RÉPUBLIQUE DU CAMEROUN
****
Paix - Travail – Patrie
****
UNIVERSITE DE YAOUNDE I
****
Faculté des Sciences
****
Département d'Informatique
****
BP: 812 Yaoundé-Cameroun

REPUBLIC OF CAMEROON
****
Peace - Work – Fatherland
****
UNIVERSITY OF YAOUNDE I
****
Faculty of Science
****
Department of Computer Science
****
P.O. Box: 812 Yaounde-Cameroon

PARALLÉLISATION DES RÉSEAUX DE NEURONES RÉCURRENTS POUR LA DÉTECTION DES MESSAGES HAINEUX

Mémoire présenté en vue de l'obtention du
Diplôme de Master Recherche en Informatique

Présenté et soutenu par:

ONANA Damase Donald
Matricule: 17R2138

Sous la direction de :

Dr MESSI NGUELE Thomas
Chargé de cours (UY1)

Année Académique 2022/2023

DÉDICACE

À ma maman ANABA Micheline Judith

REMERCIEMENTS

Je tiens sincèrement à remercier les membres du jury qui me font le grand honneur d'évaluer ce travail.

J'exprime ma profonde gratitude à mon encadrant actuel Docteur **Thomas MESSI NGUELE** qui en plus de son apport académique, m'a enseigné des leçons de vie et de milieu professionnel.

Je tiens à témoigner toute ma reconnaissance à mon encadrant sortant Professeur **Yves EMVUDU** de regrettée mémoire qui, plus qu'un encadrant à été comme un père pour moi.

Je voudrais remercier mes aînés académiques à l'instar de Regis MOGO, Ghislain WABO, Hermann NITCHEU, Nel NANVOU et très particulièrement Audrey DONGMO pour tous les encouragements et les conseils qu'ils m'ont donné.

Je tiens également à dire merci à mes camarades de promotion à l'instar de Ivan BOUH, Ghislaine GEUBOU, Godlove MUGHE, Ahmed NSANGOU, Desire OLAMA, Brenda MASSO, Audrey FONGUE, Ange HEINTCHEU, Leandra MAKAMTE, Francis NOAH, Beril EKWELE, Belle FOTSO pour tous les encouragements et les conseils qu'ils m'ont donné, ainsi que pour les nombreuses discussions parfois au-delà de la recherche, mais toujours enrichissantes.

Merci à **l'Université de Yaoundé 1** notamment au **département d'informatique** pour la qualité d'enseignement durant mon parcours.

Je remercie également tous les **enseignants du département informatique** pour leur disponibilité et leur conseil tout au long de ma formation depuis mon cycle licence.

Merci également à l'équipe **UMMISCO** pour ce confortable cadre de travail mis à notre disposition.

Merci à mes parents ONANA Jean Damase et ANABA Micheline Judith, à mes frères ESSOMBA ONANA Bruno Franklin et ONANA ONANA Joseph Junior à mes soeurs ONANA Marie Thérèse et BEDIZSSA ONANA Laurentia qui m'ont toujours soutenu, encouragé, éduqué et veillé sur mon bien-être depuis ma naissance jusqu'à aujourd'hui.

Merci à tous les intervenants et toutes les personnes qui par leurs paroles, leurs écrits, leurs conseils et leurs critiques ont guidé mes réflexions et ont accepté de me rencontrer et de répondre à mes questions durant mes recherches.

RÉSUMÉ

Les libertés d'expressions et d'opinions qu'apportent les réseaux sociaux sont très souvent des facteurs favorisant la diffusion des messages haineux. Les discours haineux constituent une menace pour la stabilité sociale et la paix car ils stimulent l'incitation à la discrimination, ce que le droit international interdit. Pour limiter les effets néfastes de ce fléau, l'on intègre très souvent aux plates-formes de réseaux sociaux des modèles fournis par des algorithmes de deep learning leur permettant de détecter et de réagir automatiquement à un message à caractère haineux. Une des particularités de ces algorithmes est qu'ils sont d'autant plus performant que la quantité de données utilisées est grande. Toutefois, l'exécution séquentielle de ces algorithmes sur des grandes quantités de données peut prendre un temps très élevé. Dans ce mémoire nous nous servons de trois variantes de réseau de neurones récurrents (RNN) pour effectuer la détection de messages haineux. Nous avons pour cela utilisé deux jeux données à savoir : un jeu de données obtenu en effectuant du *scraping* sur des pages facebook d'utilisateurs camerounais, et un notre obtenu sur la plate-forme Kaggle. Nous avons montré qu'un RNN de type *Long Short Time Memory* (LSTM) permet d'avoir de meilleurs performances métriques par rapport à un *Gated Recurrent Unit* (GRU) et un RNN standard, mais implique un temps d'exécution plus important. Pour avoir à la fois de bonnes performances métriques et un temps d'exécution réduit, nous avons procédé à une implémentation parallèle des algorithmes d'entraînement. Nous avons proposé une implémentation parallèle basée sur une stratégie de parallélisation sans agrégation en comparaison à l'approche existante qui est basée sur une stratégie avec fonction d'agrégation. Les résultats expérimentaux sur une machine de 8 cœurs à 2,20 GHz montrent qu'on obtient de meilleurs résultats avec la stratégie de parallélisation qui a été proposée. Notamment, pour l'implémentation parallèle d'un LSTM en utilisant le jeu de données obtenu sur Kaggle, l'on obtient une f-mesure de 0.70 et un speedup de 2,2 avec la stratégie d'implémentation sans agrégation, comparé à une f-mesure de 0,65 et un speedup de 2,19 avec celle effectuant une agrégation entre les unités de traitement.

Mots clés : Réseaux de Neurones Récurent, deep learning, programmation parallèle, message haineux.

ABSTRACT

The freedom of expression and opinion provided by social networks are very often factors favoring the dissemination of hate messages. Hate speech are a threat to social stability and peace because it stimulates incitement to discrimination, which international law prohibits. To limit the harmful effects of this scourge, we often integrate into social network platforms models provided by deep learning algorithms allowing them to detect and react automatically to a message with the hateful nature. One of the particularities of these algorithms is that they are more efficient as the amount of data used is large. However, the sequential execution of these algorithms on large amounts of data can take a very long time. In this thesis we use three variants recurrent neural network (RNN) to perform hate message detection. For this, we used two datasets, namely : a dataset obtained by performing *scraping* on facebook pages of Cameroonian users, and another one obtained on the Kaggle platform. We have shown that a Long Short Time Memory (LSTM) type RNN provides better metric performance compared to a Gated Recurrent Unit (GRU) and a standard RNN, but involves a longer execution time. To have both good metric performance and reduced execution time, we proceeded to a parallel implementation of the training algorithms. We proposed a parallel implementation based on a parallelization strategy without aggregation in comparison to the existing approach which is based on an aggregation function. The experimental results on an 8-core machine at 2.20 GHz show that, we have the best results with the parallelization strategy that has been proposed. In particular, for the parallel implementation of an LSTM using the dataset obtained on Kaggle, we have an f-measure of 0.70 and a speedup of 2.2 with the implementation strategy without aggregation, compared to an f-measure of 0.65 and a speedup of 2.19 with that performing an aggregation between processing units.

Keywords : Recurent Neural Network, deep learning, parallel programming, text classification, hate messages.

Table des matières

1 Introduction **1**

1.1 Contexte 1

1.2 Problème 2

1.3 Solution proposée 2

1.4 Plan du mémoire 2

2 Généralités et travaux existants **3**

2.1 Généralités 3

2.1.1 Machine Learning 3

2.1.2 Deep Learning 5

2.1.3 Mesures de performances d'un modèle de machine learning 6

2.1.4 Programmation Parallèle 7

2.1.5 Mesure de performances d'un calcul parallèle 9

2.2 Travaux existants 9

2.2.1 La détection des messages haineux avec le deep learning 10

2.2.2 Parallélisation des algorithmes de deep learning 11

3 Parallélisation des RNNs **14**

3.1 Les Réseaux de Neurones Récurrents 14

3.1.1 Généralités sur les RNNs 14

3.1.2 Les Réseaux de Neurones Récurrents Standard 15

3.1.3 Long Short Time Memory (LSTM) 19

3.1.4 Gated Recurrent Unit (GRU) 22

3.2 RNNs parallèles 25

3.2.1 Algorithme séquentiel 25

3.2.2 Stratégie de parallélisation 30

3.2.3 Algorithme parallèle 31

3.2.4 Comparaisons aux approches existantes 33

4 Expérimentations et Résultats **36**

4.1 Cadre expérimental 36

4.2 Exécution séquentielle 37

4.3 Exécution parallèle 38

5 Conclusion et perspectives **42**

5.1 Conclusion 42

5.2 Perspectives 43

.

Chapitre 1

Introduction

Les plates-formes de réseaux sociaux se réfèrent très souvent aux techniques de *Deep Learning* pour construire des modèles pouvant détecter, et réagir automatiquement aux messages à caractère haineux comme nous le faisons dans ce mémoire. Nous présenterons dans ce chapitre le contexte, ainsi qu'une idée de la solution au problème que nous voulons résoudre.

1.1 Contexte

La communication numérique désigne tout type de communication dont le canal est un support numérique (téléphone, tablette, ordinateur ...). Cette façon de communiquer a connu un grand succès depuis l'avènement d'internet, comme peut en témoigner l'étude annuelle sur le digital en 2022 [29]. Dans ce rapport, l'on apprend qu'en début d'année 2022 l'on avait 4.62 milliard d'utilisateurs actifs des réseaux sociaux, soit 58.4% de la population mondiale. Ces taux d'utilisation élevés sont dû aux différents points positifs que ce canal de communication apporte, à savoir réunir des personnes à travers le monde, prôner la liberté d'expression et d'opinion. Cependant, cette manière de communiquer a aussi un certains nombres d'inconvénients, à l'instar de la diffusion des messages haineux, qui sont souvent source de meurtre, suicide, incitation à la violence ou de discrimination de tout genre. En effet, Dans un rapport publié en mai 2023 [17] présentant des mesures à prendre pour lutter contre les messages haineux, le gouvernement camerounais identifie clairement les réseaux sociaux comme le principal canal de propagation de messages de haine. De nombreux travaux comme celui effectué dans ce mémoire sont menés dans le but de réduire cet impact négatif des plates-formes des réseaux sociaux.

1.2 Problème

L'énorme quantité des données générées par les plates-formes de réseaux sociaux constituent une aubaine pour les algorithmes de *Deep Learning*. En effet, plus la quantité de données utilisées est importante, plus l'algorithme est performant. En contrepartie, l'exécution séquentielle de ces algorithmes sur ces grandes quantités de données peut prendre un temps d'exécution très élevé [7]. Il est possible grâce à un usage approprié des architectures multi-cœurs de réduire ce temps d'exécution. Dans nos travaux nous nous intéressons à l'identification des messages haineux pouvant être diffusés sur les réseaux sociaux en utilisant les algorithmes de deep learning. L'objectif est de prendre avantage des architectures matériels multi-cœurs pour réduire le temps d'exécution de ces algorithmes tout en préservant leurs performances métriques.

1.3 Solution proposée

Les algorithmes de *Deep Learning* se sont révélés performants dans de nombreuses applications pratiques comme la reconnaissance vocale ou encore la classification d'images. Les réseaux de neurones récurrents (RNNs), qui font l'objet de ce mémoire, ont l'avantage de pouvoir traiter des données séquentielles comme du texte. Cet avantage fait d'eux les algorithmes les mieux adaptés pour des tâches comme la classification de texte [12]. Malgré leurs bonnes performances ça reste un challenge de pouvoir utiliser les RNNs en raison des calculs lourds et complexes à effectuer lors de la formation des modèles. Cela peut prendre plusieurs jours pour avoir un modèle RNN précis avec un grand ensemble de données [12]. Nous proposons dans ce mémoire un algorithme d'entraînement parallèle d'un modèle RNN destiné à la prédiction (classification) des messages haineux. L'objectif étant de réduire le temps d'exécution tout en préservant les performances métriques (que l'on obtiendrait lors un entraînement séquentiel).

1.4 Plan du mémoire

Dans la suite, nous allons au chapitre 2 présenter les notions nécessaires à la compréhension de nos travaux, et les travaux existant similaires aux nôtres. Ensuite au chapitre 3 nous présenterons l'algorithme de réseaux de neurones récurrent et ses variantes, ainsi que l'approche utilisée pour l'implémentation parallèle. Nous présenterons au chapitre 4 les résultats expérimentaux actuels, et enfin le chapitre 5 permettra de conclure et de donner quelques perspectives.

Chapitre 2

Généralités et Travaux existants

Dans ce chapitre, nous présenterons tout d'abord les notions nécessaires à la compréhension des travaux de *Machine Learning* et en programmation parallèle, ainsi qu'un ensemble d'approches pour évaluer et valider nos résultats. Nous présenterons ensuite les travaux existant concernant la programmation parallèle en *Deep Learning* ainsi que sur l'utilisation des algorithmes de *Deep Learning* pour la détection des messages haineux.

2.1 Généralités

Dans cette section, nous présentons les notions de *Machine Learning* et *Deep Learning* qui font partie intégrante de l'ensemble de nos travaux, ainsi que celles liées au paradigme de programmation parallèle.

2.1.1 Machine Learning

Le *Machine Learning* est une branche de l'intelligence artificielle (IA) qui permet aux systèmes informatiques d'apprendre directement à partir des données et d'expériences [28]. En d'autres termes, c'est la branche de l'IA qui englobe l'ensemble d'algorithmes permettant de créer de façon automatique un modèle pouvant résoudre un problème précis à partir d'un ensemble de données. L'algorithme utilisé dépend du type de problème que nous souhaitons résoudre et du type de données en notre possession. Nous sommes généralement confrontés à devoir résoudre 03 principaux types de problèmes à savoir :

- **Apprentissage supervisé** : C'est un problème en machine learning où l'objectif est de construire un modèle prédictif pouvant prédire l'étiquette inconnue y d'une variable x via un algorithme d'entraînement (K-NN, Naive Bayes, SVM, ...) et

un jeu donnée (x^i, y^i) dont l'étiquette y^i est déjà connue (voir figure 2.1).

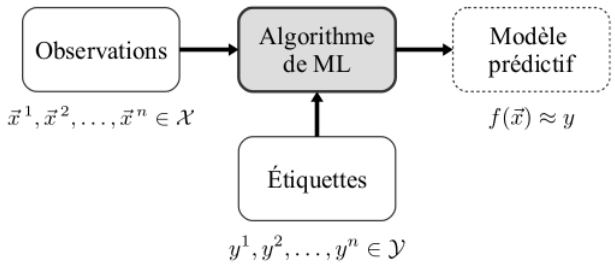

FIGURE 2.1 – Apprentissage Supervisé

Dans le cas où les étiquettes sont binaires c'est à dire $y^i \in \{0, 1\}$, l'on parle de problème de classification binaire. Dans le cas où les étiquettes correspondent à plusieurs classes c'est à dire $y^i \in \{1, 2, \ldots, C\}$, on parle de classification multi-classe, et enfin dans le cas où les étiquettes sont à valeurs réelles ($y^i \in \mathbb{R}$), on parle de problème de régression.

- **Apprentissage non supervisé** : Dans le cadre d'un problème d'apprentissage non supervisé, les données utilisées pour l'entraînement ne sont pas étiquetées. Les algorithmes (PCA, LSA, K-means, ...) détectent eux même les structures internes dans les données. On distingue principalement 02 sous problèmes pour cette catégorie, à savoir le problème de clustering dont l'objectif est d'identifier des sous-ensembles de données $\{x^i\}$ ayant des caractéristiques communes dans l'ensemble de données initiales (voir figure 2.2).

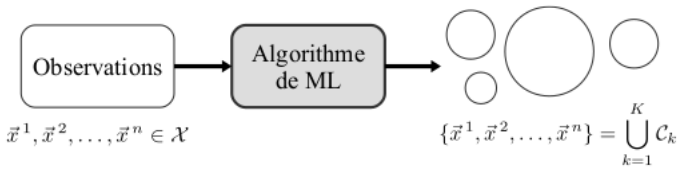

FIGURE 2.2 – Clustering

Et le problème de réduction de dimension, où il s'agit de trouver une représentation des données dans un espace de dimension plus faible que celle de l'espace dans lequel elles sont représentées à l'origine (voir figure 2.3).

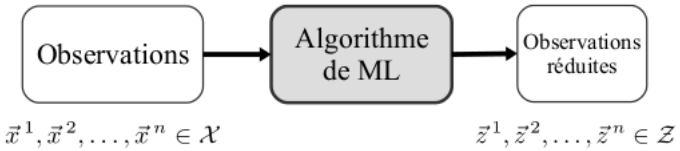

FIGURE 2.3 – Réduction de dimension

- **Apprentissage par renforcement** : Dans le cadre de l'apprentissage par renforcement, le système d'apprentissage (l'agent) peut interagir avec un environnement et accomplir des actions en fonction de son état actuel. En retour de ces actions, il obtient une récompense, qui peut être positive si l'action était un bon choix, ou négative dans le cas contraire (voir figure 2.4). L'objectif de l'apprentissage consiste alors dans ce cas à définir une politique, c'est-à-dire une stratégie permettant d'obtenir systématiquement la meilleure récompense possible. Un exemple d'algorithme à apprentissage par renforcement est le *Q-learning*.

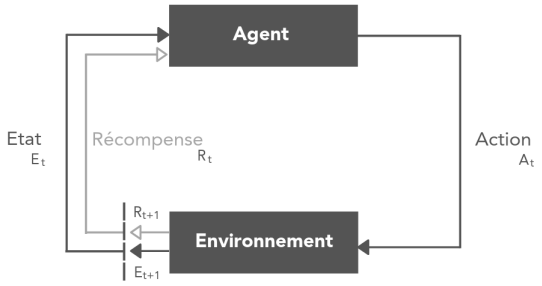

FIGURE 2.4 – Apprentissage par renforcement [30]

2.1.2 Deep Learning

Le *Deep Learning* est un sous domaine du machine learning (voir figure 2.5) dont l'objectif est de résoudre les problèmes du machine learning en utilisant des algorithmes simulant le fonctionnement des neurones biologiques humains. Ces algorithmes sont qualifiés de réseaux de neurones artificiels à l'exemple des réseaux de neurones profonds, les réseaux de neurones récurrents ou encore les réseaux de neurones convolutifs. Ils sont utilisés pour des tâches complexes de machine learning comme la reconnaissance vocale, la reconnaissance faciale, la classification d'image, ou encore la classification de texte comme nous le faisions dans ce mémoire.

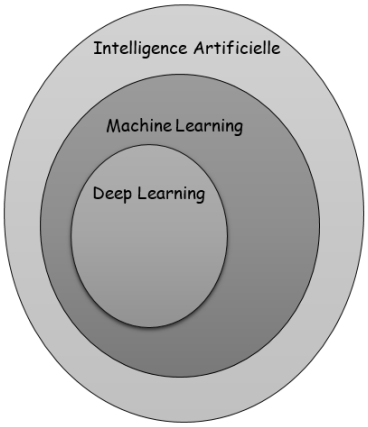

FIGURE 2.5 – l'IA, Machine Learning, Deep Learning

2.1.3 Mesures de performances d'un modèle de machine learning

Plusieurs mesures métriques sont utilisées pour évaluer la performance d'un modèle de classification, chacune d'elle ayant pour rôle d'évaluer un aspect précis. Nous présentons ici une description de quelques mesures qui seront utilisées pour évaluer nos modèles. Lors de l'évaluation de la performance d'un modèle de classification, quatre valeurs sont calculées pour chaque classe :

VP (Vrai Positif) : Il s'agit du nombre d'éléments catégorisés dans une classe à laquelle ils appartiennent effectivement.

FP (Faux Positif) : C'est le nombre d'éléments catégorisés dans une classe à laquelle ils n'appartiennent pas.

FN (Faux Négatif) : C'est le nombre d'éléments appartenant à une classe mais ayant été assignés à une autre classe.

VN (Vrai Négatif) : C'est le nombre d'éléments ayant été correctement reconnus comme n'appartenant pas à la classe.

A partir de ces valeurs, l'on définit alors les mesures métriques de performances suivantes :

- **La précision** : elle définit la proportion d'éléments bien classés dans une classe donnée. Elle est définie par :

$$Precision = \frac{VP}{VP + FP} \tag{2.1}$$

- **Le rappel** : pour une classe donnée, il représente la proportion d'éléments appartenant à cette classe et reconnu comme tel par le modèle. Il est formellement

défini par :

$$Rappel = \frac{VP}{VP + FN} \tag{2.2}$$

- **La F-mesure** : c'est la moyenne harmonique entre le rappel et la précision. Elle mesure de façon générale la performance de classification du modèle. Cette valeur varie entre 0 et 1, plus elle est proche de 1, plus le modèle est performant. Elle est définie par :

$$f - mesure = \frac{2 \times (precision \times rappel)}{precision + rappel} \tag{2.3}$$

2.1.4 Programmation Parallèle

Le paradigme de programmation parallèle désigne l'ensemble de méthodes permettant de prendre avantages des architectures multi-cœurs pour écrire des programmes avec des instructions s'exécutant simultanément, afin de réduire le temps d'exécution. La possibilité d'écrire un programme parallèle dépend fortement de l'architecture matérielle de la plate-forme d'exécution [22], et du modèle de programmation parallèle que l'on désire implémenter.

Architectures parallèles : la taxonomie de Flynn

Selon la structure du réseaux d'interconnexion entre les unités de traitement et la mémoire, ainsi que la coordination de travail entre ses unités de traitement, il existe selon la taxonomie de Flynn, 04 principaux types de machine parallèle [10] :

- **Machines SISD (*Single-Instruction, Single-Data*) :** se sont des machines avec une seule unité de traitement (processeur), qui exécutent et accèdent qu'à une seule instruction à la fois avec sa donnée correspondante.
- **Machines MISD (*Multiple-Instruction, Single-Data*) :** se sont des machines avec plusieurs unités de traitement, pouvant exécuter en parallèle un ensemble d'instructions différentes en utilisant les mêmes données chargées dans chacune de leur mémoire locale.
- **Machines SIMD (*Single-Instruction, Multiple-Data*) :** se sont des machines avec plusieurs unités de traitement contrôlées par une unité de contrôle centralisée. Chaque unité de traitement exécute en parallèle au autre la même instruction sur des données différentes : il s'agit là du parallélisme au niveau des données.
- **Machine MIMD (*Multiple-Instruction, Multiple-Data*) :** se sont des machines avec plusieurs unités de traitement chacune ayant sa propre unité de contrôle. Chaque unité de traitement exécute en parallèle au autre une instruction pouvant

être différente, avec des données différentes obtenues par accès privé à une mémoire partagée ou distribuée. Elles correspondent aux machines les plus courantes de nos jours.

Modèles de programmation parallèle

Un modèle de programmation parallèle désigne la manière avec laquelle un algorithme parallèle sera implémenté sur une architecture particulière. Il est principalement constitué de deux éléments, à savoir l'organisation de la mémoire et la décomposition du problème. L'organisation de la mémoire désigne la façon dont communiquent les différentes unités de traitement, on distingue principalement 02 types d'organisations à savoir :

- **Organisation à mémoire distribuée** : cette organisation correspond à un ensemble d'unité de traitement appelé noeuds, et d'un réseau d'interconnexion entre les noeuds ainsi qu'un support de transfert de données [22]. Comme le montre la figure 2.6 où, chaque noeud est indépendant des autres et est constitué de son processeur, sa mémoire local et parfois de ses périphériques.

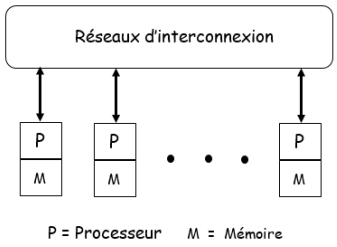

FIGURE 2.6 – organisation à mémoire distribuée

- **Organisation à mémoire partagée** : C'est l'organisation mémoire que nous avons utilisée tout au long de nos travaux. Elle désigne une machine physique avec plusieurs processeurs ou cœurs et une mémoire physique partagée (mémoire globale), ainsi qu'un réseau d'interconnexion pouvant connecter les processeurs à la mémoire (voir figure 2.7). Les cœurs d'un processeur multicœur sont un exemple pour cette organisation mémoire.

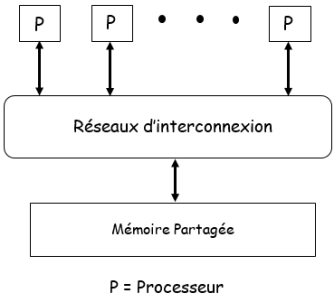

FIGURE 2.7 – organisation à mémoire partagée

La décomposition du problème désigne le niveau de parallélisation que l'on désire implémenter en fonction des calculs effectués dans la version séquentielle d'un algorithme. Il existe principalement 04 niveaux de parallélisation [22] à savoir : La parallélisation au niveau des instructions, au niveau des tâches, au niveau des boucles et au niveau des données. Cette dernière (au niveau des données) considérée dans ce mémoire, consiste à diviser les données initiales en plusieurs sous-ensemble. Chaque sous-ensemble est ensuite attribué à une unité de traitement pour une exécution simultanée.

2.1.5 Mesure de performances d'un calcul parallèle

Soit p le nombre de ressources d'une machine parallèle. Une ressource correspond à un cœur de calculs dans un processeur multi-coeurs ou à un processeur dans une machine multiprocesseurs, ou même à un nœud (PC) dans un cluster. Les performances dues à la parallélisation d'un algorithme seront mesurées à travers :

- **Le temps d'exécution T(p)** : C'est le temps d'exécution mit par un programme parallèle en utilisant p ressources.
- **Le *Speedup* S(p)** : C'est le rapport entre le temps d'exécution avec une ressource, sur le temps d'exécution sur p ressources.

$$\mathsf{S(p)} = \frac{\mathsf{T(1)}}{\mathsf{T(p)}} \tag{2.4}$$

Lorsqu'on effectue une bonne parallélisation, on devrait avoir une valeur de S(p) comprise entre 1 et p, en espérant qu'elle soit le plus proche de p.

2.2 Travaux existants

Cette section est dédiée aux travaux existant similaires aux nôtres. Nous présenterons séparément les travaux antérieurs sur l'usage du deep learning pour la détection des mes-

sages haineux, et d'autre part les travaux antérieurs sur la parallélisation des algorithmes de deep learning.

2.2.1 La détection des messages haineux avec le deep learning

La détection ou prédiction des messages haineux est une tâche de classification supervisée en machine learning (classification de texte). De nombreux travaux ont vu le jour avec pour objectif d'effectuer cette tâche le plus parfaitement possible. Nous présentons ici certains de ces travaux basé sur les algorithmes de *deep learning* en particulier ceux qui utilisent les RNNs.

Arum Sucia et al. [26] dans leurs travaux utilisent un RNN de type LSTM (Long Short Time Memory) pour former un modèle pouvant détecter de façon automatique si un message est haineux ou pas. Ils utilisent un jeu de données obtenu grâce à une API twitter. Le jeu de données contient au total 1235 tweets 652 tweets haineux et 583 tweets non haineux. Ils appliquent au préalable un ensemble d'opérations de pré-traitement au jeu de données, à savoir le *Case Folding*, *Tokenizing*, *Cleaning*, et *Stemming* mais aussi un algorithme de Word2Vec pour pouvoir représenter les mots dans chaque tweets en vecteurs numérique. Ils obtiennent un modèle ayant comme performances métriques, une précision à 91%, rappel à 90 % et l'accuracy à 91% .

Georgios et al. [21] proposent une architecture basée sur les RNNs et le paradigme d'apprentissage par ensemble pour la détection des messages haineux. Ils utilisent un ensemble de modèle formé par un RNN de type LSTM comme modèle faible, et agrègent ensuite leurs résultats pour avoir le modèle final. L'entraînement est effectué avec un jeu de données contenant 1943 tweets racistes , 3166 tweets sexistes et 10889 tweets neutres. Ils intègrent au jeu de données diverses informations associées à l'utilisateur, telles que la tendance d'un utilisateur à promulguer des commentaires raciste ou sexistes. Ils obtiennent un modèle ayant comme performances métriques, une précision à 93,05, % rappel à 93,34% et f-mesure à 93,20 % .

Ces travaux illustrent l'efficacité que peut avoir un RNN lorsqu'il est utilisé pour une tâche de détection de message haineux. Cependant, les auteurs que nous avons précédemment cité n'ont pas pris en compte le temps pour la formation du modèle. Notre objectif est donc d'utiliser un RNN pour la détection de message haineux, en prenant en compte non seulement les mesures métriques de performance, mais aussi le temps nécessaire pour la phase d'entraînement.

2.2.2 Parallélisation des algorithmes de deep learning

Plusieurs travaux ont été réalisés sur l'implémentation parallèle des algorithmes de deep learning. En analysant ces travaux, l'on observe principalement deux approches de parallélisation à savoir la parallélisation du modèle (au niveau des instructions) et la parallélisation au niveau des données qui peut s'effectuer de façon synchrone ou asynchrone.

C'est par exemple le cas de Martin Abadi et al. [1] qui, dans leurs travaux, présentent l'algorithme S-PSGD (*Synchronous Parallel Stochastic Gradient Descent*). C'est une implémentation parallèle de la descente stochastique du gradient exploitant la parallélisation au niveau des données. L'algorithme considère plusieurs unités de traitement, chacune avec une copie locale du modèle global et un sous-ensemble du jeu de données. Chaque unité de traitement calcule les gradients (∇P) des paramètres de son modèle local en utilisant le sous-ensemble du jeu de données qui lui a été attribué. L'algorithme est synchrone au sens où, les unités de traitements se synchronisent entre eux pour agréger leur gradient afin de mettre à jour le modèle global. Ce modèle de parallélisation est plus facile à mettre en place que se soit dans un environnement à mémoire partagé ou distribué [6]. Malgré quelques désavantages dus à la synchronisation à savoir : le fait que si une unité de traitement tombe en panne, c'est l'ensemble du système qui devient défaillant. Il reste tout de même le modèle de parallélisation le plus utilisé.

Jeffrey Dean et al. [7] combinent la parallélisation du modèle et la parallélisation au niveau des données de façon asynchrone. Ils présentent l'algorithme *DownPour* SGD. Le principe est que, chaque unité de traitement est chargée de mettre à jour un sous-ensemble de paramètres du modèle. Cette approche est asynchrone au sens où les unités de traitement fonctionnent indépendamment les unes des autres. La mise à jour d'un sous-ensemble de paramètres du modèle est effectuée par une unité de traitement dès qu'elle a terminé les calculs des gradients. Cet algorithme a pour avantage d'éliminer le temps nécessaire pour la synchronisation, mais elle est plus appropriée lorsque le modèle à entraîner est très volumineux (modèle d'apprentissage non supervisé) pour tenir en mémoire sur une unité de traitement. Elle ne nécessite qu'une quantité réduite de la mémoire pour chaque appareil, pour stocker et effectuer les traitements liés à un sous-ensemble de paramètres du modèle [18]. Par contre elle est moins utilisée que la version synchrone car elle est plus difficile à implémenter et la convergence n'est pas stable.

Feng Niu et al. [23] présentent dans leur travail l'algorithme *Hogwild*. C'est une implémentation parallèle asynchrone de la descente du gradient en considérant l'entraînement d'un modèle *sparse*. L'implémentation est effectuée dans un environnement à mémoire

partagée, où chaque unité de traitement peut accéder et mettre à jour le modèle via la mémoire partagée. L'approche est asynchrone au sens où, aucune synchronisation logicielle entre les unités de traitement n'est implémentée pour la lecture ni même pour la mise à jour du modèle. Les opérations de mise à jour sont gérées par le matériel comme des primitives atomiques. Étant donné que le modèle est sparse, les auteurs montrent que cette stratégie converge aussi bien que la version séquentielle, tout en produisant un gain en temps linéaire en fonction du nombre d'unités de traitement.

Zhiheng Huang et al. [12] proposent un algorithme parallèle d'entraînement d'un RNN pour la modélisation du langage. Pour leur implémentation, chaque unité de traitement contient en mémoire une copie complète du modèle, mais aussi un sous ensemble du jeu de données (tous les sous ensemble ne sont pas forcément disjoints entre eux). La différence avec l'algorithme S-PSGD est qu'ici, chaque unité de traitement met à jour son modèle local. L'étape de synchronisation agrège cette fois-ci les différents modèles locaux pour former le modèle global. Cette implémentation a pour avantage de réduire le temps de communication entre les unités de traitement, mais par contre le choix de fonction ou méthode d'agrégation est crucial pour garantir une bonne convergence.

Le point commun des travaux que nous venons de citer est le fait qu'ils utilisent tous la parallélisation au niveau des données. La principale différence est faite sur la stratégie utilisée pour la mise à jour du modèle global. Le tableau 2.1 présente un récapitulatif des travaux existants dont nous avons disputés, ainsi que notre positionnement par rapport à ces travaux.

Articles	**Objectif Principal**	**Stratégie de parallélisation**	**Algorithme utilisé**	**Résultats**
Georgios et al. [21] (2018)	Détection des messages haineux	Aucune stratégie présenté	LSTM + Apprentissage par ensemble	f-mesure = 93% ; rappel = 93% ; précision = 93%
Arum et al. [26] (2018)	Détection des messages haineux	Aucune stratégie présentée	LSTM	acuracy 91% ; rappel = 90% ; précision = 91%
Martin Abadi et al. [1] (2016)	Système distribué pour le machine Learning	Parallélisation des données : S-PSGD (*Synchronous Parallel Stochastic Gradient Descent*)	non spécifié	non spécifié
Jeffrey Dean et al. [7] (2012)	Réduire le temps d'entraînement d'un réseaux de neurone profond	Parallélisation du modèle + Parallélisation des données (Asynchrone)	réseaux de neurone profond (1,7 milliard de paramètres)	SpeedUp = 12 avec 21 unité de traitements
Feng Niu et al. [23] (2011)	Réduire le temps d'entraînement d'un modèle *sparse*	Algorithme Hogwild basé sur Parallélisation des données (Asynchrone), sans implémentation de synchronisation logicielle	SVM	SpeedUp = 4 avec 10 unité de traitements
Zhiheng Huang et al. [12] (2013)	Réduire le temps d'entraînement d'un RNN	Parallélisation des données avec agrégation de paramètres pour les opérations de mise à jour	RNN standard	SpeedUp = 20 avec 50 unité de traitements
Dans ce mémoire	Réduire le temps d'entraînement d'un RNN	Parallélisation des données avec une synchronisation *mutex* entre les unités de traitement pour les opérations de mise à jour	LSTM	SpeedUp = 2 avec 8 unités de traitements ; f-mesure = 70% ; rappel = 70% ; précision = 72%

TABLE 2.1 – Récapitulatifs des travaux existants et idée de la solution

Chapitre 3

Parallélisation des RNNs

Les réseaux de neurones récurrents (RNNs) sont des types de réseaux de neurones utilisés pour modéliser des données présentant un comportement temporel ou séquentiel. Pour cette raison, ils sont particulièrement adaptés pour plusieurs tâches de NLP (Natural Language Processing) [8] à l'exemple la reconnaissance de la parole, ou encore la classification de texte. Malgré leurs bonnes performances, le temps d'entraînement pour former un modèle avec un RNN peut être très grand (plusieurs jours). Un moyen intuitif pour accélérer la formation d'un RNN est via la programmation parallèle [12] comme nous le faisons dans ce mémoire. Nous présenterons dans ce chapitre le principe et fonctionnement des RNNs ainsi que la stratégie d'implémentation parallèle que nous avons proposée.

3.1 Les Réseaux de Neurones Récurrents

Nous présentons dans cette section le fonctionnement de trois variantes de RNNs à savoir : RNN standard, LSTM (Long Short Time Memory) , et GRU (Gated Recurrent Unit).

3.1.1 Généralités sur les RNNs

Les RNNs [27] sont des types de réseaux de neurones utilisés pour le traitement des données séquentielles. Contrairement au réseau de neurones à propagation avant (Feed-Forward Neural Network) où l'information ne se propage que dans un sens , de l'avant vers l'arrière, les RNNs sont des réseaux de neurones présentant des connexions récurrentes et où l'information est propagée dans les deux sens. Dans la littérature des RNNs pour le NLP, deux principales architecture ont été proposées en fonction de la position de la

connexion récurrente à savoir : les réseaux de neurones de jordan [13] et d' Elman [9]. les réseaux de neurones de Jordan sont des RNNs avec une couche d'entrée, une couche cachée et une couche de sortie liée de façon cyclique à la couche cachée par des unités de contexte faisant office de mémoire. Les réseaux de neurones d'Elman différent des réseaux de neurones de jordan par le fait que, c'est la couche cachée qui est liée à elle-même par des unités de contexte (voir figure 3.1). Ils sont les plus utilisés pour des tâches de classification de texte, et feront donc principalement l'objet de notre étude.

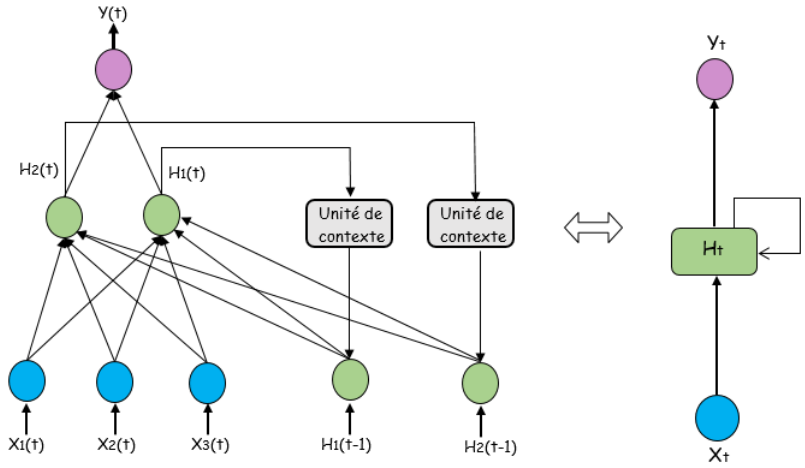

FIGURE 3.1 – RNN d'Elman : version éclatée à gauche et version compacte à droite, avec x_t l'entrée à l'instant t, H_t l'état de la couche caché à l'instant t et y_t la sortie à l'instant t

En fonction de la procédure de calcul utilisée pour obtenir les sorties y_t à partir des entrées x_t, l'on distingue 03 sous variantes de RNNs à savoir les RNNs standard , LSTM, et GRU, chacune ayant des avantages et des inconvénients dont nous en parlerons dans la suite.

3.1.2 Les Réseaux de Neurones Récurrents Standard

Un RNN standard simule un système dynamique à temps discret. C'est un système doté d'une entrée x_t, une sortie y_t et un état caché h_t formellement définit par :

$$\begin{aligned} h_t &= f_h(x_t, h_{t-1}) \\ y_t &= f_o(h_t) \end{aligned} \tag{3.1}$$

Où f_h et f_o sont respectivement des fonctions paramétrées de transition d'état et de sortie. Comme tout réseau de neurones, l'apprentissage avec un RNN est basé sur

l'exécution itérative de deux phases, à savoir la phase de propagation avant et la phase de rétro-propagation. La phase de propagation avant a pour objectif de calculer les sorties y_t à chaque instant t étant donné les entrées x_t et l'état caché antérieur h_{t-1}. Pour un RNN standard cela peut être formellement définit comme le montre les équations suivantes :

$$\begin{aligned} h_t &= tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h) \\ v_t &= W_y h_t + b_y \\ y_t &= SoftMax(v_t) \end{aligned} \tag{3.2}$$

Où $x_t \in \mathbb{R}^{n_e \times 1}$ est le vecteur en entré à l'instant t. $h_t \in \mathbb{R}^{n_h \times 1}$ l'état caché à l'instant t. Les paramètres du RNN sont : les poids $W_{xh} \in \mathbb{R}^{n_h \times n_v}$, $W_{hh} \in \mathbb{R}^{n_h \times n_h}$,le biais $b_h \in \mathbb{R}^{n_h \times 1}$ de la couche cachée ainsi que les poids $W_y \in \mathbb{R}^{n_s \times n_h}$ et le biais $b_y \in \mathbb{R}^{n_s \times 1}$ de la couche de sortie. La fonction de transition d'état f_h utilisée est la fonction Tanh (Tangente hyperbolique) définie par :

$$\begin{array}{rccl} f_h & : & \mathbb{R} & \to &]-1,1[\\ & & x & \mapsto & \dfrac{1-e^{-2x}}{1+e^{-2x}} \end{array}$$

Lorsqu'elle est utilisée comme fonction de transition d'état, elle permet dans la grande majorité des applications liées aux réseaux de neurones d'obtenir de meilleurs résultats par rapport aux autres fonctions d'activation [14]. La fonction de sortie f_o utilisée est $SoftMax$, définie par :

$$\begin{array}{rccc} f_o & : & \mathbb{R}^n & \to & \mathbb{R}^n \\ & & \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} & \mapsto & \begin{pmatrix} f_o(x_1) \\ f_o(x_2) \\ \vdots \\ f_o(x_n) \end{pmatrix} \end{array}$$

$$avec \quad f_o(x_i) = \frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}} \quad et \quad n \in \mathbb{N}^*$$

Elle convertit un vecteur de nombres réels en une distribution de probabilité où chaque probabilité est proportionnelle à l'échelle relative d'une valeur dans le vecteur initial. Elle est utilisée en apprentissage profond comme fonction d'activation en sortie pour des problèmes de classification multi-classe (ce qui est notre cas).

La figure 3.2 résume la phase de propagation avant pour un RNN standard en 02 principales étapes à savoir :

1. **L'étape 1** : Où l'on calcule l'état caché h_t à l'instant t, en fonction de l'entrée x_t et de l'état caché précédent h_{t-1} . C'est cette étape qui théoriquement, permet d'avoir les sorties à l'instant t, en fonction des sorties aux instants précédents ;
2. **L'étape 2** : L'on calcule la sortie y_t en tenant compte des informations contenues dans h_t.

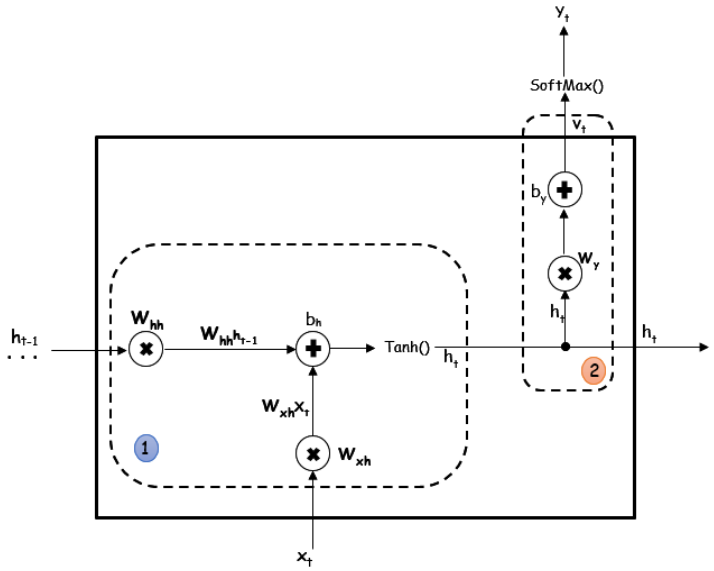

FIGURE 3.2 – Phase de propagation avant à un instant t pour un RNN Standard

La phase de propagation avant permet d'obtenir les sorties y_t. La prédiction est dite parfaite si ces sorties sont identiques aux sorties réelles $\bar{y}_t$ attendues. Dans le cas contraire, l'on dira que le réseau de neurones a commis des erreurs. L'objectif de la phase de rétro-propagation sera de lui faire apprendre de ses erreurs pour qu'il puisse améliorer ses prédictions aux prochaines itérations. c'est la phase d'apprentissage proprement dite.

On commence par définir à l'équation 3.3 la fonction $L(y_t, \bar{y}_t)$, comme étant la somme de toutes les erreurs ℓ_t obtenues jusqu'à l'instant t pour une instance $X = \{x_1, x_2, \ldots, x_t\}$ du jeux de données passée en entrée (voir figure 3.3). ℓ_t est une fonction d'erreur qui permet de quantifier l'écart entre la sortie y_t prédite et la sortie $\bar{y}_t$ attendue. Elle peut avoir différentes définitions en fonction du problème à résoudre (par exemple, l'erreur quadratique moyenne, la perte d'entropie croisée ... etc.). Nous considérons le cas spécifique où ℓ_t est définie comme étant la perte d'entropie croisée. C'est une fonction d'erreur utilisée

pour des problèmes de classification multi-classe (comme dans notre cas) , elle est définie à l'équation 3.4.

$$L(y_t, \bar{y}_t) = \sum_{t=1}^{T} \ell_t(y_t, \bar{y}_t) \tag{3.3}$$

$$\ell(y, \bar{y}) = -\sum_{i=1}^{n} \overline{y}_i log(y_i) \tag{3.4}$$

Où $\overline{y}_i$ est la probabilité réelle de la i^{eme} classe, et y_i la probabilité prédite.

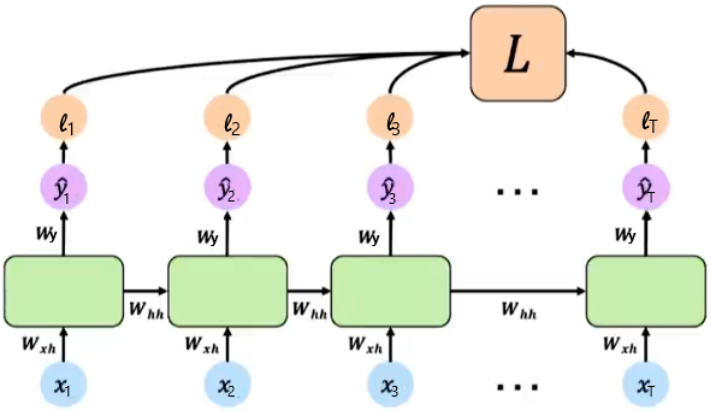

FIGURE 3.3 – Calcul de l'erreur pour une architecture RNN : avec x_t, y_t, ℓ_t respectivement l'entrée, la sortie et l'erreur à un instant t, et L la somme algébrique des erreurs ℓ_t

Formellement, l'objectif est donc de minimiser la fonction de perte L qui dépend des paramètres $\Theta \in \{W_y, W_{xh}, W_{hh}, b_y, b_h\}$. Il existe plusieurs méthodes d'optimisation pouvant nous aider à le faire. La plus utilisée en apprentissage profond est la méthode d'optimisation de la descente du gradient. Sous sa version la plus simple, pour minimiser l'erreur L, l'on doit de façon itérative pour chaque instance $X = \{x_1, x_2, \ldots, x_t\}$ mettre à jour les paramètres Θ comme le montre l'équation (3.5)

$$\Theta = \Theta - \lambda \frac{\partial \mathsf{L}}{\partial \Theta} \tag{3.5}$$

Où λ (le taux d'apprentissage) est un paramètre fixé avant le début de l'apprentissage et qui peut ou pas changer tout au long de l'apprentissage. Les dérivées partielles $\frac{\partial \mathsf{L}}{\partial \Theta}$

sont calculées en utilisant la rétro-propagation à travers le temps. c'est une adaptation de l'algorithme de rétro-propagation [25] pour les RNNs. Le principe est de rétro-propager l'erreur à travers les connexions récurrentes de l'arrière vers l'avant en sommant à chaque fois les dérivées partielles aux différents instants t. (voir figure 3.4)

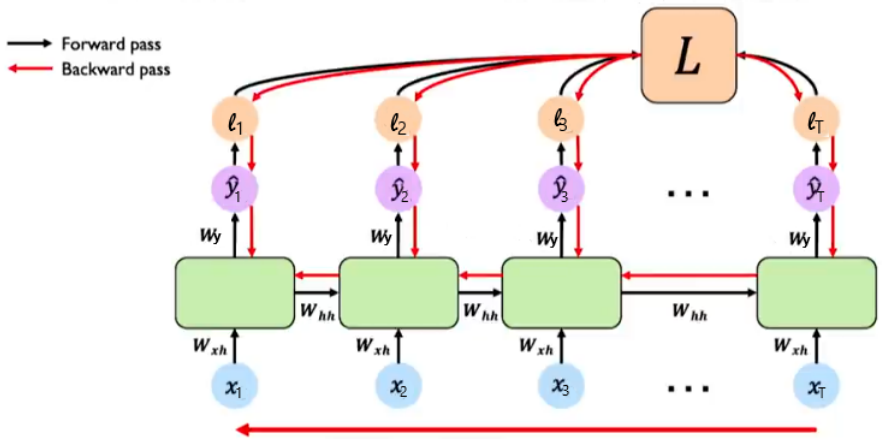

FIGURE 3.4 – Phase de rétro propagation pour une architecture RNN. Elle s'effectue en sens inverse de la phase de propagation avant : avec x_t, y_t, ℓ_t respectivement l'entrée, la sortie et l'erreur à un instant t, et L la somme algébrique des erreurs ℓ_t

En résumé, l'architecture d'un RNN standard lui donne théoriquement l'avantage de pouvoir traiter et manipuler des données présentant un comportement séquentiel. Mais comme le montrent Pascanu et al. [19] les RNNs standard ont en pratique des difficultés à pouvoir manipuler des séquences très longues dues au problème de "*vanishing*" ou "*exploding*" gradient. Pour comprendre l'origine des ces anomalies, l'on peut considérer un RNN standard comme une machine à calculer où h_t représente sa mémoire à l'instant t (une mémoire limitée). Le principal problème est que l'accès mémoire n'est pas contrôlé. C'est-à-dire , qu'à chaque pas de temps t, on lit et on réécrit la totalité de la mémoire. Par conséquent, plus une séquence en entrée est longue, plus on emmagasine de nouvelles informations et plus on en perd d'autres. Plusieurs solutions ont vu le jour pour pallier ces anomalies, comme l'usage d'architectures spéciales à l'instar des LSTM et GRU.

3.1.3 Long Short Time Memory (LSTM)

Le réseau de neurones récurrent LSTM introduit par Hochreiter et al. [11], a été conçu pour résoudre le problème de *vanishing* gradient. Un réseau de neurones LSTM subdivise sa mémoire en deux à savoir, une cellule mémoire secondaire c_t conçue pour préserver les gradients dans le temps, et la mémoire principale de travail h_t dont la valeur à l'instant t

dépend de la valeur de c_t. L'accès à la cellule mémoire est contrôlé par des portes logiques représentées par des fonctions mathématiques. L'objectif étant de filtrer via ces portes, l'information qui devrait être oubliée et celle qui devrait être conservée en mémoire. Pour une architecture LSTM, l'on distingue trois portes à savoir :

- ***Forget Gate*** (f_t) : permet de contrôler la quantité d'informations à conserver ou à oublier dans c_t ;
- ***Input Gate*** (i_t) : permet de sélectionner les informations à ajouter dans c_t ;
- ***Output Gate*** (o_t) : permet d'obtenir l'état de la mémoire h_t en fonction de c_t .

Pour une séquence $X = \{x_1, x_2..., x_t\}$ en entrée d'un réseau de neurones LSTM, la phase de propagation avant à un instant t , se déroule formellement comme le montre les équations suivantes :

$$\begin{aligned} f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \\ i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \\ \overline{c}_t &= Tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \\ c_t &= f_t \odot c_{t-1} + i_t \odot \overline{c}_t \\ o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \\ h_t &= o_t \odot tanh(c_t) \\ v_t &= W_y h_t + b_y \\ y_t &= SoftMax(v_t) \end{aligned} \tag{3.6}$$

Où $\overline{c}_t \in \mathbb{R}^{n_h \times 1}$ désigne l'état intermédiaire de la mémoire secondaire c_t (son état avant d'être filtré par les portes). f_t, i_t, et $o_t \in \mathbb{R}^{n_h \times 1}$ désignent respectivement les portes logique *forgate*, *input* , et *output gate*, elles sont conçues en utilisant la fonction d'activation sigmoïde (σ) définie par :

$$\begin{aligned} \sigma \ : \ \mathbb{R} \ &\rightarrow \quad]0,1[\\ x \ &\mapsto \ \frac{1}{1+e^{-x}} \end{aligned}$$

Elle renvoie des valeurs comprises dans l'intervalle]0,1[, ce qui permet aux portes de décider à chaque pas de temps les proportions d'informations à conserver ou à oublier. Les décisions prises par les portes sont améliorées au cours de l'apprentissage, elles sauront alors comment sélectionner et faire conserver uniquement les informations utiles. L'on peut réduire le nombre de variables à manipuler en concaténant les vecteurs $\mathsf{x_t}$ et $\mathsf{h_{t-1}}$ $(\mathsf{q_t} = [\mathsf{x_t}, \mathsf{h_{t-1}}])$, l'équation précédente change et devient donc :

$$
\begin{aligned}
f_t &= \sigma(W_f q_t + b_f) \\
i_t &= \sigma(W_i q_t + b_i) \\
\bar{c}_t &= Tanh(W_c q_t + b_c) \\
c_t &= f_t \odot c_{t-1} + i_t \odot \bar{c}_t \quad \text{(inchangée)} \\
o_t &= \sigma(W_o q_t + b_o) \\
h_t &= o_t \odot tanh(c_t) \quad \text{(inchangée)} \\
v_t &= W_y h_t + b_y \quad \text{(inchangée)} \\
y_t &= SoftMax(v_t) \quad \text{(inchangée)}
\end{aligned}
\tag{3.7}
$$

Les paramètres Θ du LSTM sont alors : les poids $W_i, W_f, W_o, W_c \in \mathbb{R}^{n_h \times (n_v + n_h)}$ et les biais b_i, b_f, b_o, $b_c \in \mathbb{R}^{n_h \times 1}$ de la couche cachée ainsi que les poids $W_y \in \mathbb{R}^{n_s \times n_h}$ et biais $b_y \in \mathbb{R}^{n_s \times 1}$ de la couche de sortie. La figure 3.5 résume la phase de propagation avant pour un LSTM en trois principales étapes à savoir :

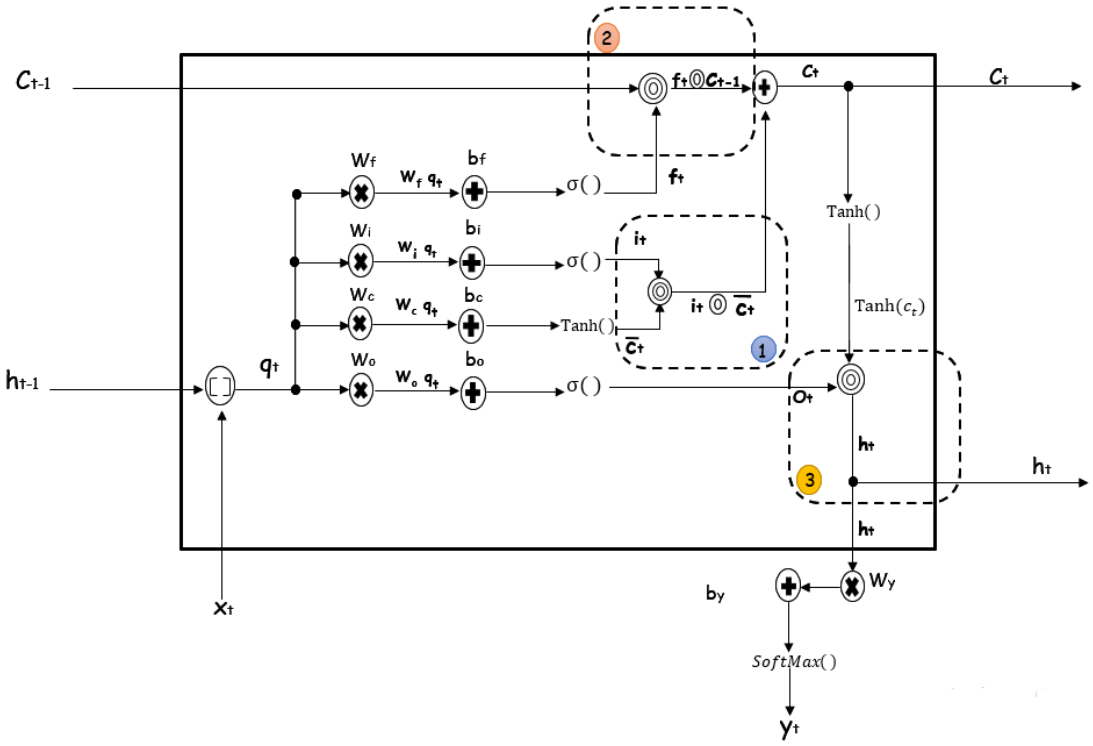

FIGURE 3.5 – Phase de propagation avant à un instant t pour un LSTM

1. **L'étape 1** : Où l'on applique l'effet de la porte *input gate* (i_t) à la mémoire

intermédiaire ($\mathsf{i_t} \odot \overline{\mathsf{c}}_\mathsf{t}$) en utilisant le produit vectoriel élément par élément '$\odot$' (le produit hadamard). Ce qui permet de sélectionner les informations qui seront finalement intégrées dans la cellule mémoire c_t

2. **L'étape 2** : L'on applique l'effet de la porte *forgate gate* (f_t) à l'état antérieur de la cellule mémoire ($\mathsf{f_t} \odot \mathsf{c_{t-1}}$). Ce qui permet d'oublier certaines informations.

3. **L'étape 3** : L'on calcule l'état caché h_t en appliquant l'effet de la porte *output gate* (o_t) à la cellule mémoire ($\mathsf{o_t} \odot \mathsf{tanh}(\mathsf{c_t})$). L'état caché h_t est ensuite utilisé pour obtenir la sortie y_t comme avec un RNN standard.

En définitive, un réseau de neurones LSTM est une architecture particulière de réseau de neurones récurrents. Sa conception est basée sur l'utilisation d'un mécanisme à porte pour contrôler le flux d'information entrant et sortant. Son architecture permet donc de pallier au problème de *vanishing* gradient afin de mieux saisir et utiliser les dépendances entre les données dans des séquences longues. Elle est cependant plus complexe que l'architecture d'un RNN standard et utilise beaucoup plus de paramètres. Il existe d'autres architecture de RNN à l'instar des GRU moins complexes que celle des LSTM, mais pouvant être tout aussi performant en fonction de la tâche à effectuer.

3.1.4 Gated Recurrent Unit (GRU)

Introduit par cho et al. [4] , un RNN de type GRU a aussi été conçu pour résoudre le problème de *vanishing gradient* qu'on rencontre avec un réseau neurone récurrent standard. Il est similaire au LSTM dans le sens où il utilise aussi un mécanisme à porte. L'on note tout de même des différences entre les deux au niveau du fonctionnement. Comme le fait qu'une architecture GRU n'utilise que l'état caché h_t comme cellule mémoire, et dont le flux d'informations entrant et sortant est directement contrôlé par des portes logiques. Mais aussi, le fait qu'une architecture LSTM utilise 03 portes par rapport à une architecture GRU qui n'en utilise que 02 à savoir :

- ***Reset Gate*** (r_t) : qui, étant donné l'état antérieur de la mémoire (h_{t-1}), permet de déterminer la quantité d'informations à oublier ;
- ***Update Gate*** (z_t) : qui, étant donné le flux d'informations entrant, permet de déterminer la quantité d'informations à ajouter dans h_t .

Pour une séquence $X = \{x_1, x_2, \ldots, x_t\}$ en entrée d'un réseau de neurone GRU, la phase de propagation avant pour un instant t, se déroule formellement comme le montre les

équations suivantes :

$$\begin{aligned} z_t &= \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \\ r_t &= \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \\ \overline{h}_t &= tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h) \\ h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \overline{h}_t \\ v_t &= W_y h_t + b_y \\ y_t &= SoftMax(v_t) \end{aligned} \tag{3.8}$$

Avec $\overline{\mathsf{h}}_\mathsf{t} \in \mathbb{R}^{n_h \times 1}$ désignant l'état intermédiaire de la mémoire $\mathsf{h_t}$ (son état avant d'être filtré par les portes) . z_t, $r_t \in \mathbb{R}^{n_h \times 1}$ désignent respectivement les portes logique *update gate*, et *reset gate*. Il est possible de réduire le nombre de variables à manipuler en concaténant les vecteurs $\mathsf{x_t}$ et $\mathsf{h_{t-1}}$ ($\mathsf{q_t} = [\mathsf{x_t}, \mathsf{h_{t-1}}]$), ainsi que les vecteurs $\mathsf{r_t} \odot \mathsf{h_{t-1}}$ et $\mathsf{x_t}$ ($\mathsf{s_t} = [\mathsf{x_t}, \mathsf{r_t} \odot \mathsf{h_{t-1}}]$). L'équation précédente change et devient donc :

$$\begin{aligned} z_t &= \sigma(W_z q_t + b_z) \\ r_t &= \sigma(W_r q_t + b_r) \\ \overline{h}_t &= tanh(W_h s_t + b_h) \\ h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \overline{h}_t \quad \text{(inchangée)} \\ v_t &= W_y h_t + b_y \quad \text{(inchangée)} \\ y_t &= SoftMax(v_t) \quad \text{(inchangée)} \end{aligned} \tag{3.9}$$

Les paramètres Θ sont alors : les poids $W_r, W_h, W_z \in \mathbb{R}^{n_h \times (n_v + n_h)}$ et les biais $b_r,\ b_h,\ b_z \in \mathbb{R}^{n_h \times 1}$ de la couche cachée ainsi que les poids $W_y \in \mathbb{R}^{n_s \times n_h}$ et biais $b_y \in \mathbb{R}^{n_s \times 1}$ de la couche de sortie. La figure 3.5 résume la phase de propagation avant pour un RNN standard en deux principales étapes à savoir :

1. **L'étape 1** : Où l'on applique l'effet de la porte *reset gate* (r_t) à l'état caché antérieur ($\mathsf{h_{t-1}} \odot \mathsf{r_t}$). Ce qui permet de sélectionner les informations à utiliser pour construire l'état caché intermédiaire $\overline{\mathsf{h}}_\mathsf{t}$
2. **L'étape 2** : L'on calcule l'état caché h_t en appliquant l'effet de la porte *update gate* (z_t) à l'état caché intermédiaire et antérieur ($\mathsf{z_t} \odot \mathsf{h_{t-1}} + (\mathsf{1} - \mathsf{z_t}) \odot \overline{\mathsf{h}}_\mathsf{t}$). $\mathsf{h_t}$ est ensuite utilisé pour obtenir la sortie y_t comme avec un RNN standard.

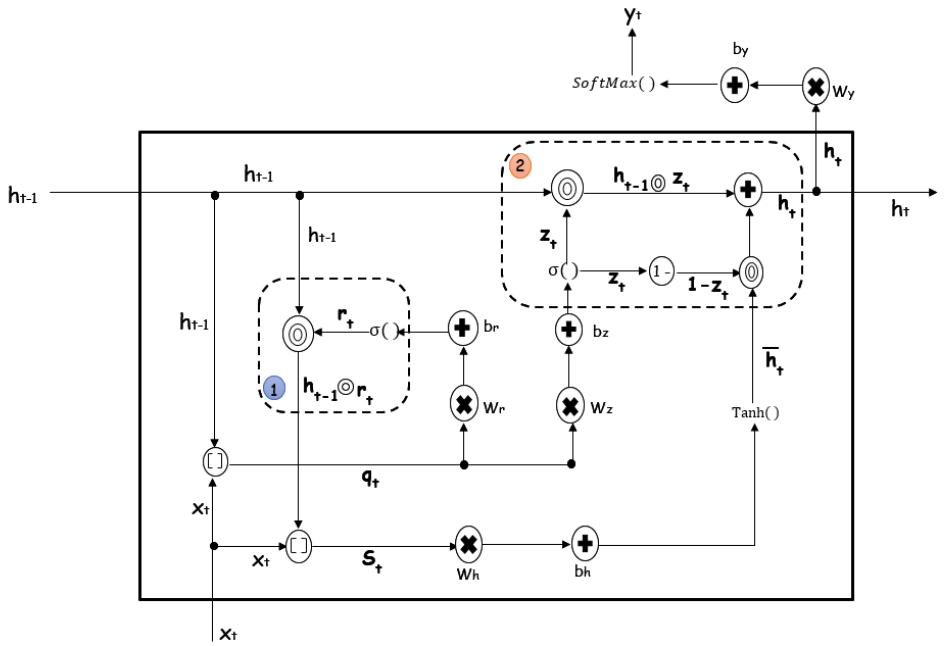

FIGURE 3.6 – Phase de propagation avant à un instant t pour un GRU

En résumé, la principale distinction entre un RNN standard et un GRU ou LSTM, est que ces derniers utilisent un mécanisme à porte afin de contrôler le flux d'information entrant et sortant. Le tableau 3.1 présente une comparaison entre les trois variantes.

	Standard	**LSTM**	**GRU**
Nombres de paramètres	$n_h(n_v + n_h + 1) + n_s(n_h + 1)$	$4n_h(n_v + n_h + 1) + n_s(n_h + 1)$	$3n_h(n_v + n_h + 1) + n_s(n_h + 1)$
Temps d'exécution	plus faible	plus grand	moins que le LSTM
État caché (cellule mémoire)	h_t	c_t et h_t	h_t
Porte de contrôle	Aucune	03 (f_t, i_t, o_t)	02 (z_t, r_t)
Capacité à conserver les dépendances à long terme	faible	forte	moyenne

TABLE 3.1 – Tableau Comparatif entre les variantes RNNs

Même si en pratique l'on ne sait pas au préalable quelle variante RNN utilisée [5], un LSTM ou un GRU est théoriquement plus performant qu'un RNN standard mais implique un temps d'entraînement plus grand. Dans le but de réduire le temps d'entraînement tout en préservant les performances, nous avons opté pour une implémentation parallèle de l'algorithme d'entraînement. La section suivante présente la stratégie de parallélisation proposée ainsi que l'algorithme parallèle implémenté.

3.2 RNNs parallèles

Nous présenterons dans cette section la stratégie de parallélisation proposée pour l'algorithme d'entraînement d'un RNN destiné à la classification de texte. Pour commencer, nous présenterons et analyserons en détail la version séquentielle de l'algorithme. Ensuite à partir de cette version séquentielle, nous mettrons sur pied une stratégie d'implémentation parallèle. La stratégie sera ensuite présentée de manière plus formelle avec une version parallèle de l'algorithme.

3.2.1 Algorithme séquentiel

La section 3.2 présentait de manière générale le principe et le fonctionnement des RNNs. Pour le cas particulier d'une tâche de classification de texte, un RNN traite une séquence de données en entrée (les mots du texte) et produit une seule sortie représentant la classe du texte. Dans cette approche, l'on ne considère pas les sorties intermédiaires pour les mots dans la séquence précédant le dernier mots (voir figure 3.7). Les mots constituant le texte sont au préalable transformés en vecteur numérique. Cette tâche est effectuée par des algorithmes de représentation vectorielle comme *fastext*, *glove*, ou *word2vec* [15, 16, 20] dont l'objectif est de représenter les mots du langage naturelle en vecteurs numériques tout en conservant le maximum d'informations sémantiques. Pour

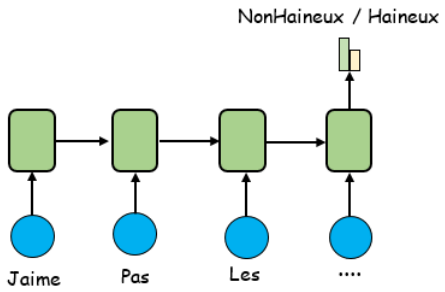

FIGURE 3.7 – La classification de textes avec un RNN. Les entrées x_t sont les mots du texte et la sortie y une distribution de probabilité permettant de déterminer la classe du texte.

nos expérimentations, nous avons utilisé un algorithme *word2vec* fournit par la bibliothèque gensim de python. Ainsi en entrée de l'algorithme d'apprentissage, l'on considère les mots $x_1, x_2, \ldots, x_n$ d'une phrase x déjà sous leurs formes vectorielles. L'algorithme 1 présente l'entraînement séquentiel d'un RNN pour la classification de texte. La procédure

Algorithm 1 : **Training**

Input (X,Y) : ensemble de phrase x et leur label y ; λ : le learning rate ;
M : le batch size ; epoch : le nombre d'époques maximal
Output Θ : Ensemble de paramètres du modèle
Start

```
1: initializeParameters(Θ)  // on initialise les paramètres Θ du modèle
2: (xtrain, ytrain) (xval, yval) ← splitData(X, Y) // on réserve une partie du jeu de données pour la validation
3: B ← getBatch(M,xtrain,ytrain)//on crée des lots de données de taille M
4: e ← 1
5: stop ← 0
6: valLoss ← ∞
7: while e ≤ epoch and stop < 4 do
8:     for b_i ∈ B do
9:         loss ← 0
10:        zeroInitialize(dΘ) // pour chaque lots de données, on initialise les dérivées partielles à zéro
11:        for (x, y) ∈ b_i do
12:            y_pred ← Forward(x, Θ) // pour chaque phrase x dans un lot de donnée, l'on prédit sa classe
13:            loss ← loss + lossEntropy(y, y_pred) // on accumule l'erreur de prédiction
14:            dΘ ← dΘ + Backforward(Θ, x, y, y_pred) //on calcul et on accumule les dérivées partielles
15:        end for
16:        update(Θ, dΘ, λ, M) //on mets à jour les paramètres
17:    end for
18:    if evalModel(Θ, xval, yval) > valLoss then
19:        stop ← stop + 1  // si les performances du modèle n'ont pas augmenté, on incrémente la variable stop
20:    else
21:        valLoss ← evalModel(Θ, xval, yval)
22:        stop ← 0          // si non, la variable stop est réinitialisé à zéro
23:    end if
24:    e ← e + 1
25: end while
26: Return Θ
End
```

d'optimisation utilisée est la descente du gradient sous version mini batch où l'on met à jour des paramètres Θ du modèle après avoir traité M éléments du jeu de donnée [24]. Elle se déroule principalement comme suit :

1. On partitionne le jeu de données en plusieurs lots $\mathsf{b_i}$ de taille M (ligne 3)
2. Pour chaque époque [1] (ligne 7 à 25), l'on parcourt tous les lots $\mathsf{b_i}$. Pour chaque phrase x dans $\mathsf{b_i}$, l'on prédit sa classe (ligne 12). L'on calcule et on accumule l'erreur de prédiction ainsi que les différentes dérivées partielles (ligne 13 et 14).
3. Une fois qu'on a traité un lot $\mathsf{b_i}$, l'on se sert des dérivées partielles précédemment accumulées pour mettre à jour les paramètres (ligne 14).
4. On évalue ensuite le modèle en calculant l'erreur de prédiction sur les données de validations (ligne 18)

1. **une époque** fait référence à un passage complet de l'ensemble du jeu de données d'entraînement à travers l'algorithme.

5. On arrête l'entraînement si le modèle n'a pas évolué au cours des quatre dernière époques ou alors si l'on a atteint le nombre d'époques maximale (ligne 7).

L'entraînement repose essentiellement sur l'exécution itérative de deux fonctions, la *forward* (ligne 12) et la *backforward* (ligne 14). La *forward* présentée à l'algorithme 2 désigne la phase de propagation avant qui a été présentée à la section 3.2 pour chacune des variantes de RNN (standard, GRU, LSTM). Elle prend en entré une phrase x constituée des mots $x_1, x_2, \ldots, x_n$, ainsi que l'ensemble de paramètres Θ du modèle, et retourne une distribution de probabilité $\mathsf{y_{pred}}$ permettant de déterminer la classe de la phrase x. La *backforward* désigne la phase de rétro propagation, où l'on calcule et renvoie

Algorithm 2 : **forward**

Input x : une phrase constitué des mots $x_1, x_2, ..., x_n$;
Θ : l'ensemble des paramètres du modèle
Output $\mathsf{y_{pred}}$: distribution de probabilité permettant de déterminer la classe de la phrase x
Start

```
1:  n ← LENGTH(x)
2:  h_0 ← 0                    //On initialise l'état caché h_0 à 0
3:  c_0 ← 0                    //On initialise la cellule mémoire c_0 à 0 pour un LSTM
4:  for t ← 1 to n do
5:      switch (RNN):
6:          case lstm:
7:              compute memory cell c_t // on calcul l'état c_t de la cellule mémoire en fonction de x_t et h_{t-1} comme à l'équation 3.7
8:              compute hiden state h_t // on calcul l'état caché h_t en fonction de c_t comme à l'équation 3.7
9:          case gru:
10:             compute hiden state h_t // on calcul l'état caché h_t en fonction de x_t et h_{t-1} comme à l'équation 3.9
11:         case standard:
12:             compute hiden state h_t // on calcul l'état caché h_t en fonction de x_t et h_{t-1} comme à l'équation 3.2
13:     end switch
14: end for
15: v ← mult(h_n, W_y) + b_y    //on utilise le dernier état caché h_n pour trouver y_pred
16: y_pred ← SoftMax(v)
17: Return y_pred
```

End

les dérivées partielles $\mathsf{d}\Theta$ de chaque paramètres Θ du modèle. L'ensemble des dérivées partielles s'obtient via l'algorithme de rétro-propagation à travers le temps. Le principe est de reparcourir le réseau de neurones en sens inverse (de l'instant t=n à l'instant t = 1), en appliquant à chaque fois la propriété mathématique de dérivation en chaîne. Des auteurs comme Chen et al. [3] fournissent un ensemble d'équations mathématique permettant d'obtenir ces dérivées. Les algorithmes 3, 4, 5 illustrent une implémentation de ces formules respectivement pour un RNN standard, un LSTM et un GRU.

Algorithm 3 : **backforward** (rnn standard)

Input x : une phrase constitué des mots $x_1, x_2, ..., x_n$; y : la classe réel, $\mathsf{y_{pred}}$: la classe prédite

Θ : l'ensemble des paramètres du modèle

Output $\mathsf{d\Theta}$: Ensemble de dérivées partielles des paramètres du modèle

Start

1: $\mathsf{n} \leftarrow$ LENGTH(x)
2: InitiazeToZero($\mathsf{d\Theta}$)
3: $\mathsf{db_y} \leftarrow \mathsf{y_{pred}} - \mathsf{y}$ // *on calcule la dérivée partielle du biais* $\mathsf{b_y}$ *de la couche de sortie*
4: $\mathsf{dW_y} \leftarrow \mathsf{mult}(\mathsf{db_y}, \mathsf{h_n})$ // *on calcule la dérivée partielle de la matrice de poids* $\mathsf{w_y}$ *de la couche de sortie*
5: $\mathsf{dh} \leftarrow \mathsf{mult}(\mathsf{db_y}, \mathsf{tr}(\mathsf{w_y}))$ // *on calcule la dérivée partielle de l'état caché* $\mathsf{h_t}$ *pour* $\mathsf{t} = \mathsf{n}$
6: **for** t $\leftarrow n$ **to** 1 **do**
7: $\quad\mathsf{cache} \leftarrow 1 - \mathsf{pow}(\mathsf{h_t}, 2)$
8: $\quad\mathsf{cache} \leftarrow \mathsf{dh} \odot \mathsf{cache}$
9: $\quad\mathsf{db_h} \leftarrow \mathsf{db_h} + \mathsf{cache}$ // *on calcul la dérivée des paramètres de la couche cachée*
10: $\quad\mathsf{dW_{hx}} \leftarrow \mathsf{dW_{hx}} + \mathsf{mult}(\mathsf{x_t},\ \mathsf{cache})$
11: $\quad\mathsf{dW_{hh}} \leftarrow \mathsf{dW_{hh}} + \mathsf{mult}(\mathsf{h_{t-1}},\ \mathsf{cache})$
12: $\quad\mathsf{dh} \leftarrow \mathsf{mult}(\mathsf{cache},\ \mathsf{W_{hh}})$ // *on met à jour la dérivée partielle de l'état caché* $\mathsf{h_t}$ *pour* $\mathsf{t} < \mathsf{n}$
13: **end for**
14: **Return** $\{\ \mathsf{dW_{hx}}, \mathsf{dW_{hh}}, \mathsf{db_h}, \mathsf{dW_y}, \mathsf{db_y}\ \}$

End

Algorithm 4 : **backforward** (lstm)

Input x : une phrase constitué des mots $x_1, x_2, ..., x_n$; y : la classe réel, $\mathsf{y_{pred}}$: la classe prédite

Θ : l'ensemble des paramètres du modèle

Output $\mathsf{d\Theta}$: Ensemble de dérivées partielles des paramètres du modèle

Start

1: $\mathsf{n} \leftarrow$ LENGTH(x)
2: compute $\mathsf{db_y}$ and $\mathsf{dW_y}$ like in algortihm 3
3: $\mathsf{dh} \leftarrow \mathsf{mult}(\mathsf{db_y}, \mathsf{tr}(\mathsf{w_y}))$
4: $\mathsf{dc} \leftarrow \mathsf{dh} \odot \mathsf{o_n} \odot (1 - \tanh(\mathsf{c_n}))$ // *on calcule la dérivée partielle de la cellule mémoire* $\mathsf{c_t}$ *pour* $\mathsf{t} = \mathsf{n}$
5: **for** t $\leftarrow n$ **to** 1 **do**
6: $\quad\mathsf{q_t} \leftarrow \mathsf{concat}(\mathsf{x_t}, \mathsf{h_{t-1}})$ // *on concatène les vecteurs* $\mathsf{x_t}$ *et* $\mathsf{h_t}$ $(\mathsf{q_t} = [\mathsf{x_t}, \mathsf{h_{t-1}}])$
7: $\quad\mathsf{do} \leftarrow \mathsf{dh} \odot \tanh(\mathsf{c_t})$ // *de la ligne 7 à ligne 9 on calcule les dérivées des portes output , forgate et input gate*
8: $\quad\mathsf{df} \leftarrow \mathsf{dc} \odot \mathsf{c_{t-1}}$
9: $\quad\mathsf{di} \leftarrow \mathsf{dc} \odot \overline{\mathsf{c_t}}$
10: $\quad\mathsf{d\overline{c}} \leftarrow \mathsf{dc} \odot \mathsf{i_t}$ // *on calcule la dérivée de la mémoire candidate* $\overline{c_t}$
11: $\quad\mathsf{dW_c} \leftarrow \mathsf{dW_c} + \mathsf{mult}(\mathsf{q_t},\ \mathsf{d\overline{c}} \odot (1 - \overline{\mathsf{c_t}}^2))$ // *de la ligne 11 à ligne 18 on calcule les dérivées des paramètres la couche cachée*
12: $\quad\mathsf{dW_i} \leftarrow \mathsf{dW_i}\ + \mathsf{mult}(\mathsf{q_t},\ \mathsf{di} \odot \mathsf{i_t} \odot (1 - \mathsf{i_t}))$
13: $\quad\mathsf{dW_f} \leftarrow \mathsf{dW_f} + \mathsf{mult}(\mathsf{q_t},\ \mathsf{df} \odot \mathsf{f_t} \odot (1 - \mathsf{f_t}))$
14: $\quad\mathsf{dW_o} \leftarrow \mathsf{dW_o} + \mathsf{mult}(\mathsf{q_t},\ \mathsf{do} \odot \mathsf{o_t} \odot (1 - \mathsf{o_t}))$
15: $\quad\mathsf{db_f} \leftarrow \mathsf{db_f} + \mathsf{df} \odot \mathsf{f_t} \odot (1 - \mathsf{f_t})$
16: $\quad\mathsf{db_i} \leftarrow \mathsf{db_i}\ + \mathsf{di} \odot \mathsf{i_t} \odot (1 - \mathsf{i_t})$
17: $\quad\mathsf{db_o} \leftarrow \mathsf{db_o} + \mathsf{do} \odot \mathsf{o_t} \odot (1 - \mathsf{o_t})$
18: $\quad\mathsf{db_c} \leftarrow \mathsf{db_c} + \mathsf{d\overline{c}} \odot (1 - \overline{\mathsf{c_t}}^2)$
19: $\quad\mathsf{dc} \leftarrow \mathsf{dc} \odot \mathsf{f_t}$ // *on met à jour la dérivée de* $\mathsf{c_t}$ *et celle de* $\mathsf{h_t}$ *pour* $\mathsf{t} < \mathsf{n}$
20: $\quad\mathsf{dh} \leftarrow \mathsf{mult}(\mathsf{do} \odot \mathsf{o_t} \odot (1 - \mathsf{o_t}),\ \mathsf{tr}(\mathsf{w_o})) + \mathsf{mult}(\mathsf{d\overline{c}} \odot (1 - \overline{\mathsf{c_t}}^2),\ \mathsf{tr}(\mathsf{w_c}))$
21: $\quad\mathsf{dh} \leftarrow \mathsf{dh} + \mathsf{mult}(\mathsf{df} \odot \mathsf{f_t} \odot (1 - \mathsf{f_t}),\ \mathsf{tr}(\mathsf{w_f})) + \mathsf{mult}(\mathsf{di} \odot \mathsf{i_t} \odot (1 - \mathsf{i_t}),\ \mathsf{tr}(\mathsf{w_i}))$
22: **end for**
23: **Return** $\{\mathsf{dW_i}, \mathsf{dW_f}, \mathsf{dW_c}, \mathsf{dW_o}, \mathsf{db_o}, \mathsf{db_f}, \mathsf{db_i}, \mathsf{db_c}, \mathsf{dW_y}, \mathsf{db_y}\}$

End

Algorithm 5 : **backforward** (gru)

Input x : une phrase constitué des mots $x_1, x_2, ..., x_n$; y : la classe réel, $\mathsf{y_{pred}}$: la classe prédite
Θ : l'ensemble des paramètres du modèle
Output $\mathsf{d\Theta}$: Ensemble de dérivées partielles des paramètres du modèle
Start

1: $\mathsf{n} \leftarrow$ LENGTH(x)
2: zeroInitialize($\mathsf{d\Theta}$) *//on initialise les dérivées partielles à zéro*
3: $\mathsf{db_y} \leftarrow \mathsf{y_{pred}} - \mathsf{y}$
4: $\mathsf{dW_y} \leftarrow \mathsf{mult}(\mathsf{db_y}, \mathsf{h_n})$
5: $\mathsf{dh} \leftarrow \mathsf{mult}(\mathsf{db_y}, \mathsf{tr}(\mathsf{w_y}))$ *// on calcule la dérivée partielle de l'état caché* $\mathsf{h_t}$ *pour* $\mathsf{t} = \mathsf{n}$
6: **for** $\mathsf{t} \leftarrow n$ **to** 1 **do**
7: $\mathsf{s_t} \leftarrow \mathsf{concat}(\mathsf{x_t},\ \mathsf{r_t} \odot \mathsf{h_{t-1}})$ *// on concatène les vecteurs* $\mathsf{x_t}$ *et* $\mathsf{h_t}$ $(\mathsf{s_t} = [\mathsf{x_t}, \mathsf{r_t} \odot \mathsf{h_{t-1}}])$
8: $\mathsf{q_t} \leftarrow \mathsf{concat}(\mathsf{x_t}, \mathsf{h_t})$
9: $\mathsf{d\overline{h}} \leftarrow \mathsf{dh} \odot (1 - \mathsf{z_t})$
10: $\mathsf{dz} \leftarrow \mathsf{dh} \odot (\mathsf{h_{t-1}} - \overline{\mathsf{h_t}})$ *// de la ligne 10 à ligne 11 on calcule les dérivées des portes reset et update gate*
11: $\mathsf{dr} \leftarrow \mathsf{d\overline{h}} \odot \mathsf{mult}(\mathsf{W_h},\ (1 - \overline{\mathsf{h_t^2}}))$
12: $\mathsf{dW_h} \leftarrow \mathsf{dW_h} + \mathsf{mult}(\mathsf{s_t},\ \mathsf{d\overline{h}} \odot (1 - \overline{\mathsf{h_t^2}}))$ *// ligne 12 à ligne 17 on calcule les dérivées des poids de la couche cachée*
13: $\mathsf{dW_r} \leftarrow \mathsf{dW_r}\ + \mathsf{mult}(\mathsf{q_t},\ \mathsf{dr} \odot \mathsf{r_t} \odot (1 - \mathsf{r_t}))$
14: $\mathsf{dW_z} \leftarrow \mathsf{dW_z}\ + \mathsf{mult}(\mathsf{q_t},\ \mathsf{dz} \odot \mathsf{z_t} \odot (1 - \mathsf{z_t}))$
15: $\mathsf{db_r} \leftarrow \mathsf{db_r} + \mathsf{dr} \odot \mathsf{r_t} \odot (1 - \mathsf{r_t})$
16: $\mathsf{db_z} \leftarrow \mathsf{db_z} + \mathsf{dz} \odot \mathsf{z_t} \odot (1 - \mathsf{z_t})$
17: $\mathsf{db_h} \leftarrow \mathsf{db_h} + \mathsf{d\overline{h}} \odot (1 - \overline{\mathsf{h_t^2}})$
18: $\mathsf{dh} \leftarrow \mathsf{mult}(\mathsf{d\overline{h}} \odot (1 - \overline{\mathsf{h_t^2}}),\ \mathsf{tr}(\mathsf{w_h}) + \mathsf{mult}(\mathsf{dz} \odot \mathsf{z_t} \odot (1 - \mathsf{z_t}),\ \mathsf{tr}(\mathsf{w_z}))$
19: $\mathsf{dh} \leftarrow \mathsf{dh} + \mathsf{mult}(\mathsf{dr} \odot \mathsf{r_t} \odot (1 - \mathsf{r_t}),\ \mathsf{tr}(\mathsf{w_r}))$ *// on met à jour la dérivée partielle de l'état caché* $\mathsf{h_t}$ *pour* $\mathsf{t} < \mathsf{n}$
20: **end for**
21: **Return** $\{\mathsf{dW_z}, \mathsf{dW_r}, \mathsf{dW_h}, \mathsf{db_r}, \mathsf{db_z}, \mathsf{db_h}, \mathsf{dW_y}, \mathsf{db_y}\}$
End

En considérant l'exécution itérative des fonctions *forward* et *backforward*, et en posant : $\mathsf{N_e}$ le nombre maximal d'époque, W le nombre d'éléments du jeu données, $\mathsf{N_h}$ le nombre de neurones dans la couche cachée, $\mathsf{N_v}$ la taille du vecteur d'entrée représentant un mot, $\mathsf{N_s}$ la taille du vecteur de sortie, et S la taille maximale d'une phrase, alors la complexité en temps dans le pire des cas pour un réseaux de neurones récurrent standard est présentée à l'équation 3.10, celle d'un LSTM à l'équation 3.11 et celle d'un GRU à l'équation 3.12.

$$\mathcal{O}(\mathsf{WN_e}(\mathsf{N_s N_h} + \mathsf{S N_h}(\mathsf{N_h} + \mathsf{N_v})) \tag{3.10}$$

$$\mathcal{O}(\mathsf{WN_e}(\mathsf{N_s N_h} + \mathsf{S}(\mathsf{N_v} + \mathsf{N_h}(\mathsf{N_h} + \mathsf{N_v}))) \tag{3.11}$$

$$\mathcal{O}(\mathsf{WN_e}(\mathsf{N_s N_h} + \mathsf{S}(\mathsf{N_v} + \mathsf{N_h}(\mathsf{N_h} + \mathsf{N_v}))) \tag{3.12}$$

En observant les complexités précédentes, l'on remarque que le temps d'exécution des algorithmes d'entraînement est proportionnel à la taille W du jeu de données. Nous pro-

posons dans la suite une stratégie d'implémentation parallèle tenant compte de cette observation pour réduire le temps d'exécution.

3.2.2 Stratégie de parallélisation

La résolution d'un problème de classification de texte suit un certain nombre d'étapes communes aux problèmes de machine learning (voir figure 3.8). Il est donc souvent nécessaire d'effectuer de façon répétitive l'étape de modélisation (l'entraînement).

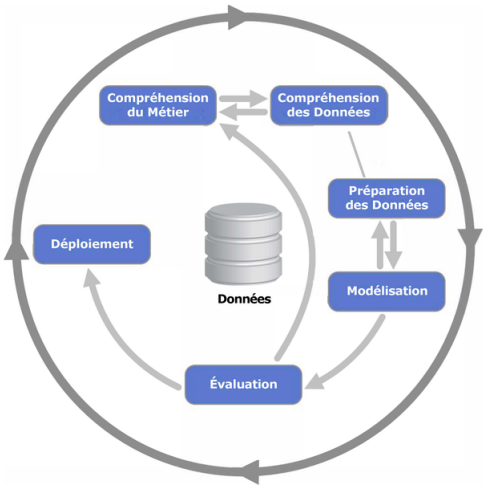

FIGURE 3.8 – *Cross-Industry Standard Process for Data Mining* (CRISP-DM) [31]. Ensemble d'étapes à suivre pour résoudre un problème en machine learning

Pour cette étape d'entraînement, la descente du gradient stochastique (SGD) est la procédure d'optimisation la plus utilisée en deep learning. Malheureusement, sa formulation traditionnelle est intrinsèquement séquentielle ce qui rend difficile son utilisation avec de très grands volume de données [7]. Cela est dû au fait que le temps nécessaire pour parcourir les données de manière entièrement séquentielle peut être prohibitif. Pour le cas particulier des RNNs, l'algorithme de rétro-propagation à travers le temps rend le processus d'entraînement encore plus lent, ce qui implique un temps d'exécution plus élevé [22]. Une solution efficace pour réduire le temps d'entraînement d'un réseau de neurones, est l'usage de la programmation parallèle.

L'approche principale pour paralléliser l'entraînement d'un réseau de neurones consiste à distribuer le calcul des dérivées sur plusieurs unités de traitement en exploitant la parallélisation au niveau des données [2]. En suivant cette approche, nous proposons dans

ce mémoire une stratégie d'implémentation parallèle effectuant une synchronisation logicielle mutex entre les unités de traitements pour la mise à jour du modèle global. Le principe est que chaque unité de traitement est responsable de la formation d'une copie du modèle sur un sous-ensemble de données (voir figure 3.9). Les unités de traitement utilisent leurs copies locales du modèle pour calculer les dérivées (∇P) des paramètres. Après le calcul des dérivées, chaque unité de traitement l'un après l'autre entre en section critique à l'aide d'une variable mutex pour mettre à jour les paramètres du modèle global accessible depuis une mémoire partagée.

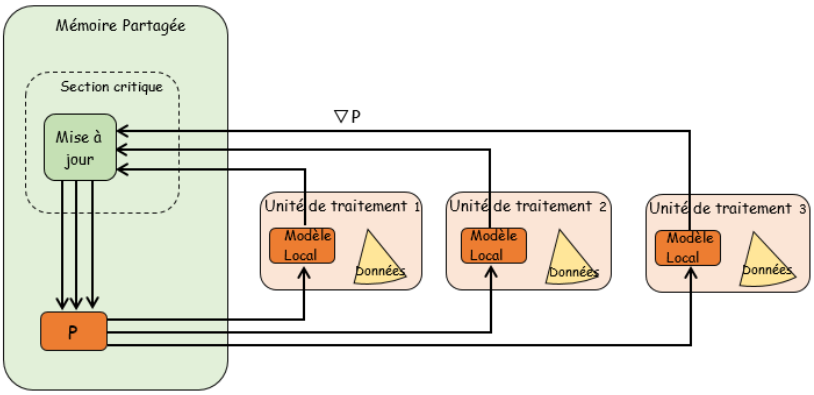

FIGURE 3.9 – Parallélisation avec synchronisation mutex. Chaque partie de données est affectée à une unité de traitement qui calcule les dérivées (∇P), et entre en section critique pour mettre à jour le modèle global.

L'intuition de cette approche est que, même si l'on considère le temps de nécessaire pour la synchronisation entre les unités de traitement, le fait d'effectuer simultanément l'entraînement sur plusieurs unités traitements avec des quantités de données réduites devrait tout de même permettre de diminuer le temps d'exécution. Nous présenterons dans la suite l'algorithme d'entraînement parallèle respectant cette approche et nous ferons une analyse de cet algorithme par rapport à la version séquentielle.

3.2.3 Algorithme parallèle

Nous présentons ici l'algorithme d'entraînement parallèle qui prend en compte la stratégie de parallélisation précédemment présentée. L'implémentation a été effectuée dans un environnement à mémoire partagée et doté d'une architecture multi-cœurs. Les unités de traitement correspondent alors aux différents cœurs d'un processeur. Chaque coeur, à

travers un *thread*[2] exécute simultanément aux autres *threads* un ensemble d'instructions. Le thread principal exécute l'algorithme 6. Il est chargé d'initialiser les paramètres du modèle global (ligne 1). À chaque époque, il partitionne les données en P sous ensemble et lance ensuite l'exécution des threads esclaves (ligne 6 à 10).

Algorithm 6 : **ParallelTraining**

Input (X,Y) : ensemble de phrase x et leur label y ; λ : le learning rate ;
M : le batch size ; epoch : le nombre d'époques ; P : le nombre de thread
Output Θ : Ensemble de paramètres du modèle
Start

1: initializeParameters(Θ)
2: B $\leftarrow$ getBatch(M,xtrain,ytrain)
3: e $\leftarrow$ 1
4: stop $\leftarrow$ 0
5: valLoss $\leftarrow \infty$
6: **while** $e \leq$ epoch **and** stop < 4 **do**
7: partition the data (X,Y) into P part d_i
8: **for** each d_i **do**
9: ParametersCopy(Θ, Θ_i) *// chaque thread fera une copie local du modèle global*
10: ThreadCode(P_i, Θ_i, λ, M) *// fonction exécutée par chaque thread*
11: **end for**
12: on évalue le modèle comme à l'algorithme 1 (ligne 17 à 21)
13: e $\leftarrow$ e + 1
14: **end while**
15: Return Θ

End

Algorithm 7 : **ThreadCode**

Input λ : le learning rate ; M : le batch size ; Θ_i : paramètres du modèle local, d_i : partion des données
Output
Start

1: B $\leftarrow$ getBatch(M,$\mathsf{d_i}$)
2: **for** $b_i \in B$ **do**
3: loss $\leftarrow$ 0
4: zeroInitialize($\mathsf{d\Theta}$)
5: **for** $(x, y) \in b_i$ **do**
6: $\mathsf{y_{pred}} \leftarrow \mathsf{forward}(\mathsf{x}, \Theta_\mathsf{i})$ *// pour chaque phrase x d'un lot de donnée, l'on prédit sa classe*
7: $\mathsf{loss} \leftarrow \mathsf{loss} + \mathsf{lossEntropy}(\mathsf{y}, \mathsf{y_{pred}})$ *// on accumule l'erreur de prédiction*
8: $\mathsf{d\Theta_i} \leftarrow \mathsf{d\Theta_i} + \mathsf{backforward}(\Theta_\mathsf{i}, \mathsf{x}, \mathsf{y}, \mathsf{y_{pred}})$ *//on calcul et on accumule les dérivées partielles des paramètres*
9: **end for**
10: **lockmutex_update**
11: update($\Theta, \mathsf{d\Theta_i}, \lambda, \mathsf{M}$) *//mise à jour des paramètres Θ du modèle global en section critique*
12: ParametersCopy($\Theta, \Theta_\mathsf{i}$) *// le thread local récupère les paramètres mise à jour du modèle global*
13: **freemutex_update**
14: **end for**

End

2. **un thread** fait référence à un seul flux séquentiel d'instructions en cours d'exécution dans un processus

Chaque thread esclave exécute l'algorithme 7 simultanément aux autres threads avec une copie locale du modèle et un sous ensemble du jeu de données. Ils exécutent localement la phase forward (ligne 6) et la backforward (ligne 8). Une fois qu'un thread esclave termine le calcul des dérivées, celui-ci entre en section critique à l'aide d'une variable mutex pour mettre à jour le modèle global accessible depuis la mémoire partagée. Il récupère ensuite les paramètres mis à jour pour les prochaines itérations (ligne 10 à 12). Pour finir, il libère la variable mutex (ligne 13) pour permettre à un autre thread d'effectuer la mise à jour.

Complexité : Soit $\hat{\mathsf{I}}$ l'indice du sous ensemble de données ayant la plus grande taille ($|\mathsf{d}_{\hat{\mathsf{I}}}|$). Alors, la complexité en temps dans le pire des cas pour un RNN standard est présentée à l'équation 3.13, et celle d'un RNN de type LSTM et GRU est présentée à l'équation 3.14

$$\mathcal{O}(|\mathsf{d}_{\hat{\mathsf{I}}}|\mathsf{N_e}(\mathsf{N_s}\mathsf{N_h} + \mathsf{D}\mathsf{N_h}(\mathsf{N_h} + \mathsf{N_e})) \tag{3.13}$$

$$\mathcal{O}(|\mathsf{d}_{\hat{\mathsf{I}}}|\mathsf{N_e}(\mathsf{N_s}\mathsf{N_h} + \mathsf{D}(\mathsf{N_e} + \mathsf{N_h}(\mathsf{N_h} + \mathsf{N_e}))) \tag{3.14}$$

***Speedup* maximal** : En considérant le même nombre d'époque maximal $\mathsf{N_e}$ utilisé pour la version séquentielle, le *speedup* maximal pour cette implémentation parallèle correspond au nombre d'unité de traitement (P) utilisé. Il est en pratique difficile d'avoir un *speedup* très proche du *speedup* maximal, cela est due au temps de synchronisation entre les unités de traitements mais aussi aux portions de code s'exécutant de façon séquentielle pour le thread central.

Convergence : La convergence du modèle lors de l'entraînement avec cette d'implémentation parallèle devrait être proche de celle observée avec une implémentation séquentielle. Il est possible de le montrer de façon formelle en s'inspirant par exemple des travaux de Feng Niu et al. [23] dans le cadre de la parallélisation d'un SVM sparse.

3.2.4 Comparaisons aux approches existantes

Nous présentons ici quelques similitudes et différences entre l'approche de parallélisation présentée dans ce mémoire et les travaux existants présentés à la section 2.2. Tout d'abord, Jeffrey Dean et al. dans leurs travaux combinent la parallélisation du modèle et la parallélisation asynchrone basée sur les données. Ils proposent l'algorithme Down-Pour SGD, où chaque unité de traitement est responsable de l'entraînement d'un sous

ensemble de paramètres du modèles en opposition avec la nôtre où chaque unité de traitement effectue l'entraînement sur l'ensemble des paramètres du modèle (voir figure 3.10).

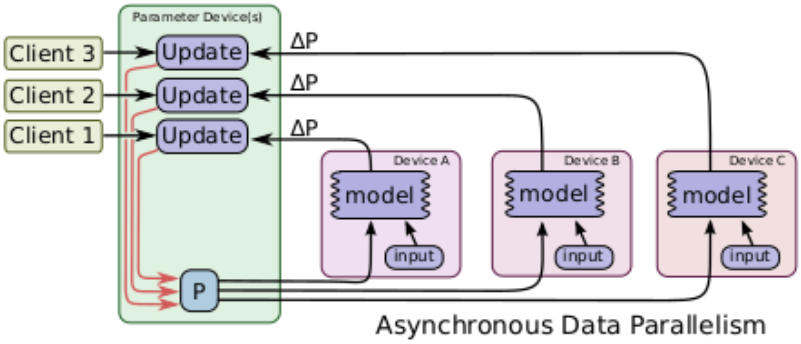

FIGURE 3.10 – Descente du gradient Parallèle Asynchrone [1]. Chaque unité de traitement est responsable de l'entraînement d'un sous ensemble de paramètres du modèle avec un sous ensemble du jeux de données. Cette approche est asynchrone car les unités de traitements effectuent les mises de façon indépendante.

Cet algorithme est plus approprié pour être implémenté dans un environnement à mémoire distribuée pour des modèles très volumineux (avec des milliards de paramètres). Il n'est pas très utilisé parce qu'il est difficile à mettre en place, cela est dû au fait qu'il nécessite généralement une conception délicate pour la connexion entre les appareils et pour la partition du modèle.

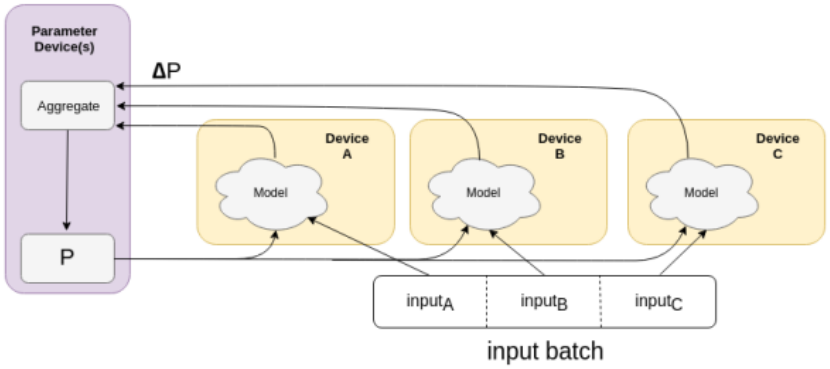

FIGURE 3.11 – Descente du gradient Parallèle Synchrone [1]. Chaque unité de traitement utilise une copie locale du modèle et un sous ensemble de données pour calculer les gradients (∇P). Une étape de synchronisation agrège tous les gradients partiels. Le résultat de l'agrégation est ensuite utilisé pour mettre à jour les paramètres (P) du modèle global. Les paramètres mises à jour sont ensuite redistribués aux unités de traitements.

La stratégie que nous proposons est plus proche de celle présentée par Martin Abadi et al. [1] qui dans leurs travaux présentent l'algorithme Synchronous Parallel SGD (S-PSGD). La principale différence avec la notre est que, pour leur approche, les différents gradients calculés par chaque unité de traitement sont ensuite **agrégés** pour mettre à jour les paramètres P du modèle global (voir figure 3.11).

Zhiheng Huang et al. [12] dans leurs travaux utilisent une approche similaire pour l'entraînement d'un RNN pour une tâche de modélisation de langage. La principale différence avec l'algorithme S-PSGD est que, après le calcul des gradients, les unités de traitements mettent directement à jour leurs modèles locaux. Dans ce cas, l'étape de synchronisation après chaque époque, n'agrégera plus les dérivés (∇P) mais plutôt les paramètres des différents modèles locaux. Cette approche permet de réduire le temps de synchronisation entre les unités de traitement. Cependant, notre intuition est que, l'agrégation (la moyenne arithmétique par exemple) s'effectue avec perte ce qui peut contribuer à réduire la qualité de la convergence du modèle global. Pour ce mémoire, nous avons considéré une synchronisation sans agrégation. Dans la section qui suit, nous présenterons les différences de performances entre notre approche et celle de Zhiheng Huang et al.

Chapitre 4

Expérimentations et Résultats

Dans ce chapitre, nous présenterons tout d'abord notre cadre expérimental. Nous présenterons ensuite les résultats expérimentaux obtenus pour l'exécution séquentielle en se servant des mesures métriques de performances telles que le rappel, la précision, et la f-mesure. Pour finir nous présenterons les résultats de l'exécution parallèle en utilisant des mesures de performances telles que le *speedup* et le temps d'exécution.

4.1 Cadre expérimental

Pour ces résultats préliminaires, les expérimentations ont été faites sur une machine multi-cœur ayant les caractéristiques suivantes : **8 cœurs à 2,20 GHz**, 8 Go de Ram, L3 de 6144 Ko, L2 de 256 Ko et L1 de 32 Ko. Les implémentations des algorithmes d'entraînement ont été effectuées en langage C et en utilisant la bibliothèque *posix thread* pour les implémentations parallèles. Les opérations de pré-traitement sur les données ont été faites avec le langage python et en utilisant des bibliothèques telles que NLTK [1](*Natural Language Tools Kit*) et *gensim* [2].

Pour les expérimentations, nous avons utilisé deux jeux de données. Le premier jeu de données (*dataset 1*) est celui que nous avons obtenu en effectuant du *scraping* sur des pages facebook d'utilisateurs Camerounais. L'objectif était d'obtenir un jeu de données en français et propre au contexte camerounais. Ce jeu de données contient au final 3110 commentaires non haineux et 3069 commentaires haineux. Le second jeu de données, (*dataset 2*) obtenu sur Kaggle [3] contient un ensemble de tweets en langue anglaise. Il est initialement non équilibré avec 77% de tweets offensif, 7% de tweets haineux et 16% de

1. https ://www.nltk.org/
2. https ://radimrehurek.com/gensim/models/word2vec.html
3. https ://www.kaggle.com/datasets/mrmorj/hate-speech-and-offensive-language-dataset

tweets ni offensifs ni haineux. Nous avons alors effectué un sous échantillonage pour le rendre équilibré. On obtient finalement un jeu de données avec 5190 tweets dont 1880 tweets offensifs, 1430 tweets haineux, 1880 tweets ni offensif ni haineux. Au deux jeux de données, nous avons appliqué un ensemble d'opérations de pré-traitement à savoir :

- **Cleaning** : qui consiste à supprimer du jeu de données tous les caractères non alphanumériques.
- **Suppression de stopwords** : dont l'objectif est de supprimer l'ensemble de mot (stopwords) n'ayant aucune influence discriminatoire et ne sont donc pas utiles lors l'apprentissage.
- **Tonkenisation et représentation vectorielle** : la tokenisation consiste à transformer un texte en une liste de mots dépourvus de séparateurs. La représentation vectorielle consiste à représenter les tokens en vecteur numérique comme nous l'avons précédemment expliqué.

Après avoir été pré-traités, les deux jeux de données ont ensuite été divisés en deux, à savoir 80% pour l'entraînement, et 20% pour le test et la validation.

4.2 Exécution séquentielle

Les valeurs d'hyperparamètres utilisées pour l'exécution de l'algorithme d'apprentissage pour chacune des variantes du RNN sont les suivantes :

- Le nombre d'époque maximale : 15
- Le learning rate λ : 0.1
- Le batch size M : 32
- Le nombre total de neurones $\mathsf{N_h}$ dans la couche cachée : 80.
- La taille $\mathsf{N_v}$ des vecteurs d'entrée représentant un mot : 30

Étant donné que nous effectuons une tâche de classification de texte et que les jeux de données utilisés sont équilibrés, les mesures de performance métrique adéquates pour évaluer le modèle sont les moyennes non pondérées : de la précision, du rappel, et de la f-mesure de chacune des classes. Le tableau 4.1 présente les résultats obtenus pour les deux jeux de données ainsi que le temps d'entraînement pris pour la formation du modèle avec chacune des variantes du RNN.

	Précision	Rappel	F-mesure	Temps d'entraînement (s)
$\text{LSTM}_{\text{dataset2}}$	**0.728**	**0.715**	**0.716**	**1970**
$\text{GRU}_{\text{dataset2}}$	0.718	0.710	0.702	1555
$\text{RnnStandard}_{\text{dataset2}}$	0.605	0.567	0.549	540
$\text{LSTM}_{\text{dataset1}}$	**0.528**	**0.533**	**0.530**	**4361**
$\text{GRU}_{\text{dataset1}}$	0.528	0.525	0.511	3134
$\text{RnnStandard}_{\text{dataset1}}$	0.239	0.500	0.354	1065

TABLE 4.1 – Tableau présentant les mesures métriques pour chacune des variantes de RNN ainsi que le temps pris pour l'entraînement du modèle

En observant le tableau 4.1, nous constatons que l'on obtient à chaque fois des modèles ayant de meilleurs performances lorsqu'on utilise un LSTM avec l'un ou l'autre des jeux de données, ensuite ceux obtenus à partir d'un GRU, et enfin ceux qui sont moins bon obtenus à partir d'un RNN standard. Cependant, le fait d'utiliser un LSTM implique un temps d'entraînement plus élevé par rapport aux autres variantes de RNN. Nous avons alors implémenté l'algorithme d'entraînement parallèle d'un LSTM en espérant réduire le temps d'exécution tout en conservant les mesures métriques. L'on constate aussi que le modèle obtenu avec le dataset 1 n'est pas assez performant, cela est certainement dû à la qualité de notre jeu de données. Il ne permet pas au réseau de neurones de trouver une corrélation entre les commentaires classifiés haineux ou non haineux. La cause serait le processus de collecte et d'étiquetage de données qui n'a pas été effectué de façon optimale. Nous comptons dans la suite de nos travaux améliorer la qualité de ce jeu de données dans l'optique d'avoir des données appropriées et propres au contexte local.

4.3 Exécution parallèle

Afin d'effectuer des comparaisons, nous avons utilisé deux stratégies d'implémentations parallèles pour l'algorithme d'entraînement d'un LSTM avec le *dataset 2*. La première (parallèle sans agrégation) est l'idée de parallélisation que nous proposons dans ce mémoire. La seconde (parallèle avec agrégation) est celle effectuant une agrégation (dans ce cas la moyenne arithmétique) des différents modèles locaux pour avoir le modèle global final. La figure 4.1 présente le temps d'exécution et le *speedup* en fonction du nombre de *threads* (cœurs) utilisés pour chacune des deux stratégies.

En observant la figure 4.1, l'on remarque que le gain en temps et le *speedup* est presque le même pour chacune des 2 stratégies. En utilisant les huit coeurs disponibles, l'on obtient un *speedup* de 2,21 pour la stratégie sans agrégation, et un *speedup* de 2,19 pour la stratégie avec agrégation.

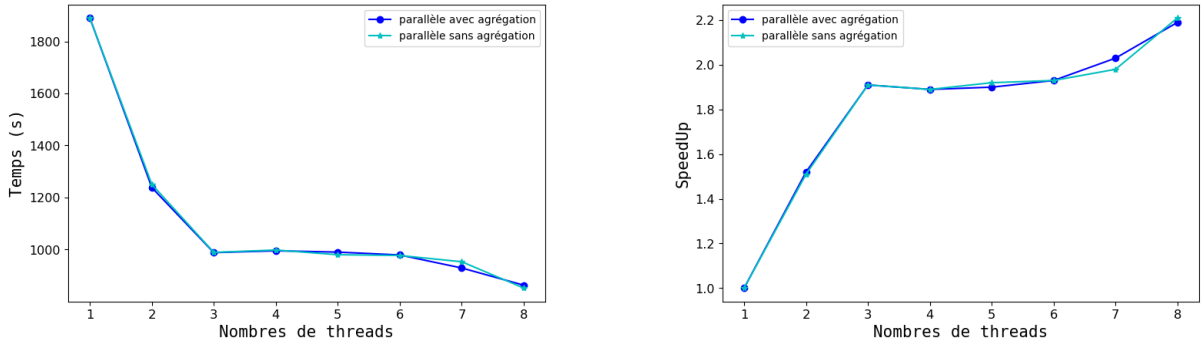

FIGURE 4.1 – *speedup* et temps d'exécution pour l'entraînement d'un LSTM en fonction du nombre de thread (nombre d'unité de traitements) avec le ***dataset 2***

Notons que pour la stratégie de parallélisation avec agrégation, il est possible d'utiliser une fonction d'agrégation plus complexe que la moyenne arithmétique, pour par exemple garantir une meilleure convergence, tout en prenant le risque que le gain en temps soit moins important.

Nous nous sommes aussi intéressés à l'évolution de la convergence du modèle pour chacune des deux stratégies à l'aide des graphes présentant l'évolution de l'erreur d'entraînement en fonction du nombre d'époques (voir figure 4.2). En examinant ces graphes, l'on constate que le modèle converge mieux avec la stratégie de parallélisation sans agrégation par rapport à celle effectuant la moyenne arithmétique des différents modèles locaux. L'on remarque aussi que le nombre d'unités de traitement utilisé impacte plus négativement la stratégie avec agrégation. L'on peut par exemple voir qu'avec huit unités de traitement, la convergence du modèle en utilisant cette stratégie est largement moins bonne que celle avec deux unités de traitement.

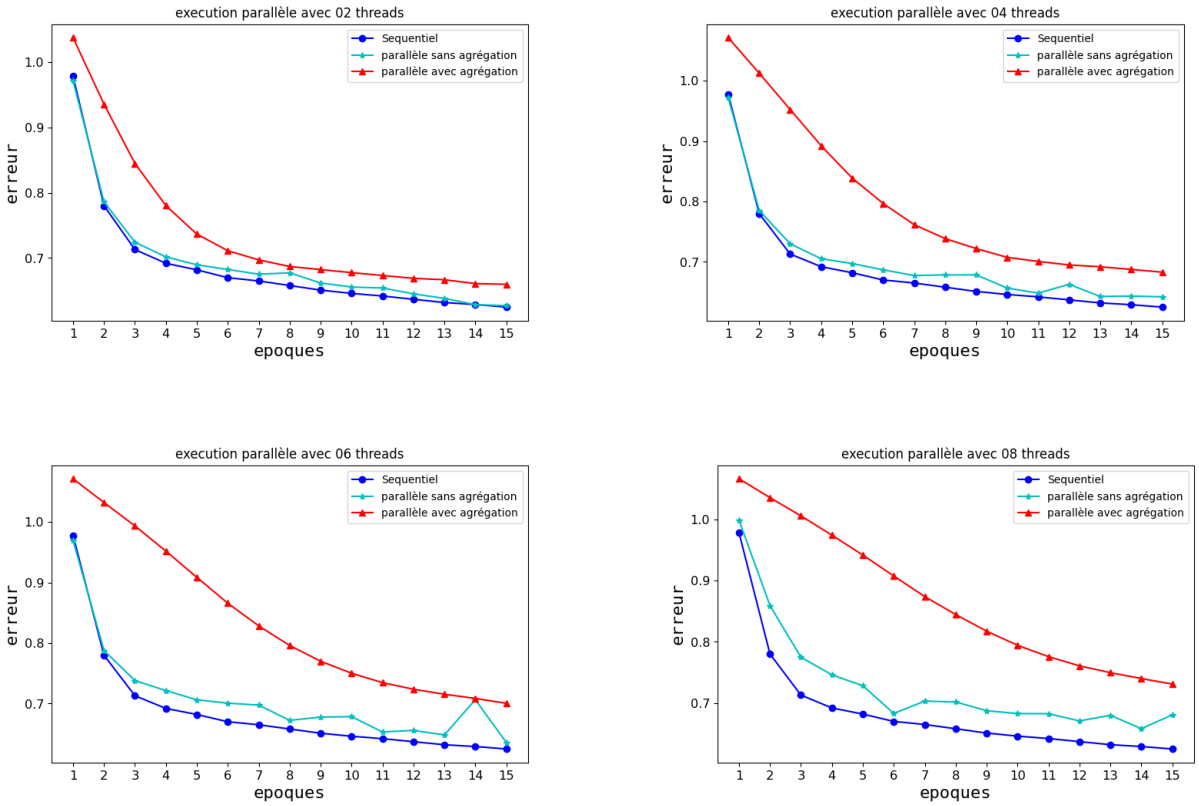

FIGURE 4.2 – Évolution de l'erreur sur les données d'entraînements pour un LSTM en utilisant le ***dataset 2***

Cela est certainement dû à l'un des désavantages de la moyenne arithmétique lorsqu'elle est utilisée comme fonction d'agrégation. Comme par exemple le fait que, la moyenne est très affectée par des valeurs extrêmes différentes de l'ensemble. Le tableau 4.2 présente les mesures métriques de précision obtenues avec chacune des deux stratégies de parallélisation en utilisant les huit unités de traitement disponible, et en comparaison avec celles obtenues avec l'exécution séquentielle. En observant le tableau 4.2, l'on peut conclure sur le fait qu'il serait plus intéressant d'utiliser une stratégie de parallélisation sans agrégation, par rapport à celle effectuant une agrégation entre les modèles locaux des différentes unités de traitement. L'une des principales difficulté ou inconvénient des stratégies utilisant une agrégation est le fait de devoir choisir la méthode ou la fonction d'agrégation adéquate garantissant une bonne convergence et un gain en temps.

		Précision	Rappel	F-mesure	Temps d'entraînement (s)
LSTM	séquentielle	0.728	0.715	0.716	1970
	parallèle sans agrégation	**0.726**	**0.699**	**0.700**	**889**
	parallèle avec agrégation	0.690	0.654	0.651	897
GRU	séquentielle	0.718	0.710	0.702	1555
	parallèle sans agrégation	**0.686**	**0.677**	**0.678**	**722**
	parallèle avec agrégation	0.668	0.652	0.651	730
Standard	séquentielle	0.605	0.567	0.549	540
	parallèle sans agrégation	**0.616**	**0.549**	**0.527**	**230**
	parallèle avec agrégation	0.587	0.522	0.504	227

TABLE 4.2 – Tableau présentant les mesures métriques de précision pour les implémentations parallèle (avec 08 threads) ainsi que le temps pris pour l'entraînement du modèle

Chapitre 5

Conclusion et perspectives

Nous présenterons dans ce chapitre un résumé des objectifs de nos travaux, les résultats préliminaires que nous avons obtenus, ainsi que des directives futures.

5.1 Conclusion

L'objectif de ce mémoire était d'implémenter de façon parallèle l'algorithme d'entraînement d'un RNN pour la détection des messages haineux. Pour le faire, nous avons tout d'abord présenté de façon générale le principe et fonctionnement de trois variantes de RNN à savoir : RNN standard, LSTM et GRU. Nous avons ensuite procédé à une exécution séquentielle en utilisant deux jeux de données. Un jeu de données propre au contexte du Cameroun que nous avons nous même collectés en effectuant du *scraping* sur des pages facebook d'utilisateurs camerounais, et un autre jeu de données obtenu sur Kaggle. Pour les deux jeux de données, l'on a obtenu des modèles avec de meilleures performances en utilisant un RNN de type LSTM à savoir : une f-mesure de 0,53 pour notre jeu de données et une f-mesure de 0.71 pour le jeu de donnée obtenu sur Kaggle. Nous avons ensuite procédé à une implémentation et exécution parallèle de l'algorithme d'entraînement d'un LSTM en utilisant le jeu de données obtenu sur Kaggle. Nous avons comparé deux stratégies d'implémentation parallèle, à savoir : une implémentation parallèle sans agrégation que nous avons proposée et présentée, et une autre effectuant une agrégation (la moyenne arithmétique) entre les unités de traitement. L'on a obtenu de meilleures performances en utilisant notre approche à savoir une f-mesure de 0,70 avec un *speedup* de 2,2.

5.2 Perspectives

Nous comptons dans la suite de nos travaux établir de façon plus formelle la différence entre l'approche proposée et celles existantes. Nous comptons aussi explorer l'influence de la manière dont on distribue le jeu de données entre les unités de traitements. Nous envisageons d'améliorer le jeu de données propre au contexte camerounais que nous sommes en train de construire. Nous souhaitons aussi continuer nos expérimentations en explorant d'autres stratégies d'implémentation parallèle, et en utilisant plus de ressources matérielles et des jeux de données plus volumineux. Il serait aussi intéressant d'utiliser et de comparer d'autres algorithmes de deep learning à l'instar des transformers, qui ont du succès en ce moment surtout pour des tâches de NLP.

Bibliographie

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow : Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv :1603.04467*, 2016.

[2] Ioan Budea, Peter Pietzuch, and Holger Pirk. Tensorbow : Supporting small-batch training in tensorflow. 2019.

[3] Gang Chen. A gentle tutorial of recurrent neural network with error backpropagation. *arXiv preprint arXiv :1610.02583*, 2016.

[4] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv :1406.1078*, 2014.

[5] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv :1412.3555*, 2014.

[6] Selima Curci, Decebal Constantin Mocanu, and Mykola Pechenizkiyi. Truly sparse neural networks at scale. *arXiv preprint arXiv :2102.01732*, 2021.

[7] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, et al. Large scale distributed deep networks. *Advances in neural information processing systems*, 25, 2012.

[8] Marco Dinarelli and Isabelle Tellier. Improving recurrent neural networks for sequence labelling. *arXiv preprint arXiv :1606.02555*, 2016.

[9] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2) :179–211, 1990.

[10] Michael J Flynn. Some computer organizations and their effectiveness. *IEEE transactions on computers*, 100(9) :948–960, 1972.

[11] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8) :1735–1780, 1997.

[12] Zhiheng Huang, Geoffrey Zweig, Michael Levit, Benoit Dumoulin, Barlas Oguz, and Shawn Chang. Accelerating recurrent neural network training via two stage classes and parallelization. In *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 326–331. IEEE, 2013.

[13] Michael I Jordan. Serial order : A parallel distributed processing approach. In *Advances in psychology*, volume 121, pages 471–495. Elsevier, 1997.

[14] Bekir Karlik and A Vehbi Olgac. Performance analysis of various activation functions in generalized mlp architectures of neural networks. *International Journal of Artificial Intelligence and Expert Systems*, 1(4) :111–122, 2011.

[15] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv :1301.3781*, 2013.

[16] Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. Advances in pre-training distributed word representations. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC 2018)*, 2018.

[17] mincom.gov.cm. Montee des discours de haine et les mesures preconisees par le gouvernement pour y faire face. [En ligne ; Page disponible le 29-juin-2023].

[18] Thomas Paine, Hailin Jin, Jianchao Yang, Zhe Lin, and Thomas Huang. Gpu asynchronous stochastic gradient descent to speed up neural network training. *arXiv preprint arXiv :1312.6186*, 2013.

[19] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International conference on machine learning*, pages 1310–1318. PMLR, 2013.

[20] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove : Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[21] Georgios K Pitsilis, Heri Ramampiaro, and Helge Langseth. Effective hate-speech detection in twitter data using recurrent neural networks. *Applied Intelligence*, 48 :4730–4742, 2018.

[22] T Rauber and G Rünger. Parallel programming : For multicore and cluster systems.[sl] : Springer science & business media, 2013. *Citado na*, page 30, 2013.

[23] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild ! : A lock-free approach to parallelizing stochastic gradient descent. *Advances in neural information processing systems*, 24, 2011.

[24] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv :1609.04747*, 2016.

[25] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088) :533–536, 1986.

[26] Arum Sucia Saksesi, Muhammad Nasrun, and Casi Setianingsih. Analysis text of hate speech detection using recurrent neural network. In *2018 International Conference on Control, Electronics, Renewable Energy and Communications (ICCEREC)*, pages 242–248. IEEE, 2018.

[27] Robin M Schmidt. Recurrent neural networks (rnns) : A gentle introduction and overview. *arXiv preprint arXiv :1912.05911*, 2019.

[28] Diksha Sharma and Neeraj Kumar. A review on machine learning algorithms, tasks and applications. *International Journal of Advanced Research in Computer Engineering & Technology (IJARCET)*, 6(10) :2278–1323, 2017.

[29] wearesocial.com. Digital 2022 global overview report, 2022. [En ligne ; Page disponible le 09-juillet-2023].

[30] wearesocial.com. Apprentissage par renforcemen, 2023. [En ligne ; Page disponible le 09-juillet-2023].

[31] Wikipédia. Cross industry standard process for data mining — wikipédia, l'encyclopédie libre, 2022. [En ligne ; Page disponible le 30-juin-2022].