLM* hiérarchique.

Espace caractéristique des noeuds

Comme l'on montré les résultats, l'approximation du *NPLM* hiérarchique est aussi performante en généralisation que la version complète. De plus, les noeuds sont indépendents l'un de l'autre et l'utilisation d'une matrice $\mathbf{W_i}^c$ partagée par tout les noeuds ne semble pas justifiée. Il semble donc possible d'éliminer complètement cette matrice et les vecteurs caractéristiques de noeuds et de les remplacer par des vecteurs de préactivation pour chaque noeud. Ainsi, au lieu d'apprendre les poids des vecteurs caractéristiques de noeuds et de la matrice $\mathbf{W_i}^c$, on apprendrait des biais sur les vecteurs de preactivation. Cette modification permet de calculer la préactivation ainsi :

$$\mathbf{s} = \mathbf{C}_t^T \mathbf{W_i}^* + \mathbf{b_i} + \mathbf{c}_{parent(n_j)}$$

où $\mathbf{c}_{parent(n_j)}$ est le vecteur associé au noeud $parent(n_j)$. Cela permet, autant pour l'algorithme séquentiel que pour sa version parallèle, de réduire le temps nécessaire pour faire les calculs de la propagation avant ainsi que le temps pour la propagation arrière.

Reconnaissance

L'algorithme de *NPLM* hiérarchique est performant pour l'entraînement ainsi que pour l'opération de test. Cependant, l'utilisation du réseau afin de prédire des probabilités n'a pas été discutée. Pour les autres méthodes à base de réseaux de neurones, l'opération de test est

identique à l'opération de reconnaissance. Cela est du au fait qu'on doit faire les calculs pour toutes les sorties. L'opération de test du *NPLM* hiérarchique prend avantage du fait que la fonction de coût (la perplexité) ne dépend que d'une seule sortie pour minimiser le calcul à faire. Cependant, lorsqu'il faut *prédire* les sorties ou lorsqu'il faut trouver la meilleure sortie, il n'est pas efficace de calculer toutes les sorties. La raison est que pour n sorties, il y a $2n-1$ noeuds pour lesquels il faut faire le calcul. En pratique, on désire rarement obtenir des valeurs pour toutes les sorties. Par exemple, on peut être intéressé à obtenir les N meilleures sorties ou simplement obtenir les sorties ayant une probabilité plus grande qu'un seuil donné.

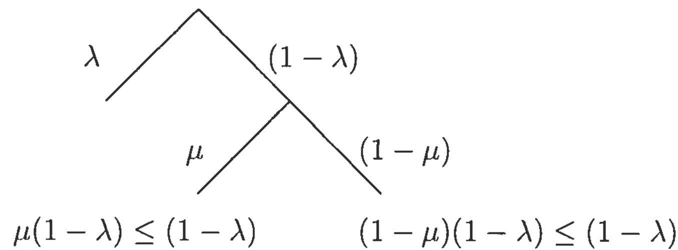

FIG. 4.2 – Propriétés des noeuds

Lorsqu'on ne désire pas calculer toutes les sorties, la décomposition hiérarchique permet de calculer les meilleures sorties de façon relativement efficace. En effet, le réseau hiérarchique possède deux propriétés essentielles. La première indique que, pour un noeud donné, la probabilité de ses enfants somme à 1. La seconde indique que la somme de toutes les probabilités des noeuds feuilles somme à 1. Cela permet d'établir un plafond sur la valeur maximale des probabilités des noeuds feuilles d'un sous-arbre.

Par exemple, pour un noeud donné n_j, si on a $p(n_j|h_t, parent(n_j)) = \lambda$, on sait que la probabilité du noeud frère de n_j à une probabilité $1-\lambda$. De plus, on sait que la probabilité de tous les noeuds feuilles descendants de n_j à une probabilité inférieure ou égale à λ. La figure 4.2 illustre les propriétés associées au calcul des noeuds de l'arbre.

Si on veut obtenir les valeurs des N meilleures sorties, il est donc possible de faire les calculs pour une petite partie de tout les noeuds. Il s'agit d'un algorithme gourmand qui explore l'arbre en favorisant les branches les plus prometteuses. L'algorithme 2 présente les étapes pour calculer de façon efficace les N meilleures sorties.

1 **begin**
2 Créer une liste ordonnée vide $L_{chemins}$, contenant des couples $(probabilit\acute{e}, noeud)$.
3 Créer une liste ordonnée vide $L_{feuilles}$, contenant des couples $(probabilit\acute{e}, noeud)$.
6 **tant que** $(|L_{feuilles}| < N)$ **et** $(L_{feuilles}[N].probabilit\acute{e} < L_{chemins}[1].probabilit\acute{e})$
7 $prob_{parent} = L_{chemins}[1]$.probabilité
8 $noeud_{parent} = L_{chemins}[1].noeud$.
9 Retirer le premier élément de $L_{chemins}$
10 Calculer les probabilités $prob_i$ pour les noeuds enfants de $noeud_{parent}$
11 **pour chaque** noeud enfant $noeud_i$ de $L_{chemins}[1].noeud$
12 **si** $noeud_i$ est une feuille
13 insérer $(prob_{parent} * prob_i, noeud_i)$ dans $L_{feuilles}$
14 **sinon**
15 insérer $(prob_{parent} * prob_i, noeud_i)$ dans $L_{chemins}$
16 Retourner les N premières valeures de $L_{feuilles}$.
17 **end**

Algorithme 2 : Calcul des N meilleures sorties

Sens multiples

La base de données *WordNet*, dont on se sert pour la clusterisation, offre plus d'un sens pour la plupart des mots du vocabulaire. Cependant, pour l'algorithme de clusterisation qui a été employé on s'est limité à utiliser le sens le plus fréquent de chaque mot. Cela fait en sorte que, lors de l'entraînement, la correction produite pour un mot w^4 pour lequel le sens n'est pas celui qui a été utilisé pour la clusterisation ne produira pas l'effet voulu. À titre d'illustration, soit les mots $\{w_1, w_2, w_3, w_4\}$ constituant un exemple de l'ensemble d'entraînement, les trois premiers mots constituant l'entrée et le dernier mot constituant le mot désiré.

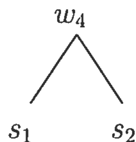

FIG. 4.3 – Senses de w_4

Soit s_1 et s_2, deux sens associés au mot w_4. Supposons que s_2 est le sens du mot dans le contexte de l'exemple et que s_1 est le sens utilisé lors de la construction de l'arbre binaire. Dans les mots du vocabulaire, il existe un certain nombre de mots qui ont respectivement des sens proches de s_1 et de s_2. Il y a ainsi, conceptuellement, un noeud dans l'arbre permettant,

[4] De façon plus précise : le gradient de correction des poids étant donné que le mot désiré est w.

lors de la présentation d'un exemple donné, de prendre une décision et de choisir lequel des deux sens est le plus probable. Lors de l'application de l'algorithme de rétropropagation, il y aura modification des poids faisant en sorte d'augmenter la probabilité du mot w_4 de sens s_1. Ainsi, les mots de sens proche à s_1 verront aussi leur probabilité augmentée alors que les mots de sens s_2 subiront l'effet contraire. En incorporant plus d'un sens par mot, il serait en théorie possible d'alléger ce problème de la façon suivante. La correction des poids fera toujours en sorte d'augmenter la probabilité du mot w_4 de sens s_1 (et les mots de sens proches à s_1) mais *également* du mot w_4 de sens s_2 (et les mots de sens proches à s_2). Cela aura pour conséquence de réduire la correction aux mots de sens proches à s_1 au profit des mots de sens proches à s_2.

L'algorithme *NPLM* hiérarchique présenté au chapitre 3 possède déjà la possibilité d'utiliser plus d'un sens par mot. Quelques expériences préliminaires ont cependant indiqué que les gains sur l'erreur de généralisation obtenus en utilisant plusieurs sens par mots sont au mieux marginals. Une raison qui explique cette situation est que l'entrée du réseau de neurones n'indique pas les sens des mots du contexte. En utilisant de l'information sur les sens des mots il serait possible de considérer chaque sens d'un mot comme des sorties à part entière et de prédire la probabilité des sens d'un mot au lieu de la probabilité du mot comme tel. Cependent cela nécessite des bases de données contenant cette information sur les sens, chose qui n'est pas habituellement disponible.

Capacité supplémentaire

Le calcul de la probabilité $P(n_j|h_t, parent(n_j))$ se base exclusivement sur les quantités $\mathbf{C}_t$ (construit à partir de h_t) et $\mathbf{n}_j$. Or, les probabilités des noeuds menant à un mot dépendent toutes de h_t de la même façon, c'est-à-dire par la quantité $\mathbf{C}_t^T\mathbf{W_i}^* + \mathbf{b_i}$. Il est possible que le partage de l'information provenant du contexte soit sub-optimal pour le calcul des probabilités de tout les noeuds.

Il est possible d'utiliser une matrice supplémentaire pour chaque noeud afin de dépendre de h_t de façon plus directe. Pour fin de simplification, on se base sur la modification du modèle proposée précédemment ou on n'utilise plus la matrice $\mathbf{W_i}^c$. Soit donc $\mathbf{W_i}^{parent(n_j)}$ la matrice de dimension $n_{inputs} * K$ où K est une quantité que l'on choisit. Alors, le calcul de la préactivation pour le noeud n_j se fait comme suit :

$$\begin{aligned}\mathbf{s} &= \mathbf{C}_t^T\mathbf{W_i} + \mathbf{b_i} + \mathbf{c}_{parent(n_j)} \\ &= \mathbf{C}_t^T\left[\mathbf{W_i}^* \ \mathbf{W_i}^{parent(n_j)}\right] + \mathbf{b_i} + \mathbf{c}_{parent(n_j)} \\ &= \left(\mathbf{C}_t^T\mathbf{W_i}^* \oplus \mathbf{C}_t^T\mathbf{W_i}^{parent(n_j)}\right) + \mathbf{b_i} + \mathbf{c}_{parent(n_j)}\end{aligned}$$

Il est important de noter que les dimensions pour les vecteurs $\mathbf{b_i}$ et $\mathbf{c}_{parent(n_j)}$ doivent être modifiées en tenant compte de la nouvelle matrice. En conséquences, leurs dimensions sont augmentées de K par rapport à ce qu'elles étaient auparavant.

Comme auparavant, la quantité $\mathbf{C}_t^T\mathbf{W_i}^*$ n'est calculée qu'une seule fois pour chaque exemple. Cependant, on doit calculer le produit $\mathbf{C}_t^T\mathbf{W_i}^{parent(n_j)}$ pour chaque noeud.

Conclusion

L'algorithme de *NPLM* hiérarchique permet d'accélérer le temps de calcul par rapport aux autres méthodes basées sur les réseaux de neurones. Cependant, même si les solutions produites sont excellentes, elle ne sont pas aussi performantes que celles des deux autres méthodes à base de réseaux de neurones.

Les gains énormes en performances offert par l'algorithme permettent également d'attaquer des problèmes de modélisation du langage de plus grande taille, comme la base de données *APNews*. Cependant, les résultats obtenus sur cette base ne sont pas significativement meilleurs que ceux obtenus par de simple *trigrammes*.

Annexe A

Produits vecteur-matrice

Cette section donne rapidement la définition du produit d'un vecteur et d'une matrice (ainsi que le produit d'une matrice par un vecteur). Il s'agit d'une version restreinte du produit "par blocs" de deux matrices. Cette version est suffisante pour les besoins de ce document.

Soit un vecteur $\mathbf{v}$ de dimension L et soit une matrice $\mathbf{M}$ de dimension $L * W$. On définit le produit $\mathbf{u}$ de la transposée de $\mathbf{v}$ par $\mathbf{M}$, écrit $\mathbf{v}^T\mathbf{M}$, de la façon suivante :

$$\mathbf{u} = \mathbf{v}^T\mathbf{M}$$

$$\mathbf{u}_k = \sum_{l=1}^{L} \mathbf{v}_l \mathbf{M}_{l,k}, \qquad k = 1, \ldots, W$$

où $\mathbf{u}$ est de dimension W. De même, on peut définir le produit d'une matrice $\mathbf{M}^T$ de dimension $W * L$ et d'un vecteur $\mathbf{v}$ de dimension L. Cependant, sachant la règle de transposition qui suit, on n'étudiera pas plus ce cas :

$$(\mathbf{M}^T\mathbf{v})^T = \mathbf{v}^T\mathbf{M}$$

Soit toujours le vecteur $\mathbf{v}$ et $\mathbf{M}$, de mêmes dimensions que précédemment. Soit $\mathbf{M_1}$ et $\mathbf{M_2}$, deux sous-matrices de $\mathbf{M}$ de dimensions respectives $L * W_1$ et $L * W_2$ où $W_1 + W_2 = W$.

$$\mathbf{M} = [\mathbf{M_1}|\mathbf{M_2}]$$

Le résultat du produit de la transposée de $\mathbf{v}$ et $\mathbf{M}$ en termes de $\mathbf{M_1}$ et $\mathbf{M_2}$ s'écrit ainsi :

$$\mathbf{u} = \mathbf{v}^T\mathbf{M_1} + \mathbf{v}^T\mathbf{M_2} \tag{A.1}$$

où l'addition est celle de deux vecteurs de dimensions W.

Il est possible de décomposer $\mathbf{M}$ en un plus grand nombre de matrices, de dimensions $L * W_i$ où $\sum_i W_i = W$. On obtient ainsi :

$$\mathbf{u} = \sum_i \mathbf{v}^T\mathbf{M_i} \tag{A.2}$$

Ce résultat s'obtient aisément en utilisant itérativement l'équation A.1.

Bibliographie

[1] BENGIO, Y., *New Distributed Probabilistic Language Models*, Rapport Technique 1215, Département d'informatique et recherche opérationnelle, 2002

[2] BENGIO, Y., DUCHARME, P., VINCENT, P. ET JAUVIN, C., *A Neural Probabilistic Language Model*, Journal of Machine Learning Research 3, 2003

[3] BENGIO, Y. ET SENÉCAL, J.-S., *Quick Training of Probabilistic Neural Nets by Sampling* Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics, Key West, Florida, 2003

[4] SENÉCAL, J.-S. *Accélérer l'entraînement d'un modèle non-paramétrique de densité non normalisée par échantillonnage aléatoire* Mémoire de maîtrise, Département d'Informatique et de Recherche Opérationnelle (DIRO), Université de Montréal, 2003

[5] BISHOP, C. M., *Neural Networks for Pattern Recognition*, Oxford Univ. Press, 1995

[6] Duda, E. O., Hart, P. E., Stork, D. G. *Pattern Classification*, Wiley-Interscience Publication, 2000

[7] FELLBAUM, C. *WordNet : An Electronic Lexical Database*, MIT Press, 1998

[8] GOOD, I.J., *The population frequencies of species and the estimation of population parameters.* Biometrika, 40, pp. 237-264, 1953

[9] HAYKIN, S., *Neural networks : a comprehensive foundation. 2nd ed.*, Prentice-Hall, 1999

[10] HINTON, G., *Learning Distributed Representations of Concepts*, Proceedings of the Eighth Annual Conference of the Cognitive Science Society, Amberst, Lawrence Erlbaum, Hillsdale, p. 1-12, 1986

[11] KATZ, SLAVA M., *Estimation of probabilities from sparse data for the language model component of a speech recognizer.* IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-35, pp. 400-401, March 1987.

[12] KONG, A., *A Note on Importance Sampling using Standardized Weights*, Rapport technique 348, Department of Statistics, University of Chicago, 1992

[13] LANK, K., *NewsWeeder : Learning to Filter News*, Proceedings of the International Conference on Machine Learning (ICML-95), pp.331-339

[14] LIU, J. S., *Monte Carlo Strategies in Scientific Computing*, Springer, 2001

[15] MANNING, C.D. ET SCHUTZE, *Foundations of Statistical Natural Language Processing*, MIT Press, 1999

[16] ROBERT, S.P. ET CASELLA, G., *Monte Carlo Statistical Methods*, Springer-Verlag, 1999

[17] PRESS, W.H., TEUKOLSKY, S.A., VETTERLING, W.T. ET FLANNERY, B.P., *Numerical Recipes in C, Second edition*, Cambridge University Press, 1995

[18] SALTON, G. ET BUCKLEY, C., *Term-Weighting Approaches in Automatic Text Retrieval*, Information Processing and Management : An International Journal, v. 24, n. 5, pp. 513-523, 1988

[19] SALTON, G. *Automatic Text Processing : The Transformation, Analysis, and Retrieval of Information by Computer*, Addison-Wesley, 1989

[20] SHANNON, C. E., *A Mathematical Theory of Communication*, Bell Systems Technical Journal 27, pp. 379-423, 1948