Contribution au développement de l'apprentissage profond dans les systèmes distribués

Corentin Hardy

▶ To cite this version:

Corentin Hardy. Contribution au développement de l'apprentissage profond dans les systèmes distribués. Intelligence artificielle [cs.AI]. Université de Rennes, 2019. Français. NNT : 2019REN1S020 . tel-02284916

HAL Id: tel-02284916
https://theses.hal.science/tel-02284916

Submitted on 12 Sep 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE UNIVERSITE BRETAGNE LOIRE

Ecole Doctorale N°601
Mathématique et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par
« Corentin HARDY »

« Contribution au développement de l'apprentissage profond dans les systèmes distribués »

Thèse présentée et soutenue à Rennes, le 8 avril 2019
Unité de recherche : Technicolor et Inria Rennes
Thèse N° :

Rapporteurs avant soutenance :

Marc TOMMASI	Professeur - Université de Lille
Sébastien MONNET	Professeur - Université Savoie Mont-Blanc

Composition du jury :

Président :	Anne-Marie KERMARREC	Directrice de recherche Inria - Rennes
Examinateurs :	Marc TOMMASI	Professeur - Université de Lille
	Sébastien MONNET	Professeur - Université Savoie Mont-Blanc
	Gilles TREDAN	Chargé de recherche CNRS au LAAS - Toulouse
	Erwan LE MERRER	ARP INRIA - Rennes
Dir. de thèse :	Bruno SERICOLA	Directeur de recherche INRIA - Rennes

REMERCIEMENTS

Je tiens en premier lieu à remercier mes encadrants Bruno Sericola et Erwan Le Merrer pour m'avoir guidé au cours de ces trois années. Ils ont su être à mon écoute, m'apporter toute l'aide dont j'avais besoin tout en me laissant une grande liberté dans mon travail. Merci également aux membres du jury, Anne-Marie Kermarec, Marc Tommasi, Sébastien Monnet, et Gilles Tredan pour l'attention qu'ils ont porté à mes travaux et les remarques constructives qu'ils m'ont données aux cours de nos échanges.

Je remercie Technicolor et l'Inria de Rennes pour m'avoir fourni un environnement de travail agréable et dynamique. Un grand merci à Pascal Le Guyadec, François Schnitzler, Jean-Renan Vigouroux, Patrick Fontaine, Christoph Neumann, Thierry Filoche, Nicolas Le Scouarnec, Nelly Savatte, Michel Morvan et tous mes autres collègues de Technicolor avec qui j'ai pris grand plaisir à travailler. Merci également à Gerardo Rubino pour son aide et son expertise concernant les réseaux de neurones aléatoires. Merci à Yves Mocquard, Imad Alawe, Yassine Hadjadj-Aoul et Jorge Graneri pour les discussions enrichissantes que nous avons tenues sur les réseaux de neurones, que ce soit devant un tableau ou un café. Et merci à Fabienne Cuyolla et tous les membres de l'équipe Dionysos pour l'accueil que j'ai pu avoir au sein de l'Inria.

Enfin, je remercie ma famille et mes amis qui m'ont soutenu lors de ces années de travaux. Merci à Marine d'avoir été présente à mes cotés et de m'avoir toujours encouragé. Merci également à mes parents et Isabelle pour leurs relectures qui m'ont été d'une aide précieuse dans la rédaction de ce manuscript.

Introduction

L'arrivée du *Big data*

Depuis le début des années 2000, les capacités de stockage numérique ont grandement évoluées afin de faire face à une profusion de plus en plus importante de données. Celles-ci viennent de domaines qui peuvent être variés. Avec l'arrivée de l'Internet des objets (*Internet Of Things* en anglais) par exemple, le nombre de machines connectées les unes aux autres explose. Ces machines, de par leurs interactions avec l'extérieur et leurs différents capteurs, génèrent une quantité de données de plus en plus importante, et qui nécessite d'être analysée. Les réseaux sociaux sont également un autre domaine dans lequel les données se sont multipliées, en particulier les données multimédia ainsi que les méta-données contenant les informations sur les profils des différents utilisateurs. L'ensemble de ces données demandent de grosses capacités pour être stocké et de nombreuses ressources afin d'être analysé. En effet, si l'arrivée de cette masse de données a nécessité des progrès en terme de stockage, elle a aussi demandé et permis des avancées importantes en terme de traitement et d'analyse.

Ces données sont traditionnellement rassemblées dans des serveurs appelés *datacenter* (ou centre de données). Elles y sont analysées afin d'en extraire des connaissances et de proposer différents services. Ce processus d'analyse est généralement fait à l'intérieur même de l'environnement de stockage en utilisant des machines spécifiques adaptées à l'exécution de calculs intensifs. Aujourd'hui ces services sont possibles grâce notamment à des méthodes de statistiques et des méthodes d'apprentissage automatique appliquées sur ces bases de données.

Les avancées de l'apprentissage profond

Les outils d'apprentissage automatique sont généralement utilisés afin de proposer des services de plus en plus perfectionnés aux utilisateurs. Par exemple, ils sont utili-

sés afin de faire de la recommandation de films ou d'articles, mais également des services d'assistant personnel, de traduction, de diagnostic de panne, ou même d'analyse automatique d'images. Les modèles d'apprentissage automatique permettent d'extraire des connaissances par l'analyse de grandes quantités de données. Depuis quelques années, les algorithmes d'apprentissage profond ont révolutionné ce domaine, en parvenant à réaliser des tâches considérées jusqu'alors comme difficiles. Par exemple, dans le domaine de l'image, le challenge ILSVRC d'ImageNet consiste à classifier une grande variété d'images diverses, trouvables sur internet, avec plus de mille classes différentes. En 2012, les méthodes d'apprentissage profond ont atteint une erreur moyenne de 16.42% [63] sur les 5 premières prédictions (contrairement aux méthodes classiques qui ont obtenu 26.17% cette même année). Les cinq années suivantes, les méthodes d'apprentissage profond ont remporté à chaque fois les meilleurs scores jusqu'à atteindre une erreur moyenne de 0.023%[53] (toujours sur les 5 premières prédictions). Outre le domaine de l'image, les méthodes d'apprentissage profond ont également permis des avancées dans le traitement des langages naturels [24], la reconnaissance vocale [110], l'intelligence artificielle pour les jeux [99], et dans bien d'autres domaines [25].

L'apprentissage profond est basé sur les modèles de réseaux de neurones avec de nombreuses couches cachées, dits *réseaux de neurones profonds*. L'apprentissage de ces réseaux profonds est particulièrement difficile car cela nécessite l'ajustement d'un nombre de paramètres important. Des avancées récentes sur les réseaux de neurones ainsi que l'apparition de bases d'apprentissage de plus en plus grandes et des progrès en terme d'architecture matérielle (tel que l'utilisation de carte graphique ou de carte dédiée) ont permis de populariser l'apprentissage profond. De plus, de nombreuses recherches ont été faites afin d'accélérer l'apprentissage de ces modèles en utilisant les techniques de calcul parallèle.

Déplacer l'apprentissage sur les systèmes distribués

Comme nous l'avons vu, les réseaux de neurones profonds peuvent être utilisés pour proposer de nombreux services à différents utilisateurs. Dans l'idéal, l'apprentissage de ces modèles se fait à l'aide d'un ensemble de données provenant directement de ces mêmes utilisateurs. En pratique, cela demande de réunir toutes ces données sur une machine ou un serveur afin d'y entraîner un réseau de neurones profond.

Ce rassemblement de données et l'apprentissage exécuté à l'intérieur d'un *datacenter* posent cependant quelques problèmes.

— En terme de contrainte matérielle, le stockage d'une grande quantité de données ainsi que l'apprentissage d'un réseau de neurones requièrent de nombreuses ressources. Ces architectures sont donc relativement coûteuses pour l'opérateur qui souhaite proposer un nouveau service.

— Regrouper les données des utilisateurs sur un serveur pose également des contraintes en terme de respect de la vie privée. La réglementation concernant la vie privée est de plus en plus importante dans de nombreux pays [1] et les utilisateurs sont sensibilisés à l'utilisation de leurs données par les opérateurs.

Ce second point peut être résolu de différentes façons. Il est possible d'utiliser un sous-ensemble des données des utilisateurs (ceux qui ont autorisé l'accès à ces données par exemple) ou de les générer artificiellement, ou bien même expérimentalement. Une autre proposition consiste à détourner une base de données existante de son but initial afin d'entraîner le réseau de neurones profond à faire une autre tâche. Ces solutions ne sont pas idéales en terme d'apprentissage et peuvent rester relativement coûteuses pour l'opérateur.

L'objectif de cette thèse est de proposer des solutions afin d'effectuer l'apprentissage des réseaux de neurones profonds directement sur les machines des utilisateurs dans lesquelles les données sont stockées ou acquises. Cette solution a pour avantage de ne pas déplacer ces données des machines vers un serveur central. L'opérateur n'y a donc plus accès directement. De plus, déplacer l'apprentissage sur les machines des utilisateurs permet également de réduire les ressources nécessaires à l'opérateur afin d'apprendre le réseau de neurones profond.

Cette méthode requiert cependant un grand nombre de participants, autant pour la base de données d'apprentissage (distribuée chez les utilisateurs), que pour les ressources nécessaires en terme de calcul. Cela nous mène donc à des problématiques de calculs sur des systèmes distribués, appliquées à des tâches d'apprentissage de réseaux de neurones profonds. Dans cette thèse, nous présentons nos contributions pour effectuer l'apprentissage profond sur des systèmes distribués afin de permettre ce type d'apprentissage collaboratif. Des chercheurs se sont également intéressés à cette problématique, tels que les ingénieurs de Google, auteurs du *Federated Learning* [76,

1. Par exemple en UE, de nouvelles réglementations sont rentrées en réglementation en 2018 : https://www.cnil.fr/fr/reglement-europeen-protection-donnees

61]. Cette méthode a été proposée afin de distribuer l'apprentissage de réseaux de neurones sur le téléphone mobile des utilisateurs. Nous allons donc également présenter ces travaux au cours de ce document afin de nous y comparer.

Cette thèse est organisée de la manière suivante :

— Le chapitre 1 est consacré à la présentation de l'état de l'art sur les réseaux de neurones profonds et leurs différentes applications.

— Dans le chapitre 2, nous expliquons les contraintes d'un apprentissage sur un système distribué. Nous évoquerons les méthodes d'apprentissage parallélisé ou d'apprentissage distribué existantes, pouvant être intéressantes à notre problématique.

— Dans le chapitre 3, nous présentons une première contribution nommée *Ada-Comp*, qui permet l'apprentissage d'un réseau de neurones profond sur de nombreuses machines avec une communication réduite.

— Dans le chapitre 4, nous présentons une étude que nous avons faite sur l'utilisation des protocoles de rumeur (*Gossip*) pour entraîner un modèle particulier de réseaux de neurones appelés réseaux antagonistes génératifs (GAN pour *Generative Adversarial Network*).

— Enfin dans le chapitre 5, nous présenterons une nouvelle contribution pour distribuer l'apprentissage de réseaux génératifs antagonistes sur de nombreuses machines tout en réduisant la charge de travail sur celles-ci.

SOMMAIRE

1 État de l'art sur l'apprentissage profond **9**
1.1 L'apprentissage automatique 9
1.2 Les réseaux de neurones 13
1.2.1 Le modèle du perceptron 13
1.2.2 Le perceptron multi-couches 20
1.2.3 Apprendre avec la rétro-propagation 22
1.2.4 Convergence de l'apprentissage 27
1.2.5 Alternatives à la descente de gradient 29
1.3 Les réseaux de neurones profonds 32
1.3.1 L'intérêt des architectures profondes 32
1.3.2 Réseaux de neurones convolutifs 33
1.3.3 Réseaux de neurones récurrents 36
1.3.4 Techniques avancées pour améliorer l'apprentissage 38
1.3.5 Réseaux de neurones non supervisés 40
1.4 Conclusion 44

2 Vers un apprentissage profond sur des systèmes distribués **45**
2.1 Collaboration pour l'apprentissage d'un réseau de neurones profond . . 45
2.2 Systèmes distribués et contraintes 47
2.3 Apprentissage profond distribué 50
2.3.1 Différents types de parallélisme 50
2.3.2 Les architectures proposées 56
2.4 Conclusion et méthodes existantes adaptées aux systèmes distribués . 60

3 Descente de gradient asynchrone avec *AdaComp* **65**
3.1 Modèle du serveur de paramètres dans le cas des systèmes distribués 65
3.1.1 Gestion des gradients avec délai 67
3.1.2 Réduire le trafic entrant au niveau du serveur 68
3.2 La méthode d'*AdaComp* 70

3.3 Expériences 74
3.3.1 Plateforme expérimentale 74
3.3.2 Configuration expérimentale et compétiteurs 76
3.3.3 Précision du modèle 77
3.3.4 *AdaComp* face aux pannes 79
3.3.5 *AdaComp* dans un contexte d'agents hétérogènes 81
3.4 Discussion sur *AdaComp* et les travaux connexes 82
3.5 Conclusion 84

4 Protocole de rumeur pour l'apprentissage des réseaux antagonistes génératifs 87
4.1 Apprentissage collaboratif d'un GAN 88
4.1.1 Modèle du GAN 88
4.1.2 GAN et systèmes distribués 90
4.2 FL-GAN vs Gossip GAN : une comparaison expérimentale 93
4.2.1 Configuration des expériences 93
4.2.2 Passage à l'échelle et performances 96
4.3 Conclusion 98

5 MD-GAN : GAN avec de multiples discriminateurs pour des bases de données distribuées 101
5.1 Contexte distribué 102
5.2 La méthode du MD-GAN 104
5.2.1 Raisonnement 104
5.2.2 Procédure d'apprentissage du générateur 106
5.2.3 Procédure d'apprentissage des discriminateurs 108
5.2.4 Caractéristiques de MD-GAN 110
5.3 Évaluation expérimentale 113
5.3.1 Configuration expérimentale 113
5.3.2 Résultats des expériences 116
5.4 MD-GAN et les travaux connexes 122
5.5 Perspectives et conclusion sur MD-GAN 123

Conclusion 127

Chapitre 1

État de l'art sur l'apprentissage profond

L'apprentissage profond fait partie des méthodes d'apprentissage automatique. Dans ce chapitre nous allons expliquer ce qu'est l'apprentissage automatique et ce qu'il permet de faire. Nous présentons ensuite le modèle du réseau de neurones qui est à la base de l'apprentissage profond. Nous montrerons ensuite les différentes architectures profondes de réseaux de neurones et leur utilité. Le but de ce chapitre est d'avoir un aperçu de l'intérêt de l'apprentissage profond, mais également de ses contraintes.

1.1 L'apprentissage automatique

L'apprentissage automatique est motivé par la réalisation de tâches difficiles à définir de manière exhaustive ou par des règles simples dans des programmes classiques. Par exemple, développer une IA (Intelligence Artificielle) pour jouer au jeu de Go en respectant les règles du jeu est relativement simple à programmer car l'ensemble des règles peut facilement être défini (Chaque joueur joue un seul pion à son tour, il ne peut poser son pion que sur une case valide, etc). Cependant, faire jouer l'IA de manière optimale afin de remporter la victoire est impossible à définir simplement. Ceci est dû au fait que le jeu de Go n'a pas de stratégie optimale connue, ce qui ne permet pas de créer une succession de règles à suivre pour l'IA afin de gagner. De plus, le nombre d'états possibles du jeu ainsi que les possibilités à chaque état sont tellement importants qu'il est impossible de tout décrire dans un programme classique pour une IA. Prenons un autre exemple avec la vision assistée par ordinateur. La reconnaissance du visage d'un être humain sur une image peut nous paraître être une tâche simple car il nous est possible de le faire sans réflexion. Mais lorsque nous combinons la grande variété de visages possibles avec l'ensemble de toutes les dispositions de ces

visages dans une image, il est impossible de décrire simplement ceux-ci à partir de la valeur de chaque pixel de l'image. Dans ces deux situations, l'apprentissage automatique permet d'entrainer des modèles statistiques afin qu'ils trouvent par eux-même les connaissances nécessaires à l'accomplissement de ces tâches à l'aide d'exemples présents dans nos données.

Une définition formelle de l'apprentissage automatique a été proposée par T. Mitchell [78] : "Un programme informatique est dit capable d'apprentissage à partir d'une expérience E dans le respect d'une classe de tâche T avec la mesure de performance P s'il accomplit la tâche T, mesurée par P, et améliorée par l'expérience E". Nous appellerons modèle (modèle statistique ou modèle d'apprentissage) ce programme informatique capable d'apprentissage. Voyons tout d'abord ce modèle comme une boîte noire, capable de prendre en entrée des données de l'extérieur (par exemple, des images d'une caméra, le trafic réseau d'un routeur, etc) et renvoyer une sortie (par exemple, prise de décision de l'IA, description d'une image, etc). Ce modèle possède des paramètres θ qui permettent d'influencer sa sortie en fonction de l'entrée (voir Figure 1.1).

Comme expliqué dans la définition de Michell, l'apprentissage nécessite une expérience E. Celle-ci consiste généralement en une base de données d'apprentissage B_{train} que le modèle analyse lors du processus d'apprentissage. Cet apprentissage d'une tâche T se fait à l'aide d'une fonction de coût J. Cette fonction de coût est calculée sur une base de données de test B_{test}, distincte de B_{train}, afin de mesurer les performances du modèle appris ; c'est la mesure de performance P. Lors de l'apprentissage, le modèle doit donc être capable de modifier ses paramètres θ à l'aide de la base de données d'apprentissage B_{train} pour améliorer ses performances mesurées par P. Le fait que la performance soit mesurée sur une base de données B_{test} distincte de B_{train} implique une capacité de généralisation du modèle, c'est-à-dire, une capacité à répondre à des cas qu'il n'a pas vu lors de son expérience E. Le but du modèle est de faire tendre ses paramètres θ vers un optimal θ^* qui minimise J sur B_{test} (la mesure P).

L'apprentissage automatique se divise en deux catégories en fonction de la nature de l'expérience E : l'apprentissage supervisé et l'apprentissage non supervisé. Dans le cas de l'apprentissage supervisé, pour chaque exemple $\boldsymbol{x} \in B_{train}$, la sortie attendue du modèle $\boldsymbol{y}$ lui est donnée. Nous présentons ici deux cas classiques de tâches d'apprentissage supervisé :

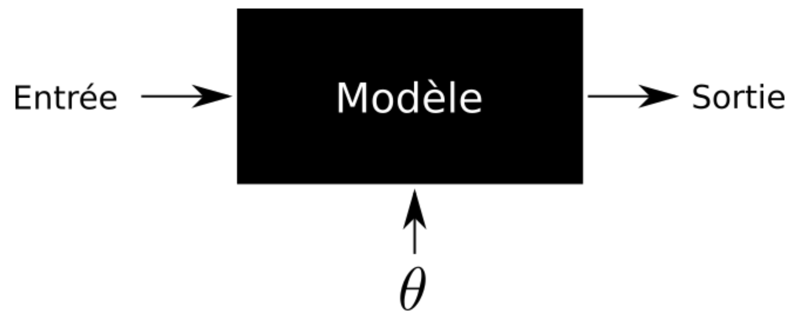


FIGURE 1.1 – Modèle d'apprentissage vu comme une boîte noire.

— La classification : Chaque donnée en entrée $\boldsymbol{x}$ est associée à un label $\boldsymbol{y} \in \{0,1\}^m$. Une valeur de 1 dans la i-ème entrée de $\boldsymbol{y}$ signifie que $\boldsymbol{x}$ appartient à la i-ème classe parmis les m possibles. La classification est dite multi-classes lorsque $m > 2$ et multi-labels lorsqu'il est possible que plusieurs entrées de $\boldsymbol{y}$ soient égales à 1. Un exemple de classification binaire (c'est-à-dire, à deux classes mutuellement exclusives) est présenté dans la Figure 1.2.
— La régression : Consiste à associer l'entrée $\boldsymbol{x}$ à un $\boldsymbol{y} \in R^m$. La principale différence avec la classification est l'espace de sortie Y qui est continu.

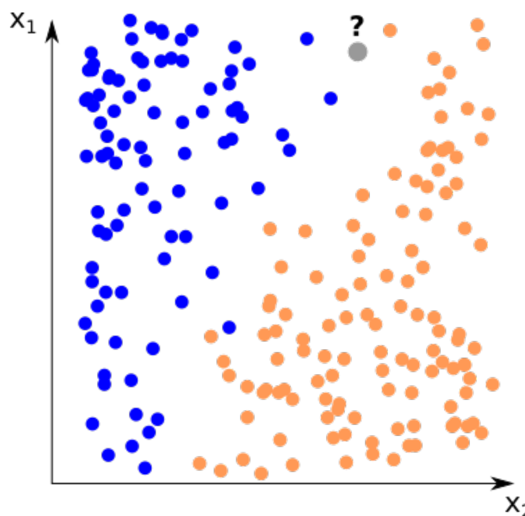


FIGURE 1.2 – Exemple d'une classification avec des données d'entrées $\boldsymbol{x} \in \mathbb{R}^2$ représentées sur le graphique. Les points bleus et les points oranges représentent des données appartenant à deux classes différentes. Le but d'un classifieur est d'apprendre la frontière entre ces deux classes. Il peut ainsi classifier des nouvelles données (en gris sur la figure).

Dans le cas de l'apprentissage non supervisé, les exemples de la base de don-

nées d'apprentissage B_{train} ne possèdent pas de label attendu en sortie. Le but de l'apprentissage est de trouver une structure cohérente aux données en entrée. Les 3 principales tâches qui peuvent être réalisées à partir d'un apprentissage non supervisé sont les suivantes :

— *Clustering* : Cette tâche consiste à séparer les données d'entrée en différents groupes en fonction de leur structure ou de leurs similarités. Contrairement aux tâches de classification, les groupes ne sont pas connus par avance. Dans cette catégorie, il est possible de citer les algorithmes des K-moyennes ou de clustering hiérarchique.
— Distribution de densité : Cette tâche consiste à trouver ou capturer la distribution (inconnue) des données en entrée. L'un des exemples le plus récent est le modèle des réseaux antagonistes génératifs [39] que nous verrons en détails au cours de ce document.
— Réduction de la dimensionnalité : Cette tâche consiste à compresser les données d'entrée (par exemple, représentées dans $\mathbb{R}^n$) sur un espace de représentation plus petit (par exemple, R^m avec $n >> m$). Outre le simple gain en terme de place, cette réduction est également intéressante pour représenter les données. Réduire la dimensionnalité permet d'éviter les problématiques liées à la "malédiction de la dimension". Cette malédiction fait référence aux problématiques que l'on rencontre sur des données de grandes dimensions. Par exemple, la recherche de plus proches voisins obtient de très mauvais résultats sur des données avec beaucoup de dimensions [11]. Un exemple intéressant de réduction de dimension est le modèle du Word2Vec [77], capable de représenter des mots, par des vecteurs denses de quelques centaines de dimensions. Cette représentation a tendance à rapprocher les mots avec des sens proches les uns des autres.

Cette liste des tâches possibles en apprentissage n'est pas exhaustive. Nous n'abordons pas le cas, un peu particulier, de l'apprentissage par renforcement [57]. De même, il est possible de trouver des cas d'apprentissage dits semi-supervisés, dans lesquels les informations des données ne sont pas complètes [59, 19]. Dans ce chapitre nous allons voir le modèle d'apprentissage des réseaux de neurones profonds. Celui-ci est utilisé pour de nombreuses tâches différentes de nos jours, qu'elles soient supervisées ou non.

1.2 Les réseaux de neurones

Les réseaux de neurones sont des modèles d'apprentissage automatique capables de représenter une relation entre des données d'un espace X et un espace de sortie Y. Ils sont utilisés dans de nombreux domaines, comme la vision assistée par ordinateur [85, 45, 102], le traitement du langage naturel [24], l'analyse audio [110, 7], mais également pour développer des IA capables de jouer à des jeux [99] ou utilisées comme assistant personnel (tel que Alexa d'Amazon, Siri d'Apple, Cortana de Microsoft ou l'Assistant de Google).

L'unité de calcul de base est le neurone. Celui-ci prend en entrée plusieurs signaux et les interprète pour envoyer un nouveau signal vers d'autres neurones ou vers la sortie du réseau de neurones, c'est-à-dire la sortie du modèle. Il existe de nombreuses architectures pour construire des réseaux de neurones (voir la Section 1.3).

Pour introduire les réseaux de neurones, nous allons commencer par présenter un modèle composé d'un seul neurone, appelé modèle du perceptron. Celui-ci va nous permettre de mettre en évidence les mécanismes de base de tout réseau de neurones.

1.2.1 Le modèle du perceptron

Dans sa version la plus simple, le perceptron est un réseau de neurones composé de seulement un neurone, qui prend en entrée n données binaires. Chacune de ses entrées i est pondérée par un poids noté w_i. Le neurone peut prendre les états $"1"$ ou $"0"$ (respectivement actif ou non-actif) en fonction de ses entrées pondérées et d'un biais noté $\beta \in \mathbb{R}$. Cet état représente la sortie du modèle. Il est donc possible de représenter le perceptron comme une fonction paramétrique $f_\theta : \{0,1\}^n \to \{0,1\}$ avec θ l'ensemble de ses paramètres, c'est-à-dire le biais β et les poids $\mathbf{w} = (w_1, \ldots, w_n)$. La sortie d'un perceptron pour un vecteur $\boldsymbol{x} \in \{0,1\}^n$ en entrée est calculée tel que :

$$f(\boldsymbol{x}, \mathbf{w}) = H(z(\boldsymbol{x}, \mathbf{w}) + \beta),$$

avec $H(t)$ la fonction de Heaviside définie pour tout $t \in \mathbb{R}$ comme $H(t) = \mathbb{1}_{\{t>0\}}$ et $z(\boldsymbol{x}, \mathbf{w})$ la somme pondérée des entrées :

$$z(\boldsymbol{x}, \mathbf{w}) = \mathbf{w}^\top \boldsymbol{x} = \sum_{j=1}^{n} w_j x_j.$$

Intuitivement, les poids $w_1, \dots, w_n$ représentent l'importance accordée à chaque entrée pour l'activation du perceptron. Pour rappel, $\mathbb{1}_A$ est la fonction indicatrice qui est égale à 1 si la condition A est vérifiée et 0 sinon. Le biais β peut être vu comme l'ajout d'un seuil à la difficulté d'activation du perceptron. En effet, si la somme pondérée $z(\boldsymbol{x}, \mathbf{w})$ dépasse $-\beta$ (l'opposé du biais), le perceptron s'active, sinon il reste inactif.

Pour simplifier les notations, nous allons inclure le biais β dans le vecteur de poids en ajoutant une constante en entrée $x_0 = 1$ (le biais devient donc w_0). Nous obtenons la fonction paramétrique $f_\theta : \{0,1\}^{n+1} \times \mathbb{R}^{n+1} \to \{0,1\}$ telle que :

$$f(\boldsymbol{x}, \mathbf{w}) = H(z(\boldsymbol{x}, \mathbf{w})) = H(\sum_{j=0}^{n} w_j x_j). \tag{1.1}$$

Exemple de classification d'e-mail Soit un perceptron modélisé par la fonction f qui a pour but de classifier les e-mails reçus avec des labels "*Important*" ou "*Non important*". Les entrées x_j sont les caractéristiques des e-mails reçus, telles que "Envoyé par un contact", "Contient une pièce jointe", "Est une réponse automatique". Le courrier électronique est étiqueté comme "*Important*" lorsque le neurone est actif et "*Non important*" lorsque le neurone reste inactif.

Fixons les poids associés à $w_1 = 2$, $w_2 = 1$, $w_3 = -1$ et le biais à $\beta = -0,5$ (ou $w_0 = -0.5$), comme décrit dans la Figure 1.3. Dans cet exemple, nous pouvons voir qu'un mail est étiqueté comme important s'il est envoyé par un contact ou s'il contient une pièce jointe, mais qu'il ne s'agit pas d'une réponse automatique.

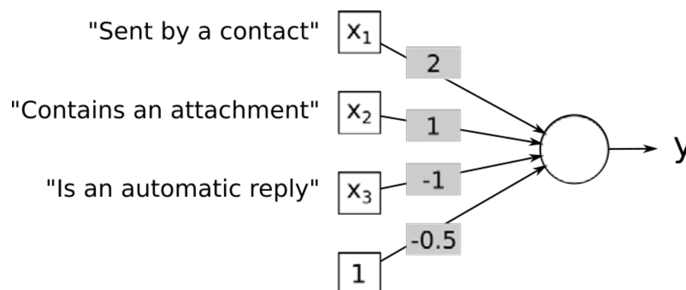


FIGURE 1.3 – Représentation graphique d'un perceptron décrit dans l'exemple de classification de e-mails. Les carrés représentent l'entrée du perceptron et le cercle représente le neurone. Les poids sont écrits sur les connexions entre les entrées et le neurone.

Supposons maintenant que nous n'ayons aucune connaissance de ce qui fait un courrier important ou non. Plus précisément, nous ne connaissons pas les poids w_j

associés à chaque entrée x_j. Cependant, nous avons accès à une base de données d'e-mails, labellisés à la main par des utilisateurs avec les labels "Important" et "Non important" ainsi que les caractéristiques associées à chacun (par exemple, "envoyé par un contact", "contient une pièce jointe",...). Il est alors possible d'extraire les connaissances concernant la description d'un mail important à partir de ces exemples en cherchant les poids associés à chaque connexion qui correspondent le mieux à cette base. Cette phase de recherche va correspondre à l'apprentissage du perceptron. Une fois ce processus terminé sur cette base de données, le perceptron devrait être capable de généraliser cette classification sur des nouveaux mails entrants.

L'apprentissage requiert une fonction dérivable en tout point pour calculer la sortie du perceptron. Au lieu d'utiliser la fonction de Heaviside comme fonction d'activation, nous allons introduire la fonction sigmoïde :

$$s(t) = \frac{1}{1 + e^{-t}}.$$

La Figure 1.4 montre la courbe représentant fonction sigmoïde. Celle-ci possède une forme en "S" proche de la fonction de Heaviside. L'avantage de la fonction sigmoïde cependant est d'être dérivable en tout point (contrairement à la fonction d'Heaviside). La fonction définie par le perceptron avec la fonction sigmoïde est donc la suivante :

$$f(\boldsymbol{x}, \mathbf{w}) = \frac{1}{1 + e^{-z(\boldsymbol{x}, \mathbf{w})}}. \tag{1.2}$$

La sortie du perceptron est maintenant définie sur $]0, 1[$. Elle peut être interprétée comme la probabilité que le neurone s'active en fonction de l'entrée.

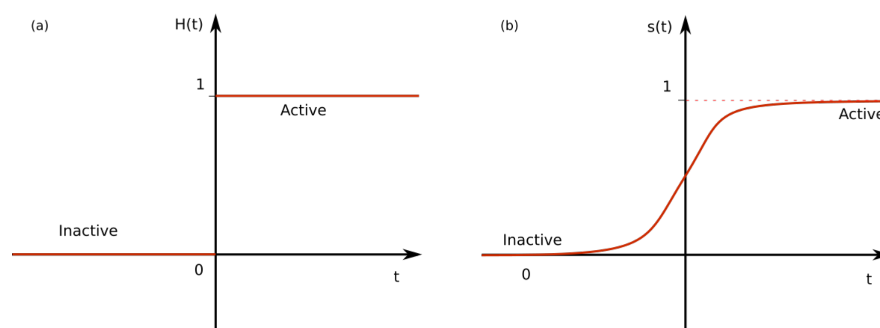


FIGURE 1.4 – Deux fonctions d'activation différentes avec une forme de "S" : (a) représente la fonction de Heaviside (avec une dérivée nulle sur $\mathbb{R} - \{0\}$ et non définie sur $\{0\}$) ; (b) représente la fonction sigmoïde.

L'apprentissage des paramètres **w** se fait par la minimisation d'une fonction de coût J_{train} sur la base de données d'apprentissage B_{train}. Cette fonction représente les erreurs faites par le modèle (avec ses paramètres actuels) sur la classification des e-mails de notre base de données d'apprentissage. Dans notre exemple, nous allons définir la fonction de coût comme :

$$J_{train}(\mathbf{w}) = \frac{1}{K} \sum_{k=1}^{K} L(\boldsymbol{x}^{(k)}, y^{(k)}, \mathbf{w}),$$

avec $\boldsymbol{x}^{(k)}$ et $y^{(k)}$ qui sont respectivement les caractéristiques et le label du k-ième exemple de la base de données d'apprentissage. L correspond à la fonction utilisée pour calculer l'erreur sur un exemple de la base. Dans notre cas nous choisissons comme fonction :

$$L(\boldsymbol{x}^{(k)}, y^{(k)}, \mathbf{w}) = \frac{1}{2} \left(f(\boldsymbol{x}^{(k)}, \mathbf{w}) - y^{(k)} \right)^2.$$

La fonction de coût J_{train} représente donc l'erreur quadratique moyenne entre la sortie donnée par le perceptron et la valeur attendue sur l'ensemble de la base de données d'apprentissage. Le but de l'apprentissage est de trouver le vecteur de paramètres $\mathbf{w}^*$ qui minimise J_{train}. Nous commençons par initialiser la valeur des paramètres **w** par le vecteur nul :

$$\mathbf{w}_0 = (0, \ldots, 0)^\top$$

avec $\mathbf{w}_0$ qui dénote l'état de **w** au temps 0, c'est-à-dire au début de l'apprentissage. Le vecteur des paramètres est modifié de manière itérative en utilisant la règle suivante :

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla J_{train}(\mathbf{w}_t).$$

$\mathbf{w}_t$ représente l'état de **w** à l'itération t, $\alpha \in (0, 1]$ est un coefficient appelé taux d'apprentissage, et $\nabla J_{train}(\mathbf{w}_t)$ est le gradient de la fonction de coût pour l'état $\mathbf{w}_t$, défini par :

$$\nabla J_{train}(\mathbf{w}_t) = \left(\frac{\partial J_{train}(\mathbf{w}_t)}{\partial w_1}, \ldots, \frac{\partial J_{train}(\mathbf{w}_t)}{\partial w_n} \right)^\top.$$

Cette méthode est un algorithme de minimisation de premier ordre appelé descente de gradient. Le vecteur de paramètres **w** se déplace à l'opposé du gradient de la

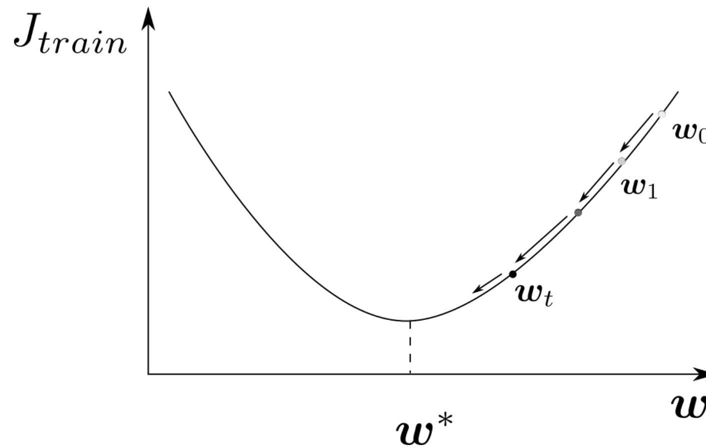


FIGURE 1.5 – Exemple de descente de gradient sur une dimension

fonction de perte afin de trouver un minimum local (voir la Figure 1.5). L'hyperparamètre α est utilisé pour moduler le pas de déplacement dans l'espace des paramètres. Plus α est grand et plus les modifications apportées au vecteur **w** lors d'une itération sont importantes. Nous verrons dans la Section 1.2.5 que ce paramètre est important pour garantir la bonne convergence de l'apprentissage sur un minimum local intéressant. Le calcul du gradient $\nabla J_{train}(\mathbf{w}_t)$ demande de calculer pour chaque poids w_j sa dérivée partielle $\partial J_{train}/\partial w_j$:

$$\begin{aligned}\frac{\partial J_{train}}{\partial w_j} &= \frac{\partial}{\partial w_j}\left(\frac{1}{K}\sum_{k=1}^{K} L(\boldsymbol{x}^{(k)}, y^{(k)}, \mathbf{w})\right) \\ &= \frac{1}{K}\sum_{k=1}^{K} \frac{\partial L(\boldsymbol{x}^{(k)}, y^{(k)}, \mathbf{w})}{\partial w_j}.\end{aligned}$$

Cette dérivée partielle est la moyenne des dérivées partielles de la fonction d'erreur L sur chaque exemple de la base de données d'apprentissage. En utilisant deux fois le théorème de dérivation de fonction composée sur la fonction d'erreur L, nous pouvons la développer en ces trois termes :

$$\frac{\partial L(\boldsymbol{x}^{(k)}, y^{(k)}, \mathbf{w})}{\partial w_j} = \frac{\partial L}{\partial f}\frac{\partial f}{\partial z}\frac{\partial z}{\partial w_j} \tag{1.3}$$

avec :

$$L \equiv \frac{1}{2}(f-y)^2,$$

$$f \equiv \frac{1}{1+e^{-z}},$$
$$z = \sum_{i=0}^{n} w_i x_i^{(k)}.$$

Le premier terme représente à quel point l'erreur faite pour l'exemple k dépend de la sortie du perceptron. Cette dérivée peut être calculée comme étant :

$$\begin{aligned} \frac{\partial L}{\partial f} &= \frac{\partial}{\partial f}\left(\frac{1}{2}\left(f - y^{(k)}\right)^2\right) \\ &= f - y^{(k)} \end{aligned} \tag{1.4}$$

Le second terme est la dérivée de la fonction d'activation, c'est-à-dire, la fonction sigmoïde :

$$\begin{aligned} \frac{\partial f}{\partial z} &= \frac{\partial}{\partial z}\left(\frac{1}{1+e^{-z}}\right) \\ &= \frac{e^{-z}}{(1+e^{-z})^2}. \end{aligned} \tag{1.5}$$

Le dernier terme représente la dérivée partielle de la somme des entrées pondérées en fonction des poids w_j :

$$\frac{\partial z}{\partial w_j} = \frac{\partial}{\partial w_j}\left(\sum_{i=0}^{n} w_i x_i^{(k)}\right).$$

L'entrée pondérée $w_j x_j$ est le seul terme non-constant de cette somme en fonction de w_j. Nous obtenons donc :

$$\frac{\partial z}{\partial w_j} = x_j^{(k)}. \tag{1.6}$$

À partir des relations (1.4),(1.5) et (1.6), nous obtenons :

$$\frac{\partial L}{\partial w_j} = (f - y^{(k)})\frac{e^{-z}}{(1+e^{-z})^2} x_j^{(k)} \tag{1.7}$$

La dérivée partielle $\frac{\partial L}{\partial w_j}$ peut se réécrire comme :

$$\frac{\partial L}{\partial w_j} = \delta^{(k)} x_j^{(k)} \tag{1.8}$$

avec le terme $\delta^{(k)} = (f - y^{(k)})e^{-z}/(1 + e^{-z})^2$ qui ne dépend pas de j. En terme de calcul, il est possible de calculer $\delta^{(k)}$ une seule fois pour toutes les entrées j à chaque exemple k. La dérivée partielle finale est

$$\frac{\partial J}{\partial w_j} = \frac{1}{K} \sum_{k=1}^{K} \delta^{(k)} x_j^{(k)}.$$

L'exécution d'une itération de la descente de gradient demande de calculer l'erreur faite par le perceptron sur chaque exemple de la base de données d'apprentissage. Cette erreur permet ensuite de calculer la dérivée partielle pour chaque entrée et donc d'avoir le gradient du vecteur de paramètres. L'algorithme de descente de gradient peut donc mettre à jour les paramètres et démarrer une nouvelle itération. La descente de gradient continue à effectuer des itérations jusqu'à l'obtention d'un vecteur de paramètres suffisamment intéressant décrit par un critère d'arrêt. Nous verrons dans la section 1.2.4 comment peut être défini ce critère d'arrêt.

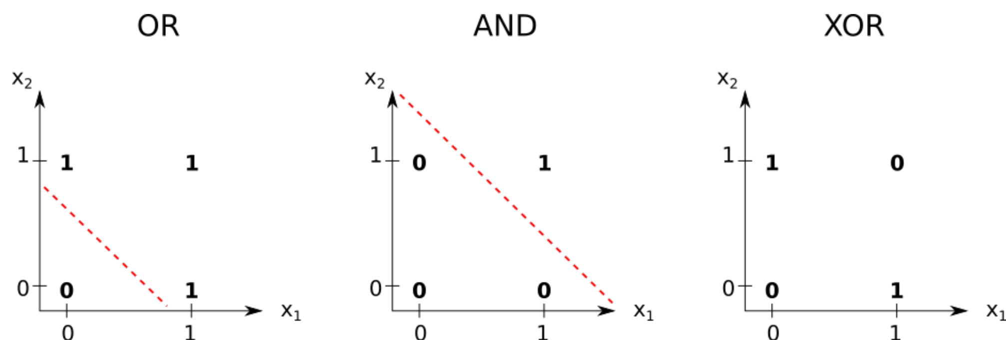


FIGURE 1.6 – Exemple de fonctions logiques : la sortie de la fonction OU et de la fonction ET sont linéairement séparables tandis que les sorties de la fonction OU EXCLUSIF ne peuvent pas être séparées linéairement. Le perceptron ne peut donc pas représenter cette fonction. Quand $x_1 = 0$, la sortie du perceptron doit augmenter si x_2 augmente et si $x_1 = 1$ la sortie du perceptron doit décroître si x_2 augmente. Le premier requiert que $w_2 > 0$ alors que le second requiert $w_2 < 0$.

Limites du perceptron Le perceptron est un classifieur dit linéaire, c'est-à-dire qu'il classifie des données à partir d'une combinaison linéaire de ses entrées. Il est donc

incapable de classifier des données dans des classes non linéairement séparables (c'est-à-dire non séparables avec un hyperplan dans l'espace des données). Par exemple, il est impossible de représenter la fonction OU EXCLUSIF avec un perceptron (voir Figure 1.6)

1.2.2 Le perceptron multi-couches

Les perceptrons multi-couches, appelé aussi MLP (pour *Multi-layer Perceptron*), sont des réseaux de neurones plus généraux que le perceptron. Ils sont composés d'une multitude de neurones interconnectés et organisés en couches successives. Un MLP peut être représenté par un graphe acyclique dans lequel chaque noeud représente un neurone. Les arcs orientés représentent les relations entre chaque neurone : un arc du noeud i aux noeud j signifie que le neurone j prend la valeur d'activation du neurone i en entrée. La Figure 1.7 montre une représentation graphique d'un MLP avec 3 couches ayant respectivement $5, 4, 3$ neurones.

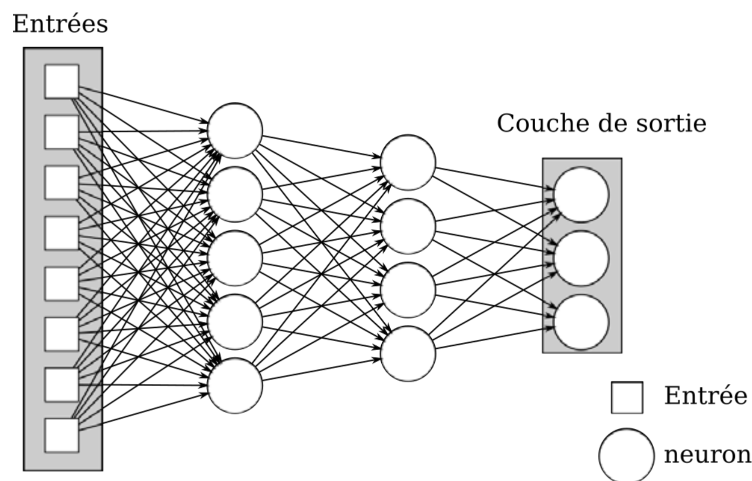


FIGURE 1.7 – Exemple d'une représentation d'un MLP.

Comme montré sur la Figure 1.7, chaque neurone de la première couche prend en entrée, l'entrée du MLP. Chaque couche suivante reçoit en entrée les valeurs d'activation de la couche précédente (c'est-à-dire, le vecteur contenant les valeurs de chaque neurone de la couche précédente). La sortie du MLP est composée de la valeur d'activation de chaque neurone de la dernière couche, appelée *couche de sortie*. Le vecteur d'activation de la couche l, composé de s neurones peut être calculé suivant le vecteur d'entrées $\mathbf{e} \in \mathbb{R}^p$ (Les entrées du MLP ou le vecteur d'activation de la couche

précédente), de la manière suivante :

$$\mathbf{a}^{(l)} = f^{(l)}(\mathbf{e}, W^{(l)}, \boldsymbol{\beta}^{(l)}) = \phi^{(l)}(W^{(l)}\mathbf{e} + \boldsymbol{\beta}^{(l)})$$

avec $W = (\mathbf{w}_{1.}, \ldots, \mathbf{w}_{s.})^\top \in \mathbb{R}^{s \times p}$ et $\boldsymbol{\beta}^{(l)} = (\beta_1, \ldots, \beta_s)^\top \in \mathbb{R}^s$ avec respectivement $\mathbf{w}_{i.}$ le vecteur des poids du neurone i et β_i son biais. La fonction $\phi^{(l)}$ est une fonction d'activation appliquée individuellement à chaque neurone de la couche l. Dans la section précédente, nous avons vu la fonction de Heaviside et la fonction sigmoïde. Dans un MLP, tous les neurones d'une même couche ont la même fonction d'activation $\phi^{(l)}$.

Le MLP est représenté par une fonction f qui prend en entrée des données $\boldsymbol{x} \in \mathbb{R}^n$ et un ensemble de paramètres $\theta = \{W^{(l)}, \boldsymbol{\beta}^{(l)} | l \in \{1, \ldots, L\}\}$, correspondant à l'ensemble des matrices $W^{(l)}$ et des vecteurs $\boldsymbol{\beta}^{(l)}$ pour toutes les couches $l = 1, \ldots, L$, et donne en sortie $\hat{y} \in \mathbb{R}^m$. Comme décrit dans l'équation d'un perceptron (voir équation 1.1), il est possible d'inclure le vecteur $\boldsymbol{\beta}^{(l)}$ dans la matrice $W^{(l)}$ en ajoutant une entrée constante pour chaque couche (ce qui rajoute une colonne à chaque matrice $W^{(l)}$). La fonction f est une composition de fonctions $f^{(l)}$ associées à chaque couche l du réseau. Par exemple, avec un MLP à trois couches, nous avons :

$$\hat{y} = f(\boldsymbol{x}, \theta) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x}, W^{(1)}), W^{(2)}), W^{(3)})$$

Un MLP avec un nombre de couches plus grand ou égal à 2 est un approximateur universel de fonctions, c'est-à-dire, qu'il est capable de représenter toutes sortes de fonctions si ses paramètres sont correctement ajustés (sous certaines conditions sur la fonction d'activation des couches cachées [51]). Pour illustrer ceci, nous prenons l'exemple de la fonction OU EXCLUSIF avec deux entrées. Soit un MLP avec deux entrées et deux couches, composées de 2 et 1 neurones. Les paramètres du MLP sont :

$$W^{(1)} = \begin{bmatrix} 0 & 1 & 1 \\ -1 & 1 & 1 \end{bmatrix}$$

$$W^{(2)} = \begin{bmatrix} 0 & 1 & -2 \end{bmatrix}$$

Nous utilisons des unités linéaires rectifiées (appelé ReLU) utilisées régulièrement dans les réseaux de neurones modernes tels que dans [63, 102]. Ce type de neurone utilise la fonction d'activation $\phi(z) = \max\{0, z\}$ (voir Figure 1.8). La totalité du MLP est

représentée graphiquement sur la Figure 1.9.

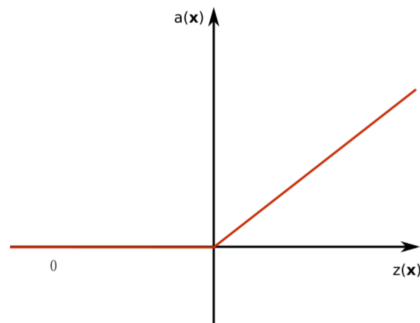


FIGURE 1.8 – La fonction d'activation des unités linéaires rectifiées.

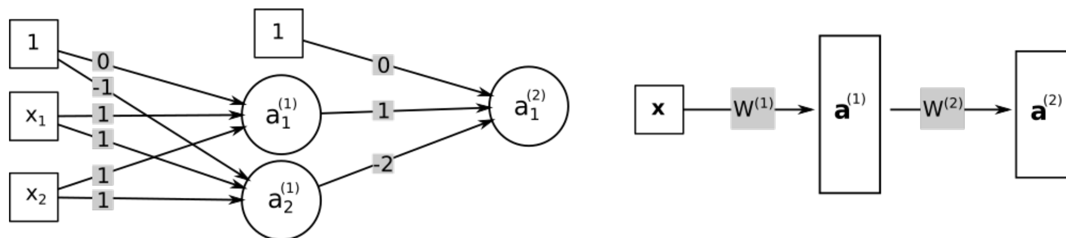


FIGURE 1.9 – Réseau de neurones représentant la fonction OU EXCLUSIF avec deux représentations graphiques. À gauche, chaque neurone est représenté par un cercle. Les poids sont représentés sur les connexions entre les neurones. De même, les biais représentés par la connexion entre une constante (carrés) et chaque neurone. À droite, dans ce style graphique, chaque couche est représentée par un rectangle. Les matrices de paramètres peuvent être indiquées sur les connexions entre les couches. L'avantage de cette seconde représentation est d'être plus compacte que la première.

Les couches intermédiaires de ce MLP transforment l'espace de représentation des données d'entrée comme montré dans la Figure 1.10. Ce nouvel espace de représentation permet de séparer linéairement les sorties de la fonction visée (la fonction OU EXCLUSIF dans cet exemple). Les couches intermédiaires peuvent être vues comme des représentations des entrées à plus haut niveau.

1.2.3 Apprendre avec la rétro-propagation

Maintenant que nous avons défini le modèle du MLP, nous allons voir que ce dernier permet d'approximer une fonction $g : \mathbb{R}^n \rightarrow \mathbb{R}^m$ avec un ensemble de paramètres θ. Un réglage à la main des paramètres θ afin de trouver θ^*, tel que $f(\boldsymbol{x}, \theta^*) \approx g(\boldsymbol{x})$

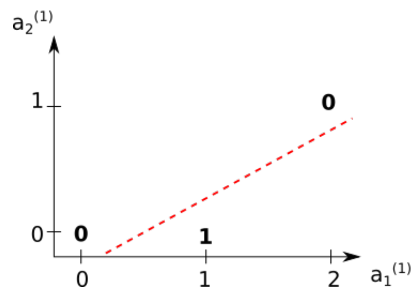


FIGURE 1.10 – Représentation intermédiaire de l'entrée dans la couche 1. Dans cette représentation, les classes sont linéairement séparables contrairement à la représentation originale des données représentées sur la Figure 1.6. C'est pourquoi le MLP à 2 couches est capable de représenter la fonction OU EXCLUSIF (contrairement au perceptron).

pour tout $\boldsymbol{x}$ des connaissances d'expert de la fonction g. De plus, la taille de θ est de l'ordre de $O(n^2)$, avec n le nombre de neurones du MLP. Cela représente jusqu'à quelques millions de valeurs dans les MLP modernes. Il est donc nécessaire d'utiliser une méthode d'optimisation automatique pour se rapprocher de θ^*.

Descente de gradient

Le MLP est un modèle adapté à l'apprentissage supervisé. Nous verrons dans la Section 1.3.5 comment les réseaux de neurones peuvent effectuer des tâches non supervisées. Dans cette section, nous allons montrer comment apprendre à un MLP une tâche de régression, en utilisant la méthode de descente de gradient (GD) et de rétro-propagation. Soit un MLP qui doit approximer une fonction inconnue $g : \mathbb{R}^n \to \mathbb{R}^m$ et f la fonction définie par ce MLP . Pour évaluer la capacité de $f(.,\theta)$ à approximer la fonction cible g, nous introduisons la fonction de coût suivante :

$$J^*(\theta) = E\left[L(\boldsymbol{x}, \boldsymbol{y}, \theta)\right] = \int L(\boldsymbol{x}, \boldsymbol{y}, \theta) p(\boldsymbol{x}, \boldsymbol{y}) \mathrm{d}\boldsymbol{x} \mathrm{d}\boldsymbol{y}$$

avec L la fonction d'erreur de l'exemple $(\boldsymbol{x}, \boldsymbol{y})$. Une fonction d'erreur couramment utilisée, que nous prendrons dans cet exemple est la fonction du moindre carré, définie comme suit :

$$L(\boldsymbol{x}, \boldsymbol{y}, \theta) = \frac{1}{2} \sum_{i=1}^{m} (f(x_i, \theta) - y_i)^2$$

L'apprentissage consiste à minimiser la fonction de coût J^*, c'est-à-dire, minimiser

l'espérance d'erreur de f étant donné θ. Pour ce faire, nous utilisons la fonction de coût empirique sur un ensemble d'apprentissage B_{train} composée du tuplet $(\boldsymbol{x}^{(k)}, \boldsymbol{y}^{(k)})$, tel que $g(\boldsymbol{x}^{(k)}) = \boldsymbol{y}^{(k)}$. La fonction de coût empirique est définie comme l'erreur moyenne du MLP pour les exemples de la base de données d'apprentissage :

$$J(\theta) = \frac{1}{|B_{train}|} \sum_{(\boldsymbol{x}^{(k)}, \boldsymbol{y}^{(k)}) \in B_{train}} L(\boldsymbol{x}^{(k)}, \boldsymbol{y}^{(k)}, \theta)$$

Le but est de trouver θ^* qui minimise la fonction de coût empirique. Cette minimisation est effectuée par la méthode appelée descente de gradient, que nous avons vue dans l'algorithme d'apprentissage du perceptron (voir Section 1.2.1).

La règle de mise à jour suivante est appliquée à chaque itération pour chacune des matrices $W^{(l)}$:

$$W_{t+1}^{(l)} = W_t^{(l)} + \alpha \Delta W_t^{(l)} \tag{1.9}$$

avec :

$$\Delta W_t^{(l)} = \begin{bmatrix} \frac{\partial L(\theta)}{\partial w_{11}^{(l)}} & \cdots & \frac{\partial L(\theta)}{\partial w_{1p}^{(l)}} \\ \vdots & \vdots & \vdots \\ \frac{\partial L(\theta)}{\partial w_{s1}^{(l)}} & \cdots & \frac{\partial L(\theta)}{\partial w_{sp}^{(l)}} \end{bmatrix}$$

Méthode de la rétro-propagation Comme pour le perceptron, la dérivée partielle de la fonction de coût peut être calculée comme la moyenne de la dérivée partielle de la fonction d'erreur sur chaque exemple de B_{train}.

$$\begin{aligned} \frac{\partial J}{\partial w_{ij}^{(l)}} &= \frac{\partial}{\partial w_{ij}^{(l)}} \left(\frac{1}{|B_{train}|} \sum_{(\boldsymbol{x}^{(k)}, \boldsymbol{y}^{(k)}) \in B_{train}} L(\boldsymbol{x}^{(k)}, \boldsymbol{y}^{(k)}, \theta) \right) \\ &= \frac{1}{K} \sum_{(\boldsymbol{x}^{(k)}, \boldsymbol{y}^{(k)}) \in B_{train}} \frac{\partial L(\boldsymbol{x}^{(k)}, \boldsymbol{y}^{(k)}, \theta)}{\partial w_{ij}^{(l)}}, \end{aligned} \tag{1.10}$$

Le processus de rétro-propagation est utilisé pour calculer la dérivée partielle $\partial L^{(k)} / \partial w_{ij}^{(l)}$ pour tout $w_{ij}^{(l)}$ étant donné une entrée $\boldsymbol{x}^{(k)}$ et le label associé $\boldsymbol{y}^{(k)}$. La rétro-propagation est calculée en 2 étapes :

— *La propagation avant* : la valeur d'activation $\mathbf{a}^{(l)}$ est calculée pour chaque couche l, de la première couche cachée jusqu'à la couche de sortie, en fonction de l'en-

trée $\boldsymbol{x}^{(k)}$ du MLP .

— La *rétro-propagation* : le terme d'erreur de chaque neurone est calculé à partir de la couche de sortie jusqu'à la première couche, en comparant la sortie $\hat{\boldsymbol{y}}^{(k)}$ du MLP (c'est-à-dire, l'activation $\mathbf{a}^{(d)}$ de la couche de sortie pour l'entrée $\boldsymbol{x}^{(k)}$) avec la sortie attendue $\boldsymbol{y}^{(k)}$.

Le terme d'erreur est calculé comme suit :

$$\delta_j^{(l)} = \frac{\partial L}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = \begin{cases} (a_j^{(l)} - y_j)\phi_j^{(l)\prime}(z_j^{(l)}) & \text{si } l \text{ est une couche de sortie,} \\ \left(\sum_{p=1} \delta_p^{(l+1)} w_{jp}^{(l)}\right) \phi_j^{(l)\prime}(z_j^{(l)}) & \text{si } l \text{ est une couche cachée,} \end{cases} \tag{1.11}$$

avec :

$$L \equiv \frac{1}{2} \sum_{o=1}^{m} \left(a_o^{(d)} - y_o\right)^2,$$
$$a_j^{(l)} \equiv \phi_j(z_j^{(l)}),$$
$$z_j^{(l)} \equiv \sum_{p=0}^{u} w_p j a_p^{(l-1)}.$$

Étant donnée l'activation $a_i^{(l-1)}$ du i-ème neurone de la couche $l-1$ et le terme d'erreur $\delta_j^{(l)}$ du j-ième neurone de la couche l, il est possible de calculer la dérivée partielle $\partial L^{(k)} / \partial w_{ij}^{(l)}$:

$$\frac{\partial L(\boldsymbol{x}^{(k)}, \boldsymbol{y}^{(k)}, \theta)}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)} \tag{1.12}$$

Preuve La preuve est comparable à la description de la descente de gradient du perceptron. Nous utilisons deux fois le théorème de la dérivée de fonction composée pour obtenir :

$$\frac{\partial L(\boldsymbol{x}^{(k)}, \boldsymbol{y}^{(k)}, \theta)}{\partial w_{ij}^{(l)}} = \frac{\partial L}{a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} \tag{1.13}$$

Le dernier terme représente la dérivée partielle de la somme des entrées pondérées en fonction du poids w_{ij} :

$$\frac{\partial z}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}} \left(\sum_{p=1}^{u} w_{pj} a_p^{(l-1)} \right)$$

L'entrée pondérée $w_{ij}a_i^{(l)}$ est le seul terme de la somme non constant en fonction de w_{ij}. Nous obtenons donc :

$$\frac{\partial z}{\partial w_{ij}} = a_i^{(l-1)}. \tag{1.14}$$

Soit $\delta_j^{(l)} = \frac{\partial L}{\partial a_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}}$. Si j est une neurone de la dernière couche, c'est-à-dire, $l = d$, nous avons :

$$\begin{aligned} \delta_j^{(l)} &= \frac{\partial}{\partial a_j^{(l)}} \left(\frac{1}{2} \sum_{p=0}^{m} \left(a_p^{(l)} - y_o^{(k)} \right)^2 \right) \frac{\partial}{\partial z_j^{(l)}} \left(\phi_j^{(l)}(z_j^{(l)}) \right) \\ &= (a_j^{(l)} - y_j)\phi'(z^{(l)}). \end{aligned} \tag{1.15}$$

Dans le cas contraire, j appartient à une couche cachée (c'est-à-dire, $1 \leq l < d$), et nous pouvons développer $\delta_j^{(l)}$:

$$\begin{aligned} \delta_j^{(l)} &= \sum_{t=1} \frac{\partial L}{\partial a_t^{(l+1)}} \frac{\partial a_t^{(l+1)}}{\partial a_j^{(l)}} \phi'(z^{(l)}) \\ &= \sum_{t=1} \frac{\partial L}{\partial a_t^{(l+1)}} \frac{\partial a_t^{(l+1)}}{\partial z_t^{(l)}} \frac{\partial z_t^{(l+1)}}{\partial a_j^{(l)}} \phi'(z^{(l)}) \end{aligned}$$

Le terme $\left(\frac{\partial L}{\partial a_t^{(l+1)}} \frac{\partial a_t^{(l+1)}}{\partial z_t^{(l)}} \right)$ est le terme d'erreur $\delta_t^{(l+1)}$ pour le neurone t de la couche $l+1$. Nous avons donc :

$$\delta_j^{(l)} = \sum_{t=1} \delta_t^{(l+1)} w_{jt} \phi'(z^{(l)}). \tag{1.16}$$

À partir des équations 1.14, 1.15 et 1.16, nous avons :

$$\frac{\partial L(\boldsymbol{x}^{(k)}, \boldsymbol{y}^{(k)}, \theta)}{\partial w_{ij}^{(l)}} = \delta_j^{(l)} a_i^{(l-1)}$$

avec

$$\delta_j^{(l)} = \begin{cases} (a_j^{(l)} - y_j)\phi'(z^{(l)}) & \text{si } l = d, \\ \sum_{t=1} \delta_t^{(l+1)} w_{jt}\phi'(z^{(l)}) & \text{sinon.} \end{cases}$$

1.2.4 Convergence de l'apprentissage

L'apprentissage par descente de gradient permet d'optimiser les paramètres du réseau de neurones par rapport à la fonction d'erreur empirique J_{train}. Du fait du modèle des réseaux de neurones, cette fonction est généralement non-convexe. C'est-à-dire qu'elle contient plusieurs minimums locaux. En pratique, il n'est pas nécessaire d'atteindre un minimum globale sur la fonction d'erreur J_{train} car celui mène généralement à un cas de sur-apprentissage, comme nous allons le voir dans le paragraphe suivant. Le taux d'apprentissage α est un paramètre important à prendre en compte. Trop petit, l'apprentissage est lent et le risque de tomber dans un minimum local peu intéressant est important, et trop grand, la recherche des paramètres risque de diverger [65, 94]. Nous verrons dans la Section 1.2.5 des méthodes permettant d'adapter automatiquement le taux d'apprentissage lors de la descente de gradient.

Problème du sur-apprentissage Un problème classique en apprentissage automatique est le sur-apprentissage (ou *overfitting* en anglais) de la base d'entrainement. Ce problème arrive lorsque le modèle appris commence à s'adapter aux cas particuliers de la base de données au détriment du cas général. Ce cas de figure est repris dans la Figure 1.11. Ce phénomène provient d'une base de données d'apprentissage pas assez grande comparée à la complexité du modèle d'apprentissage. Les réseaux de neurones étant des modèles très complexes avec un nombre de paramètres particulièrement important, leur apprentissage nécessite de faire particulièrement attention à ce phénomène de sur-apprentissage, en particulier dans le cas où la base de données d'apprentissage est relativement petite.

Ensemble de validation et cross-validation Dans les modèles d'apprentissage automatique, tels que les réseaux de neurones, le phénomène de sur-apprentissage peut être détecté par l'utilisation d'un ensemble de données non utilisées durant l'apprentissage, appelé ensemble de validation (environ entre 10% et 50% de la base d'apprentissage). Au cours de l'apprentissage, la fonction de perte est calculée sur l'ensemble

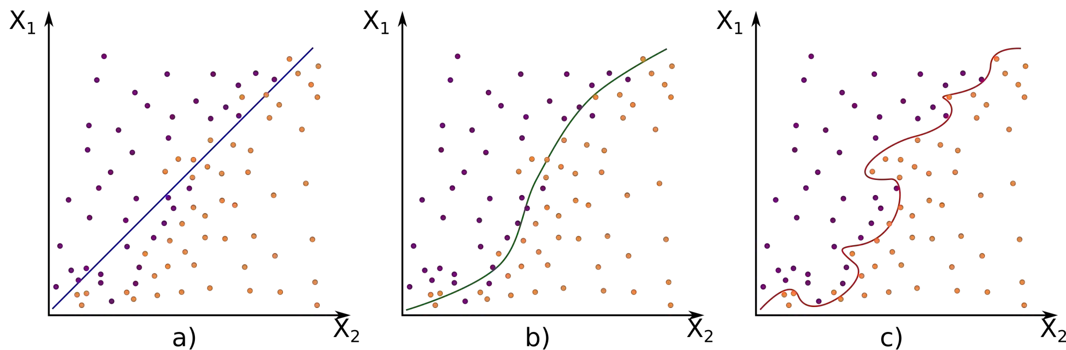


FIGURE 1.11 – Trois modèles de classifieurs à différents niveaux d'apprentissage : a) classifieur en sous-apprentissage, b) classifieur bien appris et c) un classifieur en sur-apprentissage. Les points violets et orange sont les données des deux différentes classes.

de validation de manière régulière comme simple observation (elle n'est pas utilisée pour le calcul du gradient). Le sur-apprentissage est observé lorsque la fonction de perte commence à remonter sur l'ensemble de validation alors qu'elle continue à baisser sur l'ensemble d'apprentissage (voir Figure 1.12). Ce point de rupture marque la spécialisation du modèle sur les données d'apprentissage au détriment de la capacité de généralisation du modèle (c'est-à-dire, la capacité à traiter des nouvelles données).

Early-stopping La méthode la plus simple pour éviter le sur-apprentissage consiste à arrêter la descente de gradient lorsque que la fonction de perte calculée sur l'ensemble de validation commence à augmenter et que la fonction de perte calculée sur l'ensemble d'apprentissage continue de descendre. Des méthodes ont été proposées afin de détecter automatiquement ce point de rupture lors d'un apprentissage [88].

Régularisation Le sur-apprentissage étant dû à l'apprentissage d'un modèle trop complexe par rapport au problème initial, des techniques pour l'éviter consistent à pénaliser cette complexification du modèle. Pour les réseaux de neurones, cela consiste à pénaliser les poids de connexions trop importants en ajoutant un terme de régularisation à la fonction de coût J à minimiser. Ce terme pénalise les poids des connexions trop fortes. Une autre solution de régularisation, appelé *Dropout*, a été proposée spécifiquement pour les réseaux de neurones [100]. Elle consiste à "déconnecter" des neurones pris aléatoirement à chaque itération de manière provisoire. Ces neurones ne participent donc pas à la sortie du réseau de neurones pendant une itération de

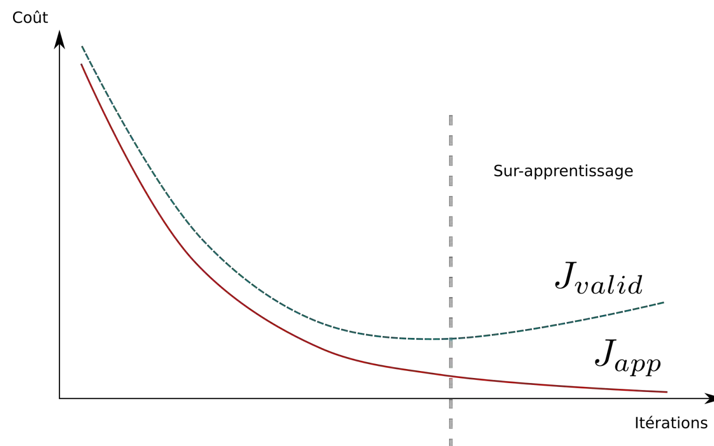


FIGURE 1.12 – Fonction de coût calculée sur l'ensemble d'apprentissage J_{train} (courbe en rouge) et sur l'ensemble de validation J_{valid} (courbe en vert) en fonction du nombre d'itérations. Lorsque J_{valid} commence à remonter alors que J_{train} continue de descendre, le modèle entre en phase de sur-apprentissage.

l'apprentissage. Les autres neurones doivent donc compenser cette absence. Cela a pour effet de rendre le réseau de neurones plus robuste face au bruit et donc d'éviter le surapprentissage.

1.2.5 Alternatives à la descente de gradient

Descente de Gradient Stochastique

La descente de gradient est un algorithme d'optimisation très populaire mais il existe de nombreuses variantes utilisées pour entraîner les réseaux de neurones. Une variante existante est la version stochastique de la descente de gradient que nous appellerons descente de gradient stochastique. Soit J la fonction de coût à minimiser en fonction d'un vecteur de paramètres **w** tel que :

$$J(\mathbf{w}) = \frac{1}{m} \sum_{i=1}^{m} L_i(\mathbf{w})$$

avec L_i est la i-ème observation faite sur la base d'apprentissage (typiquement l'erreur faite par la i-ème donnée de la base d'apprentissage). Au lieu de calculer le gradient de $J(\mathbf{w})$ sur l'ensemble des observations L_i pour modifier les paramètres **w**, ce gradient est approximé en le calculant sur une seule observation L_i. À chaque itération, les

paramètres **w** sont modifiés de la façon suivante :

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla L_i(\mathbf{w}_t)$$

Cette méthode requiert de calculer le gradient juste pour une seule entrée de la base d'entraînement réduisant considérablement le coût par itération. Elle est particulièrement utile lorsque la base de données n'est pas entièrement accessible ou trop large pour être mise en mémoire. De plus, la descente de gradient stochastique permet, par des itérations plus courtes, de s'approcher de la solution optimale de **w** plus rapidement. Cependant cette approximation du gradient sur une unique observation implique de faire un nombre plus important d'itérations avec un taux d'apprentissage plus petit. Un bon compromis consiste à prendre, non pas une seule observation, mais un batch (c'est-à-dire, entre quelques dizaines ou centaines d'observations) d'une taille b. Le gradient est alors mieux approximé, ce qui permet d'utiliser un taux d'apprentissage plus raisonnable pour converger rapidement.

Momentum

Il est encore possible d'accélérer la descente de gradient par la méthode du momentum [91]. Le nom de cette méthode provient du domaine de la physique, où le momentum représente l'inertie dans le mouvement d'un objet. L'idée consiste à garder une inertie également dans la recherche de gradient en gardant une trace des dernières modifications pour calculer le déplacement des paramètres de la fonction. Cette méthode est généralement associée à la descente de gradient avec des petits batches. La règle de mise à jour est alors la suivante :

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \Delta \mathbf{w}_t$$

avec

$$\Delta \mathbf{w}_t = \gamma \Delta \mathbf{w}_{t-1} + \frac{1}{b} \sum_{i \in X_t} \nabla L_i(\mathbf{w}_t)$$

avec X_t le batch de données utilisé à l'itération t, et $\gamma \in]0,1]$. Plus γ est proche de 1 et plus la recherche dans l'espace des paramètres a un momentum important.

Momentum de Nesterov Une variante de la méthode du momentum est appelée méthode du momentum de Nesterov proposée dans les travaux de Sutskever *et al.* [101] afin d'améliorer l'apprentissage des réseaux de neurones. Au lieu de calculer le gradient pour $\mathbf{w}_t$ à l'itération t, les auteurs proposent de calculer ce dernier au point $\widetilde{\mathbf{w}}_t$, qui correspond aux paramètres actuels θ_t plus le momentum, c'est-à-dire tel que :

$$\widetilde{\mathbf{w}}_t = \mathbf{w}_t + \gamma \Delta \mathbf{w}_{t-1}.$$

La modification à l'itération t est donc égale à :

$$\Delta \mathbf{w}_t = \beta \Delta \mathbf{w}_{t-1} + \frac{1}{b} \sum_{i \in X_t} \nabla L_i(\widetilde{\mathbf{w}}_t).$$

Les méthodes de second ordre

La descente de gradient est une méthode dite de premier ordre, c'est-à-dire qu'elle optimise les paramètres d'une fonction en utilisant sa dérivée première. Il existe d'autres méthodes d'optimisation tel que la méthode de Newton qui utilise la dérivée seconde de la fonction à minimiser afin de trouver un extremum.

Le calcul de la Hessienne, ou son approximation demande de calculer la matrice des dérivées partielles secondes ce qui correspond à une matrice de taille $n \times n$ à chaque itération (avec n le nombre de paramètres du réseau de neurones). Dans le cas de réseaux de neurones modernes, nous rappelons que n peut être de l'ordre de 10^6 voir 10^7. Ces techniques sont efficaces mais nécessitent d'approximer la Hessienne par différentes méthodes pour pouvoir fonctionner sur des réseaux de neurones classiques. Actuellement, elles sont rarement utilisées dans l'apprentissage de réseaux de neurones car d'autres méthodes se sont montrées tout aussi efficaces.

Autres techniques d'optimisation

Afin d'accélérer la descente de gradient, de nombreuses autres méthodes ont été proposées pour les réseaux de neurones tels que AdaGrad [30], Adadelta [116], RMSProp [1], ou Adam [58]. L'une des plus utilisées est la méthode d'optimisation Adam [58]. Cette méthode se base sur la descente de gradient avec de petits batches et reprend

1. Cette méthode proposée par G. Hilton n'a pas été publiée mais une description est diponible sur le page : http://www.cs.toronto.edu/ tijmen/csc321/slides/lecture_slides_lec6.pdf

les mêmes principes que AdaDelta. L'idée consiste à adapter le taux d'apprentissage à partir d'une estimation du premier et du second moment du gradient (contrairement à la méthode du momentum qui n'utilise que le premier moment). L'estimation du premier et second moment nécessite cependant le maintien et la mise à jour de deux variables supplémentaires pour chaque paramètre du réseau de neurones. Ce type de méthode à l'avantage d'être relativement robuste et permet de d'adapter automatiquement le taux d'apprentissage au cours de l'apprentissage pour chaque poids.

1.3 Les réseaux de neurones profonds

Dans la Section 1.2.2 nous avons vu comment il était possible d'apprendre des modèles tels que les réseaux de neurones, à accomplir certaines tâches. Cependant les réseaux de neurones ont très longtemps été limités dans leurs architectures, en particulier concernant leur profondeur, c'est-à-dire, le nombre de couches qu'ils pouvaient apprendre. Cette limitation s'est effondrée dans les années 2010-2012 avec l'arrivée de bases de données bien plus grandes (telles que [92]) accompagnées de capacités de calcul et de stockage plus importantes. Cette avancée a également été permises par des architectures de réseaux de neurones différentes, plus faciles à apprendre et mieux adaptées à certains types de données.

1.3.1 L'intérêt des architectures profondes

Dans les algorithmes d'apprentissage classiques, des caractéristiques doivent être extraites des données brutes afin d'effectuer la tâche d'apprentissage. Le but étant d'avoir une représentation plus haut niveau des données. Par exemple, dans le domaine de l'analyse d'image, une première étape consiste à calculer les points d'intérêts (comme les SIFT [72]) et les regrouper dans des sacs de mots (ou *bag-of-words* en anglais) pour entraîner un modèle classique d'apprentissage tel qu'un arbre de décision, un SVM [95], une forêt d'arbres aléatoires ou même un réseau de neurones.

L'extraction de caractéristiques à partir des données brutes demande des bonnes connaissances sur celles-ci et sur la tâche d'apprentissage, ainsi qu'un travail d'ingénierie pour adapter les méthodes d'extraction. Cette opération est relativement coûteuse à la mise en place, dépend du contexte et une mauvaise extraction des caractéristiques mène à de très mauvaises performances en terme d'apprentissage. L'idée

des architectures profondes consiste à intégrer cette extraction de caractéristiques, normalement faite "à la main", par un processus d'apprentissage dans les premières couches du réseau de neurones (voir Figure 1.13).

Dans la Section 1.2.2, nous avons vu que les couches intermédiaires permettent de transformer la représentation des données d'entrée en une représentation plus haut-niveau. Durant la phase d'apprentissage, chaque couche d'un MLP apprend une représentation de son entrée qui doit être intéressante pour les couches suivantes. Les informations contenues dans chacune de ces couches vont devenir de plus en plus haut niveau.

Le terme *profond* réfère donc au nombre de couches des réseaux de neurones profonds entre l'entrée et la sortie. Un réseau avec une seule couche cachée est appelé réseau peu profond, et à contrario, un réseau avec plus de 2 couches cachées est dit *profond*. De nos jours, il est possible de trouver des réseaux avec une centaine, voir un millier de couches pour les plus profonds [102, 45].

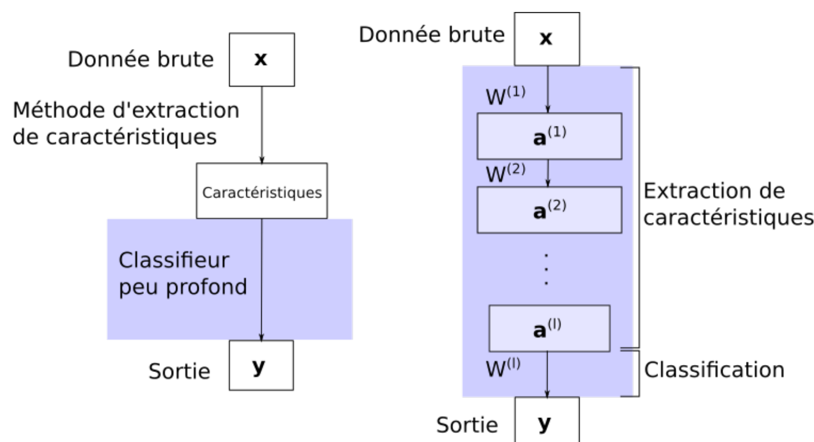


FIGURE 1.13 – La différence entre l'apprentissage automatique classique (à droite) et l'apprentissage profond (à gauche). La zone en bleu est la zone d'apprentissage.

1.3.2 Réseaux de neurones convolutifs

Les réseaux de neurones convolutifs (CNN) ont été introduits par Lecun *et al.* [66]. La particularité des CNN est l'utilisation de l'opération de convolution dans les premières couches intermédiaires du réseau de neurones. À l'origine, cette opération est utilisée comme filtre dans le domaine de l'image ou du son afin de mettre en évidence des motifs ou réduire un type de bruits. Dans les CNN, le modèle apprend lui-même les

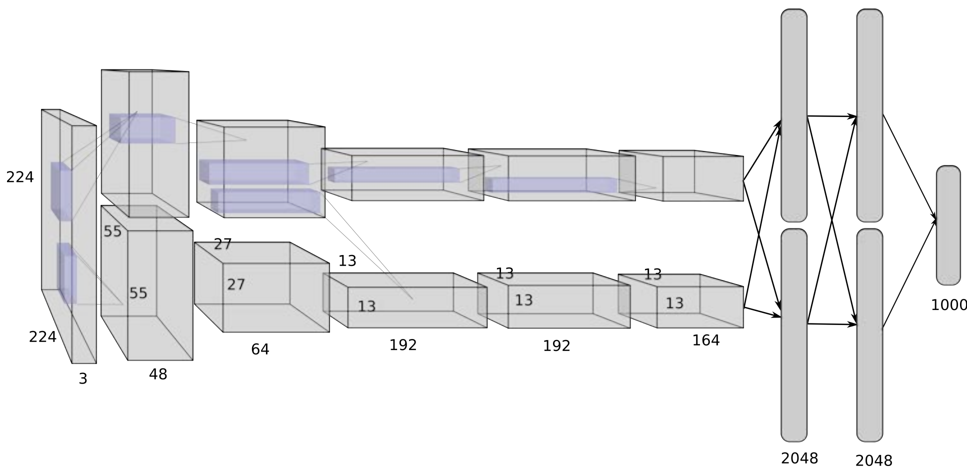


FIGURE 1.14 – Exemple du CNN appelé AlexNet [63]. Dans cette représentation, les neurones sont organisés en fonction des dimensions de largeur, hauteur et profondeur. Contrairement aux couches toute connectée, les couches convolutive garde une cohérence spatiale des informations. Chaque taille de dimension est indiqué sur la figure. En entrée, le réseau de neurones prend une image de 224×224 pixels avec 3 canaux de couleurs.

filtres des différentes convolutions afin de mettre en évidence les motifs des données d'entrée qui sont utilisés dans les couches suivantes. Un CNN classique est généralement composé de quatre types de couches :

— les couches convolutives, qui contiennent plusieurs opérations de convolutions appliquées sur la même entrée,
— les couches d'opérations de mise en commun,
— les couches d'activations,
— les couches toutes connectées.

Les couches convolutives À l'origine, l'opération de convolution est utilisée sur des données temporelles (sons) ou spatiales (images) en tant que filtre linéaire. Dans cette section, nous allons prendre l'exemple d'une opération de convolution 2D, utilisée sur des données telles que des images $X = (x_{i,j,z})_{1 \leq i \leq h, 1 \leq j \leq l, 1 \leq z \leq c}$ (avec $h \times l$, les dimensions de l'image et c le nombre de canaux). Cette opération est définie par un noyau $A = (a_{i,j,z,k})_{1 \leq i \leq m, 1 \leq j \leq n, 1 \leq z \leq c, 1 \leq k \leq f}$ où $m \times n$ est la largeur et la hauteur des filtres et f est le nombre de filtres, ainsi qu'un biais $\boldsymbol{\beta} \in \mathbb{R}^f$. La sortie de l'opération de convolution $Y \in \mathbb{R}^{h \times l \times f}$ est calculée tel que :

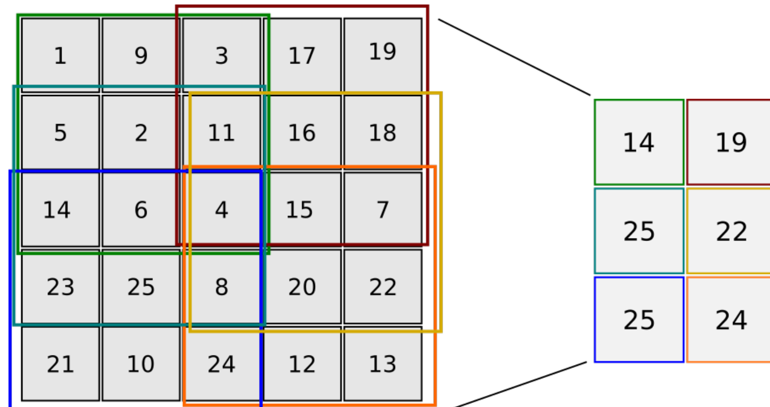


FIGURE 1.15 – Exemple d'opération de mise en commun avec la fonction *maximum*. La fenêtre glissante est de taille 3×3 et se déplace de 1 par 1 suivant l'axe y et 2 par 2 suivant l'axe x.

$$y_{i,j,k} = \sum_{i'=0}^{m} \sum_{j'=0}^{n} \sum_{z=0}^{c} x_{i+i',j+j',z} a_{i',j',z,k} + \beta_k$$

Afin de simplifier la formule, nous n'avons pas pris en compte la gestion des "bords" de l'image d'entrée dans l'opération de convolution. Les convolutions sont également applicables sur des données à une dimension (comme le son [7]) ou à trois dimensions (comme la vidéo ou un scanner 3D).

Opération de mise en commun Les couches convolutives peuvent être suivies d'une opération de mise en commun. Elle a pour but de réduire la dimension des couches de neurones en regroupant les informations présentes sur les neurones proches les uns des autres. Le principe est de faire déplacer une fenêtre glissante sur les neurones et d'y appliquer une opération de mise en commun. Il existe différents types d'opérations de mise en commun comme la fonction *maximum* ou *moyenne*. Un exemple d'opération de mise en commun est illustré dans la Figure 1.15.

Les couches toutes connectées et les couches d'activation sont identiques aux MLP (respectivement les couches de neurones classiques et la fonction d'activation appliquée à toute une couche). Les couches toutes connectées sont généralement placées à la fin des CNN (juste avant la couche de sortie). Elles permettent de mettre en corrélation tous les motifs détectés par les couches convolutives dans les couches précédentes. Les couches d'activation sont généralement placées après chaque couche de convolution et chaque couche toute connectée. Les couches d'activation et les couches

de mise en commun ne sont pas des couches de neurones car elles ne contiennent aucune connexion à apprendre (c'est-à-dire, aucun paramètre entrainable).

Contrairement au MLP, le nombre de paramètres à apprendre dans les CNN est généralement plus faible mais le nombre d'opérations reste plus important. En effet, les filtres, généralement de petite taille, sont partagés par les neurones d'une ou plusieurs dimensions de la couche de sortie. Les CNN sont principalement utilisés dans le domaine de l'image, dans lequel ils dépassent les autres méthodes d'apprentissage [63, 102]. Ils sont également utilisés dans le domaine du son [7] ou de la vidéo. Des variantes existent également pour l'analyse de graphes quelconques [80].

1.3.3 Réseaux de neurones récurrents

Alors que les CNN sont principalement utilisés pour faire ressortir des relations spatialement proches (comme des relations entre pixels proches dans une image), les réseaux de neurones récurrents (RNN) ont été développés afin de garder un contexte temporel pour chaque événement en entrée. Ils ont été particulièrement utilisés pour de l'analyse de séries temporelles, de données audio, ou de textes dans lesquelles le contexte est important afin d'analyser chaque nouvelle entrée. L'idée consiste à garder des informations au cours du temps à l'intérieur des couches de neurones afin de donner un contexte aux données analysées. La sortie du RNN, à un instant t, va dépendre non seulement de l'entrée à l'instant t mais également de l'état du RNN calculé à l'instant $t-1$.

Dans sa version la plus simple, une couche d'un RNN peut être décrite comme une couche toute connectée l qui prend en entrée la couche précédente $l-1$ à l'instant t concaténée à la sortie d'elle même (c'est-à-dire, couche l) à l'instant $t-1$. La Figure 1.16 a) représente une couche RNN.

Cellules LSTM Les cellules *Long Short-Term Memory* (en français les réseaux récurrents à mémoire courte et long terme) ont été introduites par Hochreiter *et al.* en 1997 [50]. Le but étant de faire face aux problèmes de disparition du gradient lors que l'élément courant et son contexte sont trop éloignés dans le temps. L'idée principale des cellules LSTM est de garder un état de mémoire $c \in [0,1]^n$ et 3 "portes" utilisées pour faire transiter l'information vers cette mémoire ou la faire sortir. Une première porte, dite *porte d'oubli*, sert à calculer les éléments de la mémoire c qui doivent être

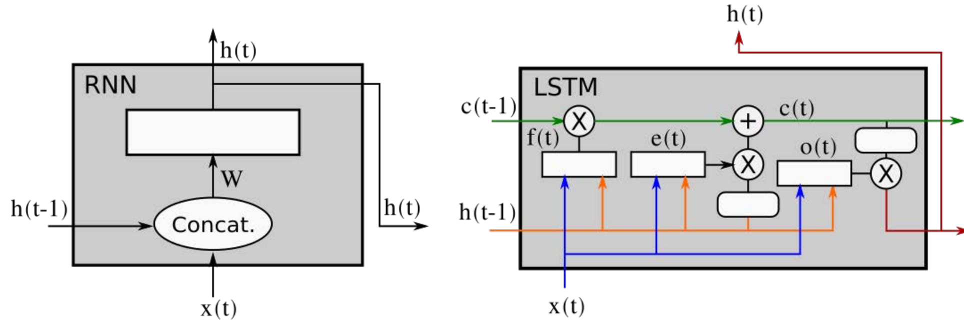


FIGURE 1.16 – a) Couche d'un RNN classique. b) Cellules LSTM.

oubliés (c'est-à-dire, mis à zéro). À l'itération t, cette porte calcule un vecteur :

$$f(t) = \phi_{sig}(W^{(f)}x(t) + U^{(f)}h(t-1) + \boldsymbol{\beta}^{(f)})$$

avec $x(t) \in \mathbb{R}^m$ l'entrée de la couche, $h(t-1) \in \mathbb{R}^n$ la sortie de la couche à l'itération $t-1$, une matrice de poids $W^{(f)} \in \mathbb{R}^{n \times m}$, une matrice de poids $U^{(f)} \in \mathbb{R}^{n \times n}$ et un biais $\boldsymbol{\beta}^{(f)}) \in \mathbb{R}^n$. La fonction ϕ_{sig} est la fonction sigmoïde. Les valeurs du vecteur $f(t)$ sont donc comprises dans $]0, 1[$. Une valeur proche de 1 (resp. 0) pour la i-ème entrée indique que la i-ème valeur de c doit être oubliée (resp. conservée en mémoire). Une seconde porte, dite *porte d'entrée*, sert à calculer quelle information doit être ajoutée à c. À l'itération t, cette porte calcule un vecteur :

$$e(t) = \phi(W^{(e)}x(t) + U^{(e)}h(t-1) + \boldsymbol{\beta}^{(e)}).$$

De même que pour la porte d'oubli, $W^{(e)} \in \mathbb{R}^{n \times m}$ et $U^{(f)} \in \mathbb{R}^{n \times n}$ représentent des matrices de poids et $\boldsymbol{\beta}^{(e)} \in \mathbb{R}^n$ un vecteur de biais.

Une fois ces deux vecteurs calculés, l'état de la mémoire peut être calculé de la manière suivante :

$$c(t) = f(t) \otimes c(t-1) + e(t) \otimes (\phi(W^{(c)}x(t) + U^{(c)}h(t-1) + \boldsymbol{\beta}^{(c)}),$$

avec $W^{(c)} \in \mathbb{R}^{n \times m}$ et $U^{(c)} \in \mathbb{R}^{n \times n}$ de nouvelles matrices de poids ainsi qu'un vecteur de biais $\boldsymbol{\beta}^{(c)} \in \mathbb{R}^n$. Le symbole $\otimes$ représente la multiplication élément par élément.

La dernière porte, dite *porte de sortie*, sert à calculer les informations envoyées à la sortie de la couche. Elle calcule le vecteur suivant :

$$o(t) = \phi(W^{(o)}x(t) + U^{(o)}h(t-1) + \boldsymbol{\beta}^{(o)}),$$

avec $W^{(o)} \in \mathbb{R}^{n \times m}$ et $U^{(o)} \in \mathbb{R}^{n \times n}$ de nouvelles matrices de poids ainsi que le vecteur de biais $\boldsymbol{\beta}^{(o)} \in \mathbb{R}^n$. La sortie de la couche peut donc être calculée en fonction de l'état $c(t)$ et la porte de sortie :

$$h(t) = o(t) \otimes \phi(c(t)).$$

Une représentation graphique est proposée dans la Figure 1.16 b). Les cellules LSTM ont été particulièrement utilisées à l'intérieur de réseaux de neurones profonds tels que [7]. Cependant, lorsque les relations entre les éléments sont trop éloignées dans le temps, comme cela peut arriver dans un texte ou une conversation (c'est-à-dire, plus d'une centaine de mots entre le contexte et l'élément courant), le LSTM ne suffit plus. Des modèles plus récents, tels que les modèles d'attention hiérachique [107] permettent de mettre en relation des éléments particulièrement éloignés dans le temps.

1.3.4 Techniques avancées pour améliorer l'apprentissage

Les derniers réseaux de neurones, en particulier dans le domaine de l'image ont un nombre de couches extrêmement importants (jusqu'à une centaine de couches pour certain réseau [45, 102]). Ceci a pour effet de réduire considérablement le gradient calculé dans les couches basses du réseau. Pour répondre à ce problème, de nombreuses solutions ont été proposées ces dernières années.

Module Inception (GoogleNet)

Proposé en 2014 par C. Szegedy *et al.* [102], le réseau de neurones appelé GoogleNet gagne le challenge ILSVRC (challenge de classification d'images sur la base de données ImageNet) cette même année en proposant de multiples améliorations à l'architecture du CNN. La plus notable est l'utilisation de modules à branches appelés module d'Inception. L'idée de base est de multiplier les filtres avec des tailles différentes. Dans la Figure 1.17 a), le module contient 4 branches, 3 couches de convolutions avec des filtres de taille 5×5, 3×3 et 1×1, et une opération de mise en commun avec des fenêtres de taille 3×3. Les opérations de convolution 1×1 consistent à mettre en relation uniquement les différents canaux sur une même position de l'image . Afin de réduire la quantité de calcul et le nombre de paramètres à apprendre, les auteurs proposent d'ajouter des convolutions 1×1 peu coûteuses pour réduire le nombre de

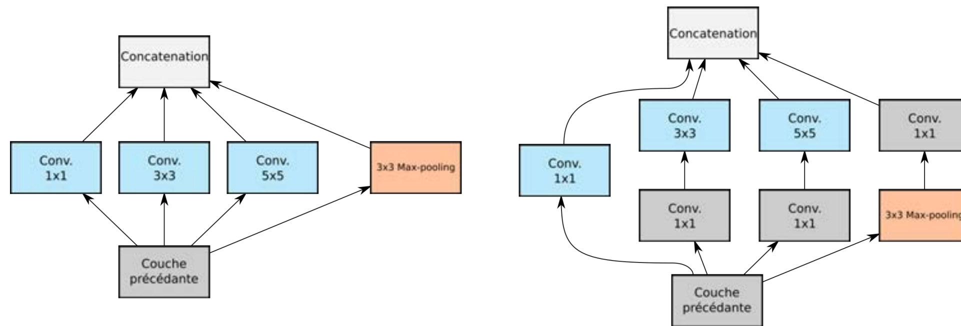


FIGURE 1.17 – Module Inception dans sa version simple a) et avec réduction de dimension b). Dans cette représentation graphique, le bloc Conv 3x3 représente une couche de convolution avec des filtres de taille 3 par 3. Max-pooling représente une opération de mise en commun avec la fonction *maximum*.

canaux avant les opérations de convolution 3×3 et 5×5. Cette version du module Inception avec une réduction de dimension est représentée dans la Figure 1.17 b).

Normalisation par batch

La technique de normalisation par batch a été proposée en 2015 par S. Ioffe *et al.* [54]. Le but est de contourner le problème lié à l'apprentissage de successions de couches dépendantes les unes des autres. Lorsqu'une couche l est modifiée lors de l'apprentissage, la représentation intermédiaire des données en entrée de la couche $l+1$ se retrouve modifiée et donc celle-ci doit ré-apprendre ses paramètres en fonction de cette nouvelle représentation.

Une solution pour réduire les perturbations dues à ce changement de représentation intermédiaire en entrée d'une couche consiste à normaliser l'activation de chaque neurone d'une couche suivant l'activation de celui-ci sur tout un batch. Une fois l'activation des neurones normalisée, celle-ci est "dénormalisée" à l'aide de deux variables μ_i, β_i pour chaque neurone, que le réseau doit apprendre en plus. Cette dénormalisation est nécessaire pour garder la capacité de représentation du réseau de neurones. Cette méthode a pour effet de rendre l'apprentissage plus stable vis-à-vis de l'initialisation du réseau de neurones profond et le taux d'apprentissage choisi (celui-ci peut donc être plus important). La normalisation par batch est maintenant utilisée dans une grande partie des réseaux de neurones de l'état de l'art.

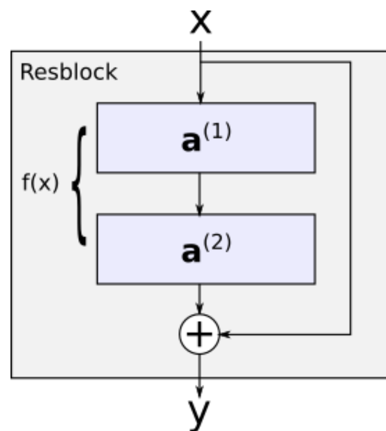


FIGURE 1.18 – Exemple d'un ResBlock classique avec deux couches intermédiaire. La fonction $f(x)$ est représentée pas les deux couches de neurones $\mathbf{a}^1$ et $\mathbf{a}^2$.

ResNet

Pour réduire la disparition du gradient sur un large nombre de couches, K. He *et al.* [45] proposent de modifier la sortie de certains blocs de couches du réseau de neurones pour obtenir :

$$\mathbf{y} = f(\mathbf{x}) + \mathbf{x},$$

avec **x**, l'entrée du bloc, $f(x)$ la fonction utile, (c'est-à-dire, celle utilisée pour remplir la tâche). L'idée d'inclure l'identité dans la fonction en sortie permet de ne pas perdre d'information sur la donnée en entrée, au fil des couches du réseau de neurones. Cette modification est faite en utilisant un type de bloc appelée bloc de ResNet (utilisé dans les réseaux de neurones appelés *Residual Neural Network*) décris dans la Figure 1.18. L'idée consiste donc à contourner l'opération d'apprentissage avec une addition de l'entrée à la sortie du bloc (suivi généralement d'une fonction d'activation). Les auteurs ont montré qu'ils étaient capables d'entraîner des réseaux de neurones avec jusqu'à $1\ 202$ couches cachées grâce à cette méthode.

1.3.5 Réseaux de neurones non supervisés

Nous avons vu au début du chapitre que lorsqu'aucun label y n'est disponible dans la base de données d'apprentissage B_{train}, il n'est pas possible d'utiliser les méthodes d'apprentissage dites supervisées vues jusqu'ici. Dans cette section, nous allons voir

les architectures de réseaux de neurones adaptés à l'apprentissage non supervisé.

Auto-encodeurs

Les auto-encodeurs sont des réseaux de neurones séparés en deux parties : un encodeur et un décodeur (voir Figure 1.19). Le but est d'apprendre comment réduire le nombre de dimensions de manière intéressante par un encodage. L'encodeur est un réseau de neurones qui transforme les données d'entrée $x \in \mathbb{R}^n$ dans un nouvel espace $\mathbb{R}^m$ avec $n > m$. Inversement, le décodeur retransforme les données x de l'espace $\mathbb{R}^m$ dans l'espace $\mathbb{R}^n$.

L'apprentissage de ces deux réseaux de neurones se fait de manière simultanée. Le but étant de minimiser l'erreur de "reconstruction" du décodeur sur les données encodées par l'encodeur :

$$J = \|x - f_\theta(g_\theta(x))\|^2$$

avec $g_\theta(x)$, la fonction paramétrique représentant l'encodeur avec ses paramètres θ et $f_\theta(x)$ la fonction paramétrique représentant le décodeur avec ses paramètres θ.

Le codage (c'est-à-dire, la représentation intermédiaire sur $\mathbb{R}^n$) étant un espace plus petit, en terme de dimension, que l'espace d'entrée, le réseau de neurones (encodeur et décodeur) doit apprendre à compresser au mieux l'information en entrée afin de la restituer lors du décodage avec le moins de perte possible.

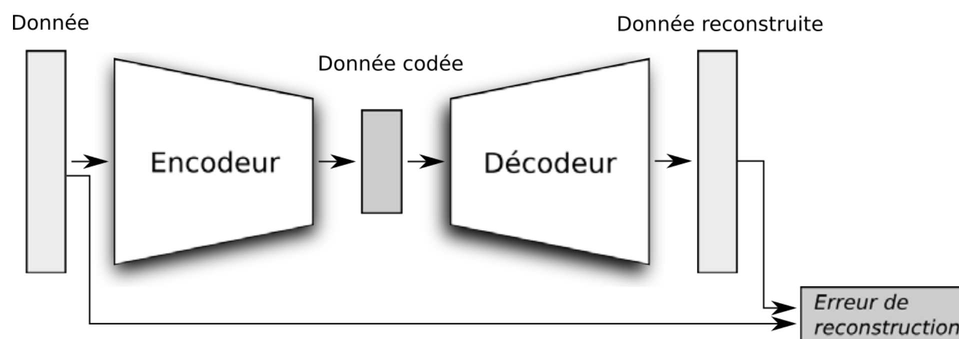


FIGURE 1.19 – Exemple d'un auto-encodeur. Les blocs "Encodeur" et "Décodeur" représentent tous deux des réseaux de neurones.

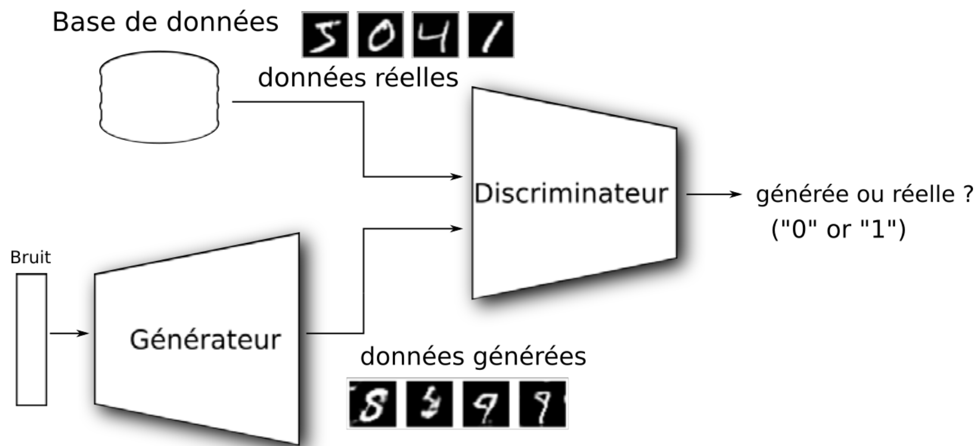


FIGURE 1.20 – Architecture du GAN avec le générateur et le discriminateur.

Les réseaux antagonistes génératifs (GAN)

Les réseaux antagonistes génératifs (appelés GAN) ont été introduits par I. Goodfellow [39]. Le principe des réseaux antagonistes est la minimisation de multiples fonctions avec des objectifs antagonistes. Ils ont été utilisés pour faire des contre-exemples afin de tromper des réseaux de neurones classiques [84], ou pour trouver de manière automatique des techniques de cryptage [1]. Dans le cas des réseaux antagonistes génératifs, l'idée consiste à apprendre à un réseau de neurones à générer des exemples réalistes qui pourraient appartenir à la base de données d'apprentissage. Ce réseau de neurones est décomposé en deux modèles différents : un générateur et un discriminateur (voir Figure 1.20). Le générateur prend en entrée un vecteur de variables aléatoires et l'associe à une sortie dans l'espace des données $\mathcal{X}$. Le discriminateur est un classifieur binaire qui prend en entrée des données de la base d'apprentissage et des données de la sortie du générateur. Son but est d'apprendre à différencier ces deux types de données. Le but du générateur est de tromper le discriminateur lorsqu'il génère des données. Au cours de l'apprentissage, la distribution de ces données en sortie se rapproche peu à peu de la distribution des données de la base d'apprentissage (voir Figure 1.21).

Variantes Ce type de réseaux de neurones est devenu très populaire dans le domaine de la recherche. De nombreuses améliorations on été proposées telles que [9, 8, 93, 47]. Toutes ces différentes variantes sont également utilisées dans de nom-

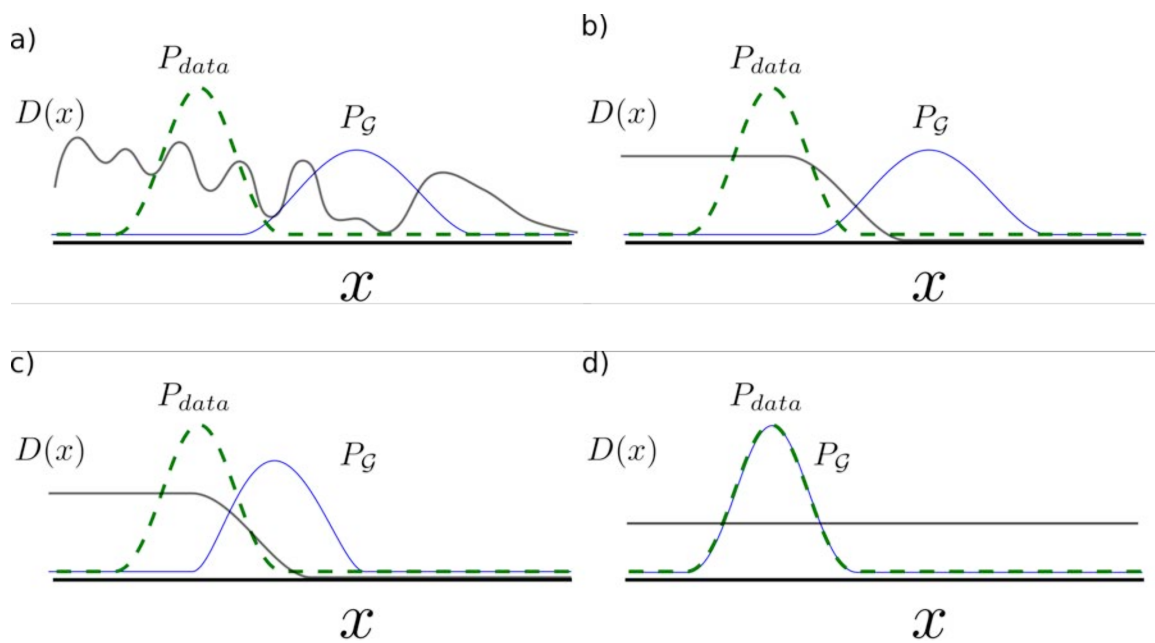


FIGURE 1.21 – Distribution des données réelles P_{data}, des données générées $P_{\mathcal{G}}$ et sortie du discriminateur $D(x)$ dans l'espace $\mathcal{X}$ au cours de l'apprentissage. Au début de l'apprentissage (a), le discriminateur est incapable de différencier les données réelles des données générées. Après quelques itérations, le discriminateur est capable de trouver la frontière entre les données réelles et les données générées (b). Afin de tromper le discriminateur, le générateur doit modifier ses paramètres afin de générer des données de plus en plus proches des données réelles (c), jusqu'à avoir la même distribution que celles-ci (d). Le discriminateur est alors incapable de différencier les données générées des vraies.

breuses applications [73, 55, 109].

Le simple but de générer des nouvelles données a un intérêt limité dans l'utilisation des GAN. Des variantes existantes dans lesquelles des informations supplémentaires sont données au générateur afin de produire des données. Par exemple, dans les travaux de A. Odena *et al.* [82], les auteurs proposent d'ajouter la classe de la donnée en entrée du générateur. Cela permet au générateur de générer un type de donnée particulier, en plus d'améliorer son apprentissage. Avec un principe assez proche, il est possible de générer une image à partir d'un texte [90]. Celui-ci est encodé et donné à l'entrée du générateur ainsi qu'à l'intérieur du discriminateur. Les GAN peuvent également être utilisés pour augmenter la résolution d'une image en utilisant celle-ci en entrée [68]. De même, certains GAN peuvent être utilisés afin de changer le style d'une image par celui d'un artiste [121].

1.4 Conclusion

Dans ce chapitre nous avons montré un aperçu de l'état de l'art de l'apprentissage profond. Celui-ci est particulièrement riche et le nombre d'applications augmente constamment depuis quelques années. Les réseaux de neurones peuvent être appris de manière supervisée ou non-supervisée. L'apprentissage se fait par une descente de gradient (ou une variante de cette méthode) sur une base de données la plus large possible. Nous allons voir dans le chapitre suivant les motivations pour effectuer cet apprentissage dans un système distribué ainsi que les premier travaux qui permettent de paralléliser cette descente de gradient sur plusieurs machines.

CHAPITRE 2

VERS UN APPRENTISSAGE PROFOND SUR DES SYSTÈMES DISTRIBUÉS

Dans ce chapitre nous allons décrire l'exemple d'un apprentissage collaboratif qui motive l'intérêt de l'apprentissage profond sur des systèmes distribués. Nous allons ainsi pouvoir détailler les contraintes liées aux systèmes distribués dans ce contexte. Ensuite, nous donnerons un aperçu de l'état de l'art concernant l'apprentissage distribué. Celui-ci est généralement limité à une exécution à l'intérieur même de *datacenters* dans lesquels les serveurs sont équipés de plusieurs processeurs puissants ou de cartes graphiques. Ce chapitre a pour but d'expliquer les contraintes liées à un environnement distribué plus ouvert ainsi que les différentes méthodes pour paralléliser l'apprentissage. Nous verrons au cours de ce chapitre qu'il existe certaines méthodes actuellement capables d'effectuer un apprentissage profond sur ce genre de systèmes distribués à large échelle.

2.1 Collaboration pour l'apprentissage d'un réseau de neurones profond

La motivation de nos travaux consiste en l'apprentissage d'un réseau de neurones profond à l'aide des ressources fournies par de nombreux participants. Dans cette "application", nous supposons un service permis par un réseau de neurones profond, comme par exemple, un assistant vocal personnel tel que Alexa ou Siri. Afin de construire le modèle nécessaire à ce service, nous proposons aux futurs utilisateurs de participer à l'apprentissage de celui-ci. Pour cela, ils mettent à disposition leur(s) machine(s) ainsi que leurs données pour servir de base à l'apprentissage. Plus précisément, les données (par exemple, des requêtes vocales) sont stockées et mises à disposition sur ces machines. Ces dernières se chargent d'en extraire les connais-

sances via une étape d'apprentissage (c'est-à-dire, une descente de gradient comme expliqué dans la Section 1.2.2) et partagent ces connaissances entre elles à travers Internet.

L'avantage de ce type de collaboration est de tirer partie des machines participantes lorsqu'elles ne sont pas (ou peu) utilisées. De plus, il n'est plus nécessaire de fabriquer et de réunir une grande base de données dans un *datacenter* puisque le modèle utilise les données présentes sur ces machines. Enfin, l'utilisateur peut avoir l'assurance que ses données, utiles à l'apprentissage, restent sur sa machine. Une fois l'apprentissage fini, il peut profiter du service apporté par le réseau de neurones profond.

Ce type de travail collaboratif a été expérimenté notamment avec le projet SETI @home [1], dans lequel les participants mettent à disposition leur ordinateur afin d'analyser des petits bouts de séquences d'enregistrements réalisées par un radiotélescope à Puerto Rico. Le but de cette analyse était de trouver des fréquences prouvant l'existence de signes de vie en dehors de la terre. Bien qu'aucune vie extra-terrestre n'ait été détéctée, le projet a permis de mettre en évidence les capacités d'un travail collaboratif avec un grand nombre de participants. En effet, plus de 100 000 internautes actifs ont participé à ce projet. De nombreux autres projets similaires ont également vu le jour avec des buts différents. Plus récemment, en 2016, les ingénieurs de recherche de Google ont proposé une solution appelée *Federated Learning* [2] afin de mettre en commun l'apprentissage d'un réseau de neurones profond effectué sur les machines de ses utilisateurs. Ils expérimentent actuellement cette méthode avec l'application GBoard sur Android afin de faire des propositions de requêtes automatiques. Nous présenterons cette méthode plus en détail à la fin de ce chapitre.

Les machines que nous considérons lors de la collaboration peuvent aussi bien être des téléphones mobiles, des tablettes connectées, des ordinateurs personnels ou bien même des *gateways* ou des *set-top box*. Il est évident qu'en terme de puissance, ces machines sont bien plus limitées que les ressources disponibles à l'intérieur d'un *datacenter*. Par exemple, il n'est pas envisageable de supposer que chacune dispose d'une carte graphique. De plus, les données contenues sur ces machines correspondent aux données d'un seul utilisateur (ou d'une famille). La variété et la quantité de données sur une unique machine ne sont pas suffisantes pour l'apprentissage d'un modèle tel qu'un réseau de neurones profond. Il est donc nécessaire que l'apprentissage soit par-

1. Voir le site du projet : https://setiathome.berkeley.edu/
2. Voir l'article sur les applications du *Federated Learning* : https://ai.googleblog.com/2017/04/federated-learning-collaborative.html

tagé par de nombreuses machines d'utilisateurs. Cela permet de mettre en commun les connaissances tirées d'un plus grand nombre de données sans partager directement ces données. Le réseau de neurones profond obtenu à la fin de l'apprentissage devrait donc être capable d'avoir une très bonne capacité de généralisation.

Nous allons voir dans la section suivante en quoi ce type de collaboration se rapporte aux problématiques bien connues des systèmes distribués.

2.2 Systèmes distribués et contraintes

Le domaine des systèmes distribués est connu de la recherche, particulièrement depuis les années 1980-1990. Il a donné lieu à de nombreux outils, comme les bases de données distribuées, le calcul parallèle, et plus récemment la technologie de la blockchain [105].

Un système distribué est défini comme un ensemble de noeuds de calcul indépendants et vus par les utilisateurs comme un système unique. Cela peut correspondre à un ensemble de processeurs reliés par une mémoire partagée ou un ensemble de machines interconnectées avec chacune leur propre mémoire qui peuvent communiquer entre elles par messages. Cet ensemble de nœuds a une tâche commune, tel que la gestion d'une grande base de données, ou le calcul de modèles de grosses tailles (tels que ceux utilisés pour la météorologie). L'intérêt d'un système distribué est de permettre le partage des ressources de ces différents nœuds à travers une mémoire partagée ou un réseau de communication. Cependant, chaque nœud a sa propre notion du temps et il n'est pas possible de tous les synchroniser à chaque opération (du fait des temps de communications qui peuvent être longs par rapport au temps de calcul). Cela mène à des problèmes de concurrence entre les évènements survenant sur chacun des nœuds. Par exemple, si nous supposons une base de données gérée par plusieurs machines, il est possible que deux opérations d'écriture sur la même variable soient effectuées sur des machines différentes de manière concurrente, c'est-à-dire, à un temps d'exécution trop proche pour que les machines se synchronisent. Si aucune mesure n'est prise, la base de données n'est plus cohérente : elle peut donner deux valeurs différentes suivant la machine interrogée par les utilisateurs. Ce temps de communication et la possibilité d'événements concurrents obligent les systèmes distribués à mettre en place des algorithmes capables de résister à ces incohérences locales ou de synchroniser correctement les différents noœuds. De plus, ces algorithmes doivent

être capables de faire face à de nombreuses autres contraintes :

— Le temps de communications entre les nœuds d'un système distribué n'est pas nécessairement bornées. Il faut donc généralement considérer que les communications sont asynchrones.

— Des pannes peuvent apparaître à tout moment au niveau du réseau ou directement au niveau des machines lors de l'exécution de l'algorithme. L'algorithme doit donc être capable de les gérer (ou de les tolérer).

Dans notre problématique, nous considérons principalement les systèmes distribués avec des machines interconnectées. Les systèmes multi-processeurs, telles que les cartes graphiques, ont déjà été largement étudiés dans le cadre de l'apprentissage profonds et utilisés à l'intérieur des *datacenters*. Ils ne sont pas applicables à notre contexte d'apprentissage collaboratif. Il n'est cependant pas exclu que les participants à l'apprentissage utilisent des systèmes multi-processeurs sur leur machine.

Apprentissage sur un système distribué Dans notre contexte d'apprentissage distribué d'un réseau de neurones profond, chaque machine a accès à une base de données locale. Soit B_i l'ensemble des données en local sur la machine i, nous pouvons définir la base de données d'apprentissage distribuée comme étant $B_{train} = \bigcup_i B_i$. Le but est de trouver les paramètres du réseau θ qui minimisent la fonction d'erreur calculée sur B_{train}. Soit J cette fonction d'erreur tel que :

$$J(\theta) = \frac{1}{|B_{train}|} \sum_{j \in B_{train}} L_j(\theta)$$

avec $L_j(\theta)$ l'erreur faite sur la j-ème donnée de B_{train} avec les paramètres θ. Nous pouvons définir J_i, la fonction d'erreur locale à un agent i. Dans le cas où chaque machine a le même nombre de données dans sa base B_i, la fonction d'erreur totale peut s'écrire :

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} J_i(\theta) = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{|B_i|} \sum_{j \in B_i} L_j(\theta)$$

Cette problématique peut donc se ramener à minimiser la moyenne d'une fonction d'erreur J_i qui peut être calculée en local sur chaque machine i. En pratique, il est peu probable que B_i soit de la même taille pour toutes les machines i, mais il est possible de supposer que l'ordre de grandeur reste le même. Pour une simplification

des notations nous considérons par la suite que θ est composé d'un unique vecteur de paramètres **w**.

Cohérence des paramètres Le but des différentes machines, que nous appellerons *agents* lors de l'apprentissage, est donc de trouver les paramètres **w** qui minimisent J. Comme nous l'avons vu, cet apprentissage peut se faire localement en minimisant J_i. Cependant cette fonction d'erreur prend en entrée les paramètres du modèle **w**. Dans un système distribué, la question de la cohérence des paramètres du modèle contenues sur les différents agents se pose. En effet, chaque agent doit modifier les paramètres du modèle afin d'apprendre des connaissances. Cette modification demande la lecture et l'écriture des paramètres pour tous les agents de manière simultanée. Plus le nombre d'agents est grand et plus cette incohérence peut être importante.

Performances Les réseaux de neurones profonds sont utiles car ils permettent d'obtenir des modèles très performants sur des tâches complexes. Les performances du modèle final vont dépendre de la manière dont l'apprentissage distribué se déroule. Les points qui se différencient de l'apprentissage centralisé sont la cohérence du modèle au cours de l'apprentissage ainsi que les données utilisées pour calculer les gradients à chaque itération. Pour évaluer une méthode d'apprentissage, il est donc important de calculer les performances finales du réseau de neurones.

Contraintes des communications Afin de synchroniser les agents, en particulier pour garder une cohérence sur les paramètres **w**, ceux-ci doivent communiquer entre eux. Cependant les communications entre machines sont contraintes, en particulier si elles sont connectées via Internet. Ce point met en évidence un compromis à faire entre communication et cohérence du modèle. La bande passante et la latence entre les machines sont donc des points à prendre en compte. En particulier, du fait que le nombre de paramètres d'un réseau de neurones profond est grand : les envoyer au travers le réseau peut s'avérer particulièrement coûteux. La coût de communication d'un algorithme d'apprentissage distribué est une mesure importante à évaluer.

Tolérance aux pannes Comme dans n'importe quel système distribué, les machines participantes peuvent tomber en panne. De nombreux travaux proposent des algorithmes distribués résistants à un certain nombre de pannes f. C'est-à-dire, des al-

gorithmes capables de rester fiables même avec f machines qui tombent en panne au cours de l'exécution. En plus des pannes, un autre type de fautes, appelées fautes byzantines [64], est envisagé par les systèmes distribués. Dans le cas de fautes byzantines, les processeurs atteints peuvent modifier en local leur algorithme comme bon leur semble. Ils peuvent envoyer des données erronées aux autres processeurs, choisir de suivre l'algorithme ou s'arrêter à tout moment. De même, différents algorithmes ont été proposés afin d'être capable de résister jusqu'à f fautes byzantines [64].

Dans notre contexte, les pannes sont à considérer car chaque participant peut à tout moment se déconnecter du système. Les raisons peuvent être une panne de la machine, une coupure de réseau, ou de manière volontaire, par exemple afin d'effectuer une tâche concurrente prioritaire. Il est donc important de s'intéresser à cette problématique de tolérance aux pannes présentes dans les systèmes distribués. Les fautes byzantines sont également intéressantes car elles peuvent représenter un utilisateur malveillant, c'est-à-dire un participant qui aurait pour but de compromettre l'apprentissage du modèle. Nous discutons de ce point dans la Section 5.5.

2.3 Apprentissage profond distribué

Cette section a pour but de donner un aperçu des méthodes existantes pour distribuer l'apprentissage des réseaux de neurones sur plusieurs machines ou GPUs. Ces techniques ont été principalement proposées pour accélérer l'apprentissage ou réduire la contrainte en terme de mémoire sur les différentes machines au sein même d'un *datacenter*. Elles n'abordent donc pas toutes les problématiques liées aux systèmes distribués de manière générale (temps de communications non bornés, large échelle, tolérance aux pannes, ...). Nous n'aborderons pas le cas des architectures matérielles pour accélérer l'apprentissage sur un seul processeur tel que les cartes graphiques, les cartes FPGA ou les *Tensor Processor Unit* de Google. Comme expliqué dans la Section 2.2, ces architectures n'ont pas les mêmes problématiques que les systèmes distribués.

2.3.1 Différents types de parallélisme

Pour paralléliser l'apprentissage du réseau de neurones, deux types de méthodes ont été proposées : le parallélisme du modèle et le parallélisme des données. Il n'est

pas rare de voir des architectures employant ces deux types de parallélismes dans les travaux de recherche comme [28, 22] que nous verrons dans la Section 2.3.2.

Parallélisme du modèle

Le parallélisme du modèle consiste à diviser la charge de calcul pour l'inférence et la rétropropagation du réseau de neurones. Il existe plusieurs granularités possibles pour partager le réseau de neurones sur différentes machines :

— Il est possible de répartir les différentes couches sur les agents disponibles. Cette solution permet notamment de pipeliner l'exécution du réseau de neurones lorsque plusieurs données sont calculées les unes après les autres (comme c'est le cas dans la descente de gradient avec mini-batch).

— Avec un niveau de granularité plus fin, il est possible de diviser les neurones présents sur chaque couche entre les agents. Dans le cas des CNN, les neurones peuvent être répartis selon plusieurs dimensions : la hauteur, la largeur ou les canaux (voir Figure 2.1).

— De manière encore plus fine, il est possible de partager chaque opération matricielle du réseau de neurones de manière indépendante. Une couche de neurones peut nécessiter plusieurs opérations différentes ce qui permet de répartir encore plus finement la charge sur chaque agent.

Il existe différents travaux [111] qui se sont intéressés à la recherche de la répartition optimale du modèle sur un ensemble d'agents. Le parallélisme du modèle a pour avantage d'accélérer la vitesse d'inférence et de rétropropagation du réseau de neurones profond mais également de répartir la charge de mémoire nécessaire pour contenir toutes ces opérations. Cette répartition de la mémoire est particulièrement utile dans le cas où celle-ci est limitée pour les différents agents (comme dans le cas des GPUs).

Le parallélisme du modèle nécessite beaucoup de communications entre les agents. Même si certaines architectures de réseaux de neurones ont parfois été prévues pour limiter ces communications, il est généralement nécessaire d'avoir l'activation des neurones d'une couche pour calculer la suivante. De plus, les données, qu'elles soient sous forme brute ou sous forme d'activation de neurones à l'intérieur des couches, sont accessibles en partie à plusieurs machines (ou GPUs). Ceci ne permet pas de répondre aux problématiques de préservation de la vie privée évoquées en Section 2.2.

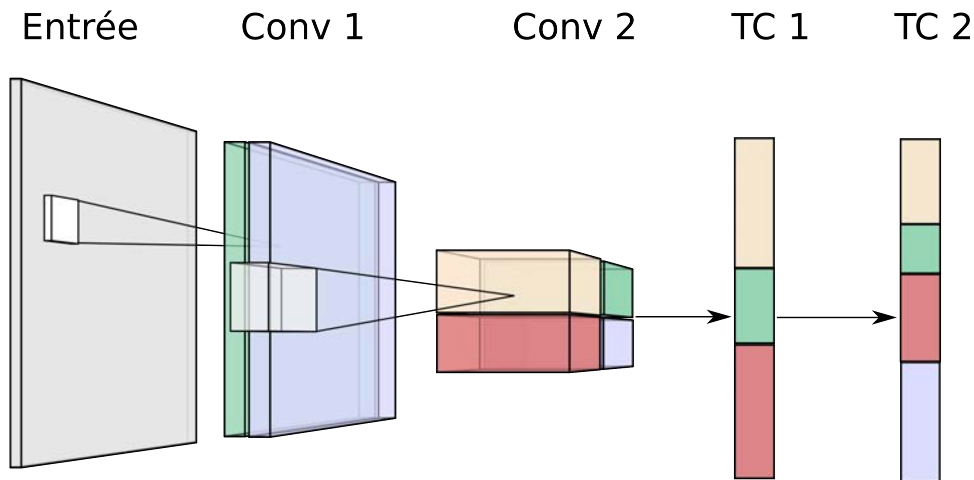


FIGURE 2.1 – Exemple d'un CNN parallélisé sur quatre agents. Dans ce type de granularité, le calcul de l'activation de chacun des neurones est distribué entre les agents. Chaque couleur représente l'attribution à un agent différent. Les premières couches, appelées Conv 1 et Conv 2 sont des couches convolutives tandis que TC1 et TC2 sont des couches toutes connectées.

Parallélisme des données

La descente de gradient nécessite le calcul d'un gradient à l'aide de multiple données. Soit une fonction de coût J en fonction d'un vecteur de paramètres **w** de la forme :

$$J(\mathbf{w}) = \sum_k L_k(\mathbf{w})$$

avec L_k une observation sur la k-ième donnée ou une observation sur le k-ième batch de données X_k. Il est possible de paralléliser la minimisation de $J(\mathbf{w})$ en calculant les gradients $\Delta\mathbf{w}^{(k)} = \partial L_k(\mathbf{w})/\partial\mathbf{w}$ sur des agents différents. Chaque agent doit donc avoir accès à l'ensemble du réseau de neurones profond afin de calculer les gradients. Ce type de parallélisme permet cependant de partager le temps de calcul de la descente de gradient total.

Descente de gradient parallèle synchrone La descente de gradient par mini-batch est un calcul d'une succession de gradients $\Delta\mathbf{w}_0, \Delta\mathbf{w}_1, \ldots, \Delta\mathbf{w}_I$ (avec I le nombre d'itérations) appliqués aux paramètres du modèle au cours du temps. Chacune de ces modifications demande de calculer le gradient pour b exemples tirés de la base de

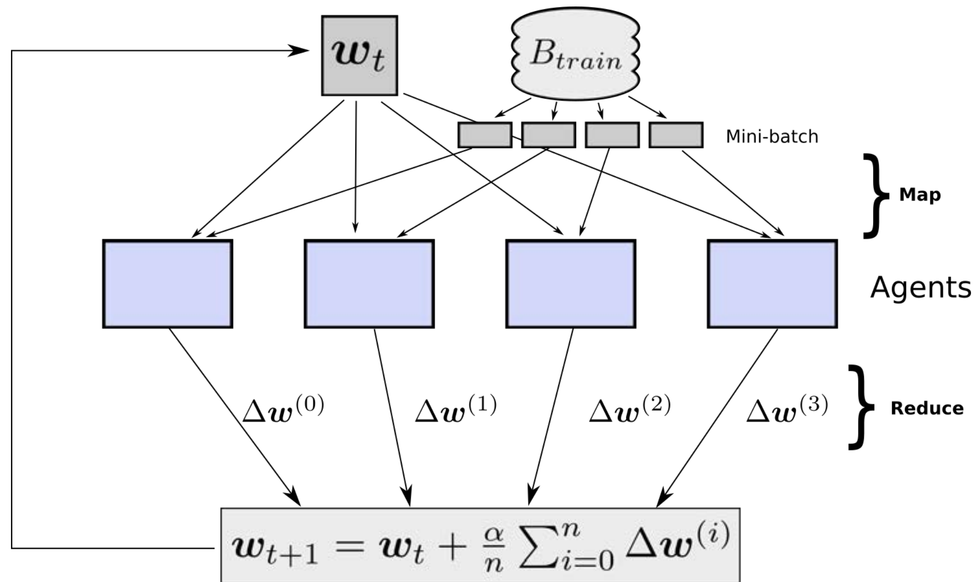


FIGURE 2.2 – Descente de gradient synchrone entre 4 agents. L'exécution d'une itération peut être vu comme une opération de MapReduce.

données d'apprentissage.

Une première solution consiste à utiliser une méthode de type MapReduce [27] pour calculer ce gradient. L'idée est de répartir des données sur chacun des agents comme montré sur la Figure 2.2. À l'itération t, chaque agent calcule un gradient $\Delta\mathbf{w}_t^{(i)}$ sur un batch $X_t^{(i)}$ de taille b qu'il reçoit. Les gradients sont agrégés en faisant la moyenne de ceux-ci :

$$\Delta\mathbf{w}_t = \frac{1}{n}\sum_{i=1}^{n}\Delta\mathbf{w}_t^{(i)}$$

Ce gradient correspond exactement au gradient qui aurait été calculé sur un batch contenant toutes les données de $X_1, \ldots, X_n$, de taille $b \times n$. La descente de gradient synchrone contraint donc à faire des modifications sur des batchs plus grands lorsque n est grand. Cependant, à chaque itération les agents ont les mêmes paramètres $\mathbf{w}_t$. Cette méthode n'induit donc pas d'incohérence des paramètres.

Descente de gradient asynchrone La descente de gradient parallèle synchrone a l'avantage de répartir simplement la charge de calcul sur n agents. Cependant, tous doivent attendre que l'opération d'agrégation soit finie avant de commencer une nouvelle itération. De nombreux travaux [4, 118, 83, 89, 120] font le choix d'une descente

de gradient dite asynchrone. C'est-à-dire que chaque agent peut mettre à jour par lui-même les paramètres du modèle. Dans ce cas les paramètres sont modifiés de la manière suivante dès qu'un agent i envoie son gradient :

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \Delta \mathbf{w}_{t'}^{(i)}.$$

Il est important de noter que $\Delta \mathbf{w}_{t'}^{(i)}$ a été calculé à un temps $t' \leq t$. En effet, les agents modifient les paramètres de manière asynchrone. Lors de la mise-à-jour des paramètres, les dernières modifications ne sont pas accessibles à tous les agents. Un agent calcule donc un gradient pour une ancienne valeur des paramètres $\mathbf{w}_{t'}$. Le gradient calculé $\Delta \mathbf{w}_{t'}^{(i)}$ est dit gradient avec un délai $r = t' - t$.

Ce phénomène est typiquement dû à l'incohérence des paramètres pour l'ensemble des n agents. Elle permet cependant la lecture et l'écriture asynchrones sur $\mathbf{w}_t$ par tous les agents. Le délai des gradients peut engendrer des problèmes en terme d'apprentissage. En effet, lorsque celui-ci est trop important ou mal géré (par exemple, avec un taux d'apprentissage α trop grand), la recherche des paramètres optimaux peut diverger ou être fortement ralentie. Cependant le gain en terme de vitesse de calcul est intéressant, en particulier lorsque le nombre d'agents est grand ou que leur vitesse de calcul n'est pas homogène. De plus, ces gradients avec délai ont un effet proche de celui de la méthode du momentum utilisée dans le cas centralisé. Ils leur est également attribué le rôle d'un régularisateur (à la manière du *drop-out*).

Gestion de l'incohérence des paramètres Comme nous l'avons vu, la descente de gradient asynchrone mène aux calculs de gradients avec délais. Plus le délai d'un gradient est grand et moins la modification qu'il va apporter a un sens pour la minimisation de la fonction d'erreur. Pour éviter ce phénomène, certains travaux, comme W. Zhang *et al.* [118] proposent une méthode semi-synchrone, dans laquelle le système attend une fraction n/p des gradients calculés par les agents avant de modifier les paramètres. Cela a pour effet de réduire le délai moyen des gradients. J. Chen *et al.* [20] proposent d'utiliser la méthode de descente de gradient synchrone (sans délai pour les gradients) mais de réduire le temps d'attente des agents les plus rapides en ignorant les gradients des k agents les plus lents à chaque synchronisation (ils ne sont donc pas attendus). Une autre solution proposée par W. Zhang *et al.* [118], mais également dans les travaux de Odena [81], consiste à adapter le taux d'apprentissage α directement aux délais r des gradients ou à la variance des gradients calculés durant

les r dernières itérations.

Synchronisation par modèle moyen et méthodes de moyennes élastiques

Les travaux de H. Brendan *et al.* [76] montrent qu'à condition que deux modèles soient initialisés avec les mêmes paramètres, il est possible d'utiliser la moyenne de leurs paramètres après quelques itérations de descente de gradient pour obtenir un nouveau modèle plus performant. Partant de ce principe, cette méthode permet d'entraîner de multiples réseaux de neurones (avec une même initialisation) sur des données différentes pendant plusieurs époques et d'agréger ces modèles par une moyenne de leurs paramètres afin de partager les connaissances acquises. Plus formellement, soit n agents avec chacun un réseau de neurones profond dont les paramètres $\mathbf{w}^{(i)}$ sont initialisés à $\mathbf{w}_0$. Tous les agents effectuent une descente de gradient durant E époques avec des données différentes (c'est-à-dire, qu'ils calculent un gradient en utilisant chacune de leurs données E fois). Une fois que tous les agents ont fini, ils mettent en commun leur modèle en calculant :

$$\mathbf{w}_t = \frac{1}{n}\sum_{i=0}^{n}\mathbf{w}^{(i)}$$

Les agents prennent alors comme nouveaux paramètres $\theta^{(i)} = \theta_t$. Cette technique de *Federated Learning* est actuellement celle utilisée pour entraîner des réseaux de neurones sur des smartphones Android.

Les travaux de Zhang *et al.* [117] proposent quant à eux une méthode, nommée *Elastic Averaging SGD* (EASGD), semblable à la méthode du *Federated Learning*, mais de façon asynchrone avec une moyenne dite "élastique". Lorsqu'un agent i, avec des nouveaux paramètres $\theta^{(i)}$ partage son apprentissage aux paramètres courants du modèle θ_t, il modifie θ_t de la manière suivante :

$$\theta_{t+1} = (1-\alpha)\theta_t + \alpha\theta^{(i)}$$

avec $\alpha \in]0,1]$ le coefficient d'influence. Ses paramètres en local sont également modifiés par cette même méthode :

$$\theta^{(i)} = (1-\beta)\theta_{t+1} + \beta\theta^{(i)}$$

avec $\beta \in]0, 1]$. Ces deux méthodes supposent que les agents ont une mémoire locale. Dans le cas du *Federated Learning*, tous les agents évoluent de manière différente durant plusieurs itérations puis se "synchronisent" lors de l'étape de mise en commun. La méthode de EASGD, quant à elle, permet aux agents de "graviter" autour des paramètres du réseau de neurones pour faire une meilleure exploration de l'espace des paramètres.

Apprentissage d'ensembles parallèles Les ensembles de modèles sont quelques peu particulières car ils n'ont pas pour but de paralléliser l'apprentissage d'un modèle en soit, mais créent un méta-modèle, composé de plusieurs modèles (généralement plus simples) appris de manière différente. Ce méta-modèle, que nous appelons ensemble, combine les sorties de ses modèles de différentes façons afin de réduire la variance de ses prédictions. Cette réduction de variance permet de réduire considérablement les erreurs de prédiction sur des exemples non vus durant l'apprentissage (erreur de généralisation).

Les méthodes d'ensemble ont été utilisées sur de nombreux types de modèle d'apprentissage. Par exemple, ils ont également utilisé avec une architecture de cascade de classifieurs en image [108]. De même, le modèle de forêt d'arbres aléatoires est un modèle d'ensembles car il utilise la moyenne des réponses apportées par de multiple arbres aléatoires [17]. Il est également possible de trouver des exemples d'ensemble de réseaux de neurones profonds, en particulier pour l'apprentissage de GAN [49, 112, 104].

Bien que le but premier des ensembles est d'obtenir un modèle plus performant à partir d'une multitudes de modèles plus simples, l'avantage évident de ces méthodes est de généralement pouvoir apprendre chacun des modèles de manière totalement indépendante.

2.3.2 Les architectures proposées

De nombreuses architectures ont été proposées afin de mettre en place ces différents types de parallélismes sur une ou plusieurs machines. Nous présentons ici plusieurs grands groupes de méthodes ayant été développées.

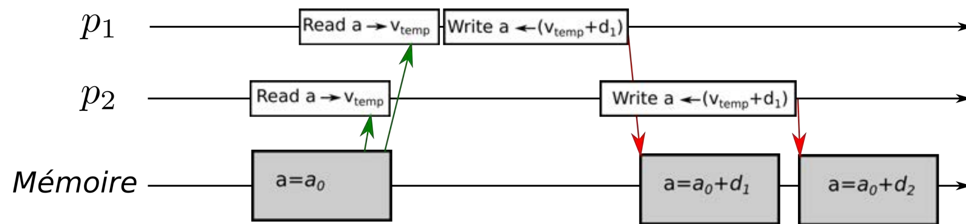


FIGURE 2.3 – Exemple de problème de sur-écriture d'une variable classique lors de l'utilisation d'une mémoire partagée sans synchronisation des processeurs. Lorsque le processeur p_2 modifie la variable a dans la mémoire, il ne prend pas en compte les modifications apportée par p_1 de manière concurrente.

Architecture avec une mémoire partagée

Une version de descente de gradient asynchrone a été proposée dans les travaux de [89]. Dans ce modèle, plusieurs agents (des processeurs) ont en commun une mémoire. Les auteurs ont montré que chaque processeur peut faire un calcul de gradients sur un mini-batch différent en parallèle et appliquer la mise à jour des paramètres sans synchronisation (c'est-à-dire, sans blocage au niveau de la lecture et de l'écriture des paramètres). Cette absence de blocage a pour effet de permettre l'écrasement de variables lors d'une écriture concurrente sur la mémoire partagée (voir une illustration sur la Figure 2.3). Cependant ce phénomène a un impact relativement faible sur l'apprentissage du fait que les modifications faites à chaque itération sont généralement parcimonieuses, c'est-à-dire, que le gradient calculé est important pour quelques paramètres du modèle et très faible pour tous les autres lors du calcul d'un gradient sur un minibatch. Les paramètres concernés par une modification dépendent beaucoup des données contenues dans le batch X_t de l'itération t. Cette parcimonie réduit considérablement les problèmes dus à l'écrasement d'écriture lors de la mise-à-jour des paramètres. Cette architecture est décrite dans la Figure 2.4. Bien qu'elle ne permette pas de répondre à des problématiques de système distribué, cette architecture montre la tolérance de l'apprentissage des réseaux de neurones profonds face à des opérations concurrentes asynchrones.

Architecture avec un serveur central

À une plus large échelle, il est possible d'utiliser un ou plusieurs serveurs centraux afin de prendre le rôle de la mémoire partagée. Les agents sont alors situés sur des

machines avec leur propre mémoire et une portion de la base de données d'apprentissage. Ces serveurs sont appelés serveurs de paramètres car ils gardent en mémoire l'ensemble des paramètres du réseau de neurones profond et gèrent les modifications apportées par les agents. Ce modèle a été proposé pour la première fois par J. Dean *et al.* [28] avec une architecture appelée DistBelief afin de réaliser une descente de gradient asynchrone sur un grand nombre de machines. Elle est couplée à une méthode de parallélisation du modèle afin de réduire la charge de travail de chaque machine (agents et serveurs de paramètres). Elle est présentée dans la Figure 2.4 b).

Cette architecture a également été reprise et améliorée par [22], qui propose notamment de rajouter un serveur de données pour distribuer les batches de manière efficace aux agents. De plus les auteurs apportent de nombreuses optimisations en terme d'architecture, de gestion de la mémoire et des communications qui leur permettent des meilleures performances que l'architecture DisBelief avec 50 fois moins de machines.

Enfin S. Gupta *et al.* [41] proposent différentes architectures avec une communication basée sur un réseau en forme d'arbre. L'ensemble des serveurs de paramètres forment un arbre binaire. Les noeuds de cet arbre se chargent de récupérer les gradients de deux agents (lors d'une descente de gradient synchrone, semi-synchrone ou asynchrone). Les gradients sont alors agrégés de bas en haut jusqu'à atteindre la racine de l'arbre. Une fois ces gradients agrégés à la racine, la modification peut être appliquée sur les paramètres du réseau de neurones. Dans une première version de l'architecture, les nouveaux paramètres sont renvoyés de haut en bas pour atteindre tous les agents. Dans une variante, les auteurs proposent d'envoyer les paramètres mis à jour directement à la racine d'un nouveau réseau en arbres composé uniquement d'agents.

Architecture décentralisée

La méthode du serveur de paramètres repose sur des serveur centraux contenant les paramètres. Il est important de bien les configurer afin de ne pas avoir des problèmes de goulot d'étranglement dus à la réception et aux traitement des modifications envoyées par les agents. De plus, en cas de panne d'un des serveurs, tout l'apprentissage se retrouve compromis. Pour cette raison des méthodes ont été proposées afin de se passer de ce noeud central. L'idée consiste à mettre en commun l'apprentissage avec des méthodes de réduction ou de consensus pour chacun des agents.

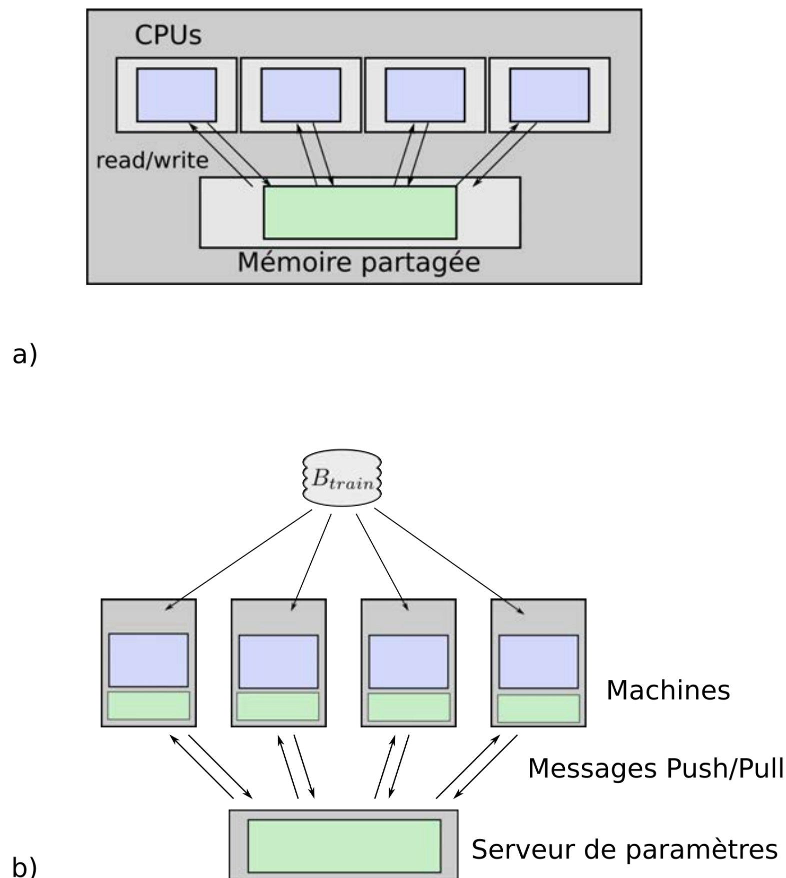


FIGURE 2.4 – a) Modèle avec une mémoire centrale et b) avec un serveur de paramètres. Les rectangles bleus représentent les agents tandis que les rectangles verts représentent les paramètres du modèle stockés en mémoire.

Réduction par anneau La technique de réduction par anneau a d'abord été introduite pour faire la moyenne sur des grosses bases de données dans un contexte de *datacenters* [86]. Elle permet en effet de faire la moyenne d'un vecteur de valeurs contenu sur différentes machines avec un coût total de communication qui ne dépend pas du nombre de machines. La méthode est décrite sur la Figure 2.5. Elle a été utilisée principalement pour mettre en commun les gradients calculés sur de nombreuses GPUs au sein d'un même serveur.

Cette solution requiert $2 \times (n-1)$ communications pour effectuer une mise en commun d'un gradient mais chacune d'entre elles est de taille $|\theta|/n$. Elle est donc particulièrement intéressante lorsque la bande passante est limitante mais que la latence des communications reste faible et que la vitesse des différents agents est relativement homogène. Elle convient donc bien à un environnement comme un *datacenter* mais

elle est plus difficilement applicable dans un système distribué asynchrone avec des pannes potentielles.

Cette architecture a été proposée par Baidu [3], puis reprise par les ingénieurs de Uber avec la méthode Horovord [96]. Cette dernière version propose de nombreuses améliorations en terme de communications afin d'accélérer l'apprentissage. Elle est actuellement disponible sur Tensorflow.

Protocole de rumeur Les protocoles de rumeur ont pour but de diffuser une information (comme une valeur) dans un environnement de systèmes distribués. Elles sont particulièrement intéressantes car efficaces et robustes dans des environnements asynchrones avec des pannes [16]. Il est possible d'utiliser ces méthodes afin de partager les paramètres des réseaux de neurones profonds [13] ou les gradients calculés par les différents participants au cours de l'apprentissage [26].

Dans les travaux de M. Blot *et al.* [13], chaque agent envoie ses paramètres à un de ses pairs avec une certaine probabilité à chaque itération. Au début de chaque itération, il fait une moyenne pondérée entre son modèle et chaque modèle qu'il a reçu de ses pairs durant l'itération précédente. Cette méthode est donc basée sur la technique de modèle moyen présentée en Section 2.3.1 et permet de tirer profit des avantages des protocoles de rumeur. Nos travaux dans le Chapitre 4 reprennent cette méthode ainsi qu'une variante pour les évaluer sur l'apprentissage de GAN.

Dans les travaux de J. Daily *et al.* [26], chaque agent envoie ses gradients calculés, ainsi que ceux reçus, à un voisin choisi en suivant un schéma de communication particulier. Ce schéma permet de faire en sorte que chaque agent ait reçu les n gradients d'une itération en $O(\log(n))$ communications. Ils proposent, pour plus d'efficacité, d'effectuer ces communications de manière totalement asynchrone (c'est-à-dire, une descente de gradient asynchrone).

2.4 Conclusion et méthodes existantes adaptées aux systèmes distribués

Durant la période couvrant mes travaux de thèse, des méthodes ont été proposées par différentes équipes de recherche pour résoudre ce problème d'apprentissage sur

3. Voir http://andrew.gibiansky.com/

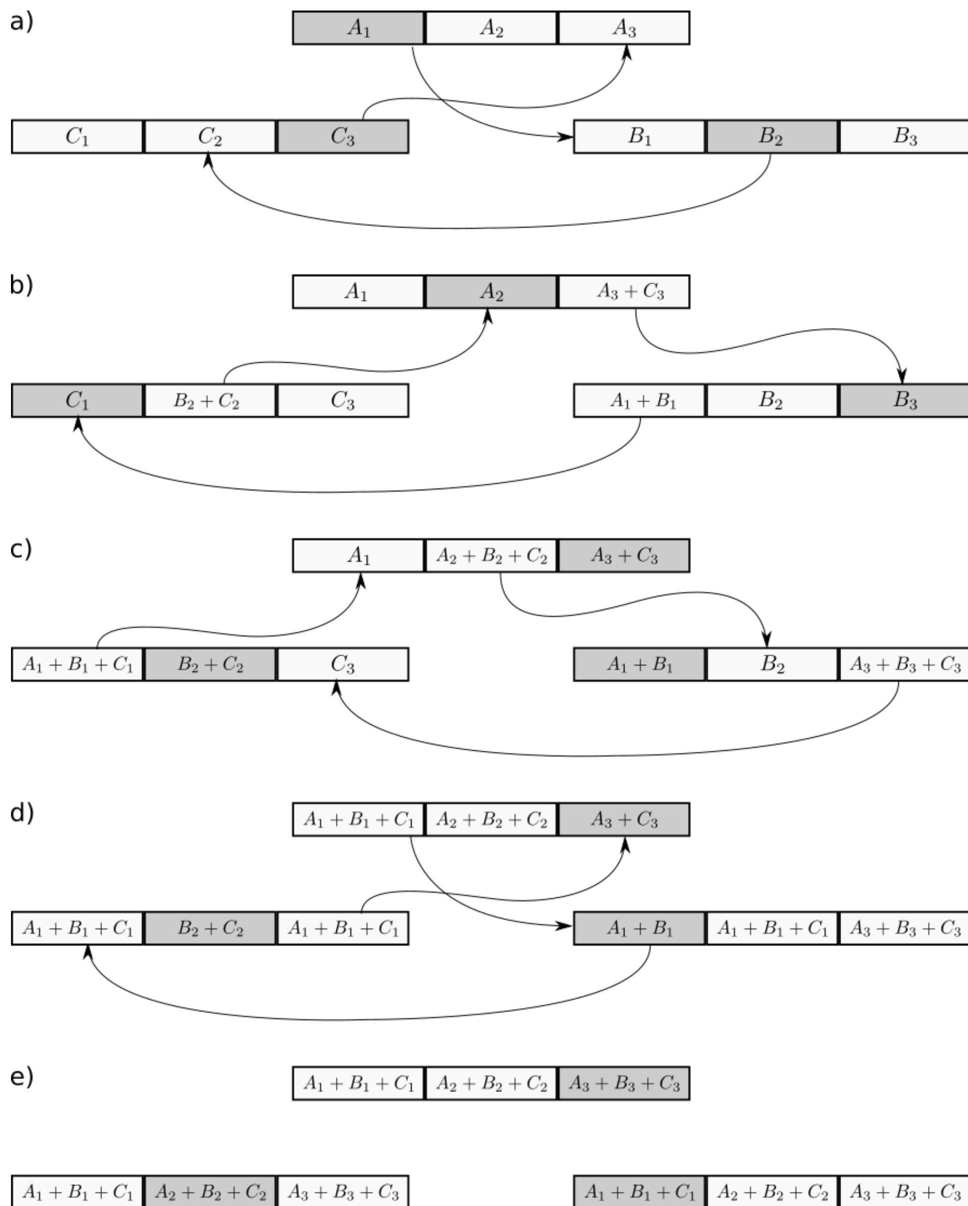


FIGURE 2.5 – La mise en commun d'un vecteur via une méthode de réduction par anneau (*ring-reduce*). Le vecteur est découpé en trois sur chaque agent (c'est-à-dire, A_1, A_2, A_3 sur l'agent a). Les étapes de communications sont illustrées de a) à e). Chaque agent contient le vecteur $A + B + C$ au bout de 4 communications.

des machines périphériques. La méthode la plus notable à laquelle nous nous comparons dans le chapitre 4 et le chapitre 5 est la méthode dite *Federated Learning* [76, 61, 60] proposée par les ingénieurs de Google. Elle est actuellement testée dans l'application Gboard d'Android afin d'apprendre à faire des propositions de requêtes à l'utilisateur. Comme expliqué dans la Section 2.3.1, le *Federated Learning* utilise la mise en commun par calcul d'un modèle moyen après de multiples itérations d'apprentissage effectuées en parallèle sur de machines. Elle utilise l'architecture du serveur de paramètres afin de calculer et stocker les paramètres moyens du réseau de neurones. L'avantage de cette méthode est de réduire les contraintes en terme de communication pour l'apprentissage. En effet, contrairement à une descente de gradient synchrone ou asynchrone, les communications entre les agents et le serveur central sont beaucoup moins nombreuses. De plus, elle est capable de supporter un grand nombre de machines participantes, elle est tolérante aux pannes et de nombreuses améliorations ony été proposées [61, 60].

Une autre méthode [103] a également été proposée pour apprendre un modèle avec une parallélisation du modèle entre les capteurs, les routeurs et le serveur placé sur le cloud. Le modèle est distribué de la manière suivante : les premières couches du réseau de neurones sont directement placées sur les capteurs, des couches intermédiaires sont placées sur des routeurs chacun reliés à plusieurs capteurs et les couches finales sont placées sur le cloud. Sur chaque partie (capteur, routeur, cloud), des sorties intermédiaires sont placées à la manière du réseau de neurones Inception [102]. Le but de cette méthode est d'utiliser les couches supérieures du réseau uniquement lorsque les couches basses n'ont pas une confiance suffisante en leur résultat. L'apparition de ces sujets de recherche montrent l'importance que va avoir l'apprentissage profond effectué sur des systèmes distribués.

Dans les chapitres suivants nous proposons plusieurs contributions liées à l'apprentissage profond sur des systèmes distribués. Dans chacune des contributions, nous avons pris en compte les contraintes suivantes :

- les capacités de mise-à-l'échelle des différentes méthodes,
- les contraintes imposées sur le réseau du système distribué,
- la tolérance aux pannes au cours de l'apprentissage,
- la nécessité de garder les données des utilisateurs uniquement sur leurs machines.

Le dernier point est nécessaire, mais non suffisant, pour garantir le respect de la vie

privée des utilisateurs participant à l'apprentissage. Nous discuterons en détails de ce point dans les perspectives de la thèse (Section 5.5).

CHAPITRE 3

DESCENTE DE GRADIENT ASYNCHRONE AVEC *AdaComp*

Dans ce chapitre, nous allons présenter une première contribution de cette thèse afin de permettre l'apprentissage profond sur des systèmes distribués. Cette méthode, que nous appelons *AdaComp* (pour ***Ada**ptive learning rate for **Comp**ressed updates in Asynchonous Stochastic Gradient Descent*), permet d'effectuer un apprentissage profond sur un grand nombre de machines à l'aide d'un serveur de paramètres tout en réduisant le trafic entre les machines et le serveur. Dans nos expérimentations, nous montrons que *AdaComp* obtient de très bons résultats en terme d'apprentissage avec un nombre d'agents allant jusqu'à 200, tout en réduisant le trafic de deux ordres de grandeur comparé à une descente de gradient asynchrone classique faite avec le serveur de paramètres.

3.1 Modèle du serveur de paramètres dans le cas des systèmes distribués

Nous considérons le modèle du serveur de paramètres (voir Section 2.3.2) dans lequel les agents ont été placés sur des machines mises à disposition par les participants. Le serveur de paramètres reste, quant à lui, dans un serveur central. La Figure 3.1 représente cette configuration avec différents types de machines utilisées comme agents.

Dans ce contexte, nous supposons que les données d'entraînement sont fournies par les utilisateurs. Les données sont mises à disposition sur la machine, ou peuvent être produites directement au cours de l'apprentissage (comme dans le cas de l'analyse d'un trafic réseau, d'une vidéo surveillance ou de l'entrée d'un micro). Dans ce scénario, les données ne quittent pas la machine de l'utilisateur, ce qui peut servir de

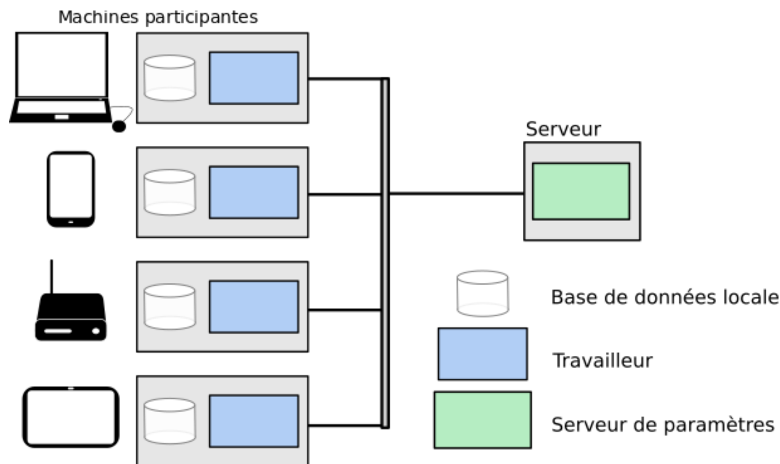


FIGURE 3.1 – Exemple du modèle de paramètres configuré sur des machines utilisateurs .

base à un scénario de préservation de la vie privée (tout comme dans les travaux de Shokri et al. [98]). Nous prendrons pour exemple un cas dans lequel les utilisateurs veulent contribuer à l'apprentissage d'un classifieur de photo(s) (à la manière d'ImageNet [92]). Les données d'entraînement fournies par les utilisateurs peuvent être leurs photos personnelles. Pour des raisons de respect de la vie privée, il est alors préférable de garder ces photos uniquement sur la machine de leur propriétaire et donc de ne pas les envoyer vers un service *cloud*.

Le modèle du serveur de paramètres se déroule de la même manière que dans le cas d'un *datacenter*. Nous prenons les hypothèses suivantes :

— Les agents ont assez de ressources pour faire tourner le réseau de neurones et mettre en mémoire les paramètres θ. Cette contrainte est importante mais de nombreux progrès ont été faits pour faire tourner des réseaux de neurones sur des téléphones mobiles dans le cadre de l'inférence. De plus, les capacités actuelles des machines utilisateurs sont généralement suffisantes pour faire fonctionner des réseaux de neurones profonds de taille raisonnable.

— Les utilisateurs participants à l'apprentissage sont honnêtes et n'envoient pas de fausses informations. Le cas de la tolérance aux fautes byzantines est abordé dans des travaux d'apprentissage distribué [12]. Il est imaginable d'adapter ces techniques dans notre contexte afin de permettre une tolérance à ce genre de fautes (voir Section 5.5).

— La base de données d'apprentissage (donc l'ensemble des données fournies

par les utilisateurs) est distribuée de manière uniforme sur l'ensemble des utilisateurs. Nous supposons donc que chaque utilisateur a le même nombre de données et que la distribution de ces données est identique. En pratique, il est raisonnable de supposer un même ordre de grandeur pour le nombre de données fournies par les utilisateurs. La distribution des données dépend généralement du contexte.

- La base de données totale est suffisante pour apprendre la tâche finale. C'est-à-dire que si nous supposons l'ensemble de la base distribuée $B_{train} = \bigcup_{i=1}^{N} B_i$ centralisée dans le *datacenter*, il serait possible d'apprendre le modèle avec une erreur suffisamment faible pour répondre à la tâche visée.
- Nous supposons que les agents sont reliés par une connexion ADSL/câble asynchrone avec le serveur (c'est-à-dire, 100Mo/10Mo montants/descendants).

Du fait de notre contexte de systèmes distribués, nous avons choisi d'étudier particulièrement le cas de la descente de gradient asynchrone avec le modèle du serveur de paramètres. La raison à cela est la forte disparité possible en terme de rapidité d'exécution des agents sur les machines des utilisateurs, comme expliqué dans la Section 2.1. De plus, la descente de gradient asynchrone a l'avantage d'être particulièrement résistante aux pannes et facilite l'arrivée de nouveaux agents à tout moment au cours de l'apprentissage. En effet, chaque agent peut se connecter ou déconnecter à tout moment du serveur de paramètres. À chaque connexion, il doit cependant reprendre l'apprentissage en récupérant les paramètres sur le serveur de paramètres. Nous allons voir dans cette section, quelles sont les problématiques liées à cette méthode dans le contexte de systèmes distribués.

3.1.1 Gestion des gradients avec délai

La descente de gradient asynchrone implique une incohérence dans l'état des paramètres comme nous l'avons vu dans la Section 2.3.1. Cette incohérence mène à des gradients avec délai. Dans le cas du modèle du serveur de paramètres, ce délai correspond à la différence du nombre de mises à jour entre le modèle du serveur de paramètres et le modèle local de l'agent concerné (celui qui a calculé le gradient). En effet, une fois qu'un agent a récupéré les paramètres sur le serveur central, ceux-ci peuvent être modifiés par d'autres agents à tout moment. Nous appelons le *retard d'un agent* i, le nombre d'itérations effectuées sur le serveur de paramètres depuis le

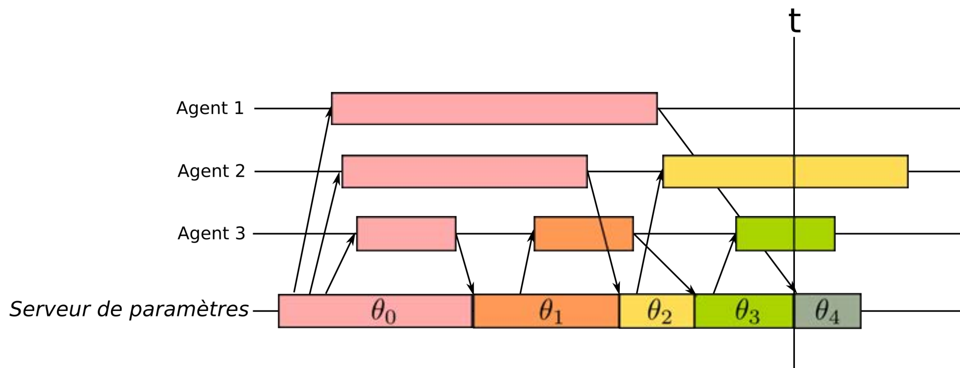


FIGURE 3.2 – Exemple de configuration possible dans une descente de gradient asynchrone. Les différentes mises à jour des paramètres θ sont représentées par des couleurs différentes. À l'instant t, l'agent 1 a un retard de 3, l'agent 2 a un retard de 1 et l'agent 3 a un retard de 0. Le gradient envoyé par l'agent 1 a donc un délai de 3.

moment où l'agent a mis à jour ses paramètres. Le *délai d'un gradient* $\Delta\theta^{(i)}$ correspond au retard de l'agent i lorsque la modification associée est appliquée au serveur de paramètres. Ces deux concepts sont illustrés dans la Figure 3.2.

Comme nous l'avons vu dans la Section 2.3.1, ce problème d'incohérence de l'état du modèle lié à ce phénomène de délai, peut être problématique lorsque le nombre d'agents est important.

D'après S. Gupta *et al.* [41], le délai moyen est proportionnel au nombre d'agents dans le cas d'une descente de gradient asynchrone classique. Cependant, dans le cas où les agents ont des vitesses d'exécution très variées, ce délai peut prendre des valeurs très grandes pour les agents les plus lents.

Dans notre contexte d'apprentissage sur des systèmes distribués (Section 2.1), le nombre d'agents peut être particulièrement important. De plus, ceux-ci ne peuvent pas être considérés comme homogènes en terme de temps d'exécution. Le délai des gradients peut donc devenir particulièrement grand et donc poser problème à l'apprentissage. La Section 3.2 détaille quelle solution *AdaComp* utilise afin de prendre en compte ce problème de modification avec délai.

3.1.2 Réduire le trafic entrant au niveau du serveur

Le contexte d'apprentissage de réseau de neurones collaboratif avec de nombreux participants entraîne des difficultés techniques. L'une d'entre elles est clairement la bande-passante nécessaire à l'apprentissage entre les agents et le serveur de pa-

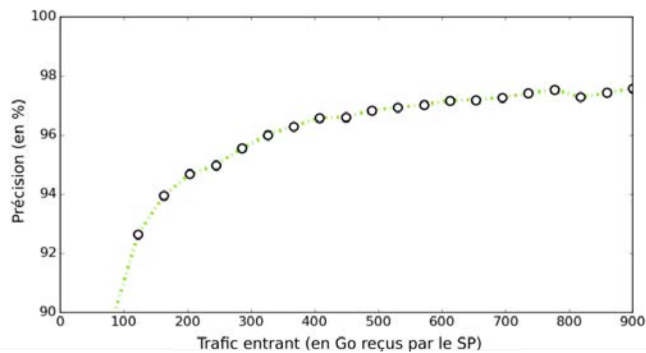


FIGURE 3.3 – Précision (en %) vs trafic entrant agrégé (en Go). Mesure du trafic entrant pour une descente de gradient asynchrone avec 200 agents avec un modèle de CNN sur la base de données MNIST (voir Section3.3).

ramètres. Plus précisément, le trafic qui provient des agents et arrive sur le serveur central. Il y a deux raisons à cela :

— Le dimensionnement des serveurs dans le *cloud* est souvent directement lié aux capacités de réceptions et de traitements souhaitées (voir par exemple, Amazon Kinesis, dont le prix dépend de la quantité de données à réceptionner par seconde). Afin de proposer une solution peu coûteuse, il est donc nécessaire de réduire au maximum le trafic entrant du serveur de paramètres.

— La tâche d'apprentissage doit rester une tâche de fond pour les machines participantes. Leur débit sortant étant généralement limité, il n'est pas raisonnable de le saturer en permanence afin d'envoyer des informations au serveur de paramètres.

Il est donc important de mesurer le trafic entrant au niveau du serveur de paramètres pour avoir un point de vue global du trafic montant généré par l'apprentissage sur les agents. Nous avons illustré cette contrainte imposée par le modèle, avec une expérience de 200 agents et un serveur de paramètres qui ont pour but d'apprendre un classifieur simple sur la base de données MNIST [67]. Le trafic entrant sur le serveur de paramètres est reporté sur la Figure 3.3. Dans cette expérience, la quantitié de donnée nécessaire pour apprendre le modèle à un niveau de précision supérieur à 96% (c'est-à-dire, plus de 96% des données de l'ensemble de test sont bien classifiées) est de près de 400Go. Ce trafic entrant est beaucoup trop important comparé à la taille de la base de données d'apprentissage MNIST (environ quelques centaines de Mo).

Compression Shokri *et al.* [98] proposent une mécanique de compression qui permet de réduire la taille des gradients envoyés par les agents vers le serveur de paramètres, appelée la descente de gradients sélective. Une fois le gradient calculé par un agent sur un batch de données, le principe consiste à sélectionner seulement une partie des paramètres sur laquelle le gradient sera appliqué. La sélection peut se faire de manière aléatoire ou en choisissant les valeurs du gradient les plus importantes. Le taux de compression ainsi gagné est représenté par une valeur fixe $c \in (0, 1]$. Leurs expérimentations montrent que la précision du modèle appris en utilisant la sélection de gradients reste très proche du modèle appris par une descente de gradient classique. Dans leurs exemples, un réseau de neurones profond voit sa précision finale passer de $98,10\%$ à $97,07\%$ sur le dataset MNIST avec une sélection $c = 0.01$.

Le trafic total entrant au niveau du serveur de paramètres peut être calculé analytiquement. Dans le cas du gradient sélectif, le trafic entrant du serveur de paramètres est de l'ordre de $I \times |\theta| \times c$ par agent (avec I le nombre total d'itérations). Le point important à mesurer est la précision du modèle au cours de l'apprentissage. Nous avons vu que cette précision n'était pas linéaire par rapport au nombre d'itérations, par conséquent, elle n'est pas linéaire par rapport au trafic entrant (voir Figure 3.3). C'est pourquoi il est important de mesurer la précision par rapport au trafic expérimentalement. Comme nous allons le voir dans la Section 3.3, l'approche de Shokri *et al.* [98] dans notre contexte permet de réduire la taille des modifications envoyées par les agents au coût d'une perte en précision du modèle. Cette dégradation du modèle est problématique car les réseaux de neurones profonds ont la capacité de pouvoir être très précis. Afin de réduire la taille des gradients envoyés par les agents, tout en maintenant une bonne précision lors de l'apprentissage, nous introduisons l'algorithme d'*AdaComp* dans la section suivante.

3.2 La méthode d'*AdaComp*

AdaComp est une solution pour combiner les principes de compression des gradients et d'adaptation du taux d'apprentissage afin de faire face aux problèmes dus aux délais des modifications. Elle permet de réduire considérablement le trafic entrant au niveau du serveur de paramètres et de garder une précision proche de celle de l'état de l'art.

Raisonnement Nous avons tout d'abord observé que le gradient des paramètres envoyés par les agents était parcimonieux, c'est-à-dire que la majorité des valeurs des gradients sont proches de zéro (comme indiqué dans l'article de Hogwild [89]). La raison à cela vient du type de fonction d'activation (ReLU) utilisée dans la plupart des réseaux de neurones modernes, ainsi que de la taille des batchs relativement petite. Dans un second temps, nous avons remarqué que le délai était géré uniquement avec une granularité au niveau du gradient tout entier dans les méthodes existantes.

À partir de ces deux remarques, nous proposons une méthode pour coupler la compression des modifications avec une prise en compte du délai non plus au niveau des gradients entiers, mais au niveau de chaque paramètre.

Intuitivement, l'utilisation de gradients parcimonieux associée à une prise en charge du délai au niveau de chaque paramètre permet d'augmenter l'efficacité des opérations asynchrones des agents. Le calcul d'un délai indépendant pour chaque paramètre revient à faire l'hypothèse que l'apprentissage de chaque paramètre est indépendant. Bien que fausse en théorie, cette simplification semble raisonnable en pratique (c'est-à-dire, elle est utilisée dans Hogwild [89]).

Méthode de sélection Nous proposons une méthode de sélection du gradient alternative par rapport à Shokri *et al.* [98]. Seule une portion c des gradients les plus grands pour chaque couche du réseau de neurones profond est conservée à chaque itération. Cette méthode de sélection permet de mieux répartir l'apprentissage du réseau de neurones profond sur chacune des couches et donc de garder un apprentissage efficace. De plus, cela permet d'approximer les matrices de gradients plus efficacement qu'une approche de sélection aléatoire (comme montré dans Shokri *et al.* [98]).

Détails de l'algorithme *AdaComp* fonctionne de la manière suivante (décrit également dans les algorithmes en pseudo-codes 1 et 2). Le serveur de paramètres garde une trace de chaque modification reçue, c'est-à-dire, les index des paramètres modifiés à chaque itération ainsi qu'un horodatage. Cet horodatage consiste en un entier incrémenté à chaque itération du serveur de paramètres. Lorsqu'un agent récupère les paramètres θ du serveur de paramètres, il reçoit l'horodatage associé. Il caclcul alors le gradient sur un batch et compresse ce dernier à l'aide de la méthode de sélection. Une fois le gradient compressé, l'agent l'envoie accompagnée de l'horodatage reçu précédemment. Pendant tout ce temps, des modifications ont pu être faites sur

le serveur de paramètres, augmentant ainsi l'horodatage de celui-ci (de t' à t). Au lieu de calculer un délai global pour tout le gradient envoyé par l'agent i ($= t - t'$), nous calculons un délai adapté à chaque paramètre impliqué dans la modification (soit une fraction c des paramètres du réseau de neurones). Le délai associé au paramètre θ_k est le suivant :

$$\sigma_k(t) = \sum_{u=t'}^{t-1} \mathbb{1}_{\{\Delta\theta_k(u) \neq 0\}}, \tag{3.1}$$

avec $\mathbb{1}_A$ la fonction indicatrice égale à 1 si A est vrai et 0 sinon. Le délai $\sigma_k(t)$ est ensuite utilisé pour moduler la modification du paramètre de la manière suivante :

$$\theta_k(t+1) = \theta_k(t) - \alpha_k(t)\Delta\theta_k(t), \tag{3.2}$$

avec

$$\alpha_k(t) = \begin{cases} \alpha/\sigma_k(t) & \text{si } \sigma_k(t) \neq 0 \\ \alpha & \text{sinon.} \end{cases} \tag{3.3}$$

Le paramètre α_k est le taux d'apprentissage calculé pour le k-ième paramètre de θ avec un délai σ_k. Les paramètres de θ_k qui ont été modifiés le plus souvent depuis la dernière mise à jour de l'agent i (avec l'horodatage t'), vont donc avoir un délai plus important. Ce délai va impliquer un taux d'apprentissage plus faible. Cela permet de réduire automatiquement le taux d'apprentissage dans le cas où les mises à jour sont contradictoires ou ont déjà été effectuées par un agent plus rapide. Ce calcul du délai au niveau de chaque paramètre ne peut être fait que dans le cas où les modifications sont parcimonieuses (comme c'est le cas avec *AdaComp*). Dans le cas contraire, chaque paramètre est modifié à chaque itération, et donc le délai de chaque paramètre est le même (c'est-à-dire, le délai de l'itération $t - t'$).

Nos expériences (Section 3.3) montrent que cette adaptation du taux d'apprentissage par rapport au délai de chaque paramètre permet de réduire considérablement la perte de précision due à la compression.

Complexité *AdaComp* implique une augmentation de la complexité au niveau du serveur de paramètres afin de calculer le délai σ_k de chaque paramètre θ_k. En terme de mémoire, le serveur de paramètres doit garder une trace des d dernières modifi-

Algorithm 1 *AdaComp* coté agent

1: **procedure** WORKER(PS, B_i, I, b, c)
2: $k \leftarrow 0$
3: **while** $k < I$ **do**
4: $\theta^{(i)}, k \leftarrow$ Pull_$\theta(PS)$ ▷ récupération de $\theta(k)$ du serveur de paramètres
5: $\Delta\theta^{(i)} \leftarrow$ SGD_STEP(B_i, b)
6: $\Delta\tilde{\theta}^{(i)} \leftarrow$ SELECT_GRAD($\Delta\theta^{(i)}, c$)
7: Push($\Delta\tilde{\theta}^{(i)}, k$) ▷ modification envoyée au serveur de paramètres avec l'hordatage k
8: **end while**
9: **end procedure**
10: **procedure** SGD_STEP(B_i, b)
11: *Selection d'un batch de taille b à partir de la base locale B_i et calcul de $\Delta\theta^{(i)}$ par la méthode de rétro-propagation.*
12: **return** $\Delta\theta^{(i)}$
13: **end procedure**
14: **procedure** SELECT_GRAD($\Delta\theta^{(i)}, c$)
15: *Sélection de $\Delta\tilde{\theta}^{(i)} \subset \Delta\theta^{(i)}$ en gardant les $(100 \times c)\%$ plus grandes valeurs du gradient, en valeur absolue, de chaque matrice W_l et vecteur β_l.*
16: **return** $\Delta\tilde{\theta}^{(i)}$
17: **end procedure**

cations (avec d le délai de l'agent ayant les paramètres les plus en retard). La trace consiste en une liste d'index qui ont été modifiés à chaque itération. Cela représente une complexité en mémoire de $O(d \times c \times |\theta|)$. En terme de calcul, dans le pire des cas (c'est-à-dire lorsque l'agent le plus en retard envoie sa modification), le serveur de paramètres doit faire d recherches d'occurrences des $c \times |\theta|$ index dans les traces de même taille. Si nous supposons que les index ont été envoyés triés (ce qui n'augmente pas la complexité de calcul au niveau de l'agent), la complexité de cette opération est de $O(d \times c \times |\theta|)$ par itération. En moyenne (sur les itérations), le retard est de n. Notons également que d a le même ordre de grandeur que n (le nombre d'agents) dans le cas où les machines ont des temps de latences de même ordre de grandeur. Nous obtenons donc une complexité totale $O(n \times c \times |\theta|)$. Dans le cas classique du modèle serveur de paramètres, la complexité de calcul est de $O(|\theta|)$. Pour les agents, *AdaComp* requiert juste la sélection des $c \times |\theta|$ plus grands gradients en plus du calcul du gradient. Cela peut se faire sans surcoût de complexité ni en terme de mémoire, ni en terme de calcul. Nous avons donc juste modifié la complexité pour le serveur de paramètres.

Algorithm 2 *AdaComp* coté serveur de paramètres

1: **procedure** PS(α, I)
2: *Initialise* $\theta(0)$ *avec des valeurs aléatoires.*
3: **for** $k \leftarrow 0, I$ **do**
4: Get($\Delta\widetilde{\theta}^{(i)}, m$) ▷ Attente d'un gradient
5: $\Delta\theta(k) \leftarrow \Delta\widetilde{\theta}^{(i)}$
6: **for all** $\Delta\theta_j(k) \in \Delta\theta(i)$ **do**
7: $\sigma_j \leftarrow \sum_{u=m}^{k-1} \mathbb{1}_{\{\Delta\theta_j(u)\neq 0\}}$
8: **if** $\sigma_j = 0$ **then** $\alpha_j \leftarrow \alpha$ **else** $\alpha_j \leftarrow \alpha/\sigma_j$
9: $\theta_j(k+1) \leftarrow \theta_j(k) - \alpha_j\Delta\theta_j(k)$
10: **end for**
11: **end for**
12: **end procedure**

3.3 Expériences

3.3.1 Plateforme expérimentale

Pour évaluer l'intérêt de *AdaComp* dans un environnement contrôlé et surveillé, nous avons choisi d'émuler le système global sur un serveur puissant et unique. Le serveur nous permet de réduire les contraintes matérielles et les contraintes liées à la configuration du système distribué. Nous avons choisi de représenter les machines via des containers Linux (LXC) sur un puissant serveur Debian (processeur Intel Xeon E5-2667 v3 @ 3.20GHz, soit 32 coeurs et 1/2 To de RAM). Chaque LXC contient une session Tensorflow afin d'entraîner localement un réseau de neurones profond. De même, un container joue le rôle du serveur de paramètres avec une session Tensorflow. Le trafic entre les LXCs peut être géré depuis le serveur avec des connexions *virtual Ethernet* (ou veth). La Figure 3.4 illustre la plateforme de test qui comporte tous ces éléments.

Une expérience sur cette plateforme se déroule comme suit. Un ensemble de n containers LXC est déployé afin de représenter les agents. Chaque agent d'un container a accès à une proportion $1/n$ de l'ensemble des données d'entraînement. Un container LXC est déployé pour exécuter le code du serveur de paramètres. Tous les agents sont connectés au serveur de paramètres par un réseau virtuel veth. Enfin, un dernier container LXC, que nous appellerons Superviseur, est déployé pour évaluer la précision du modèle avec les paramètres θ. Le Superviseur contient un ensemble de test (distinct de l'ensemble d'apprentissage) ainsi qu'une copie du réseau de neurones

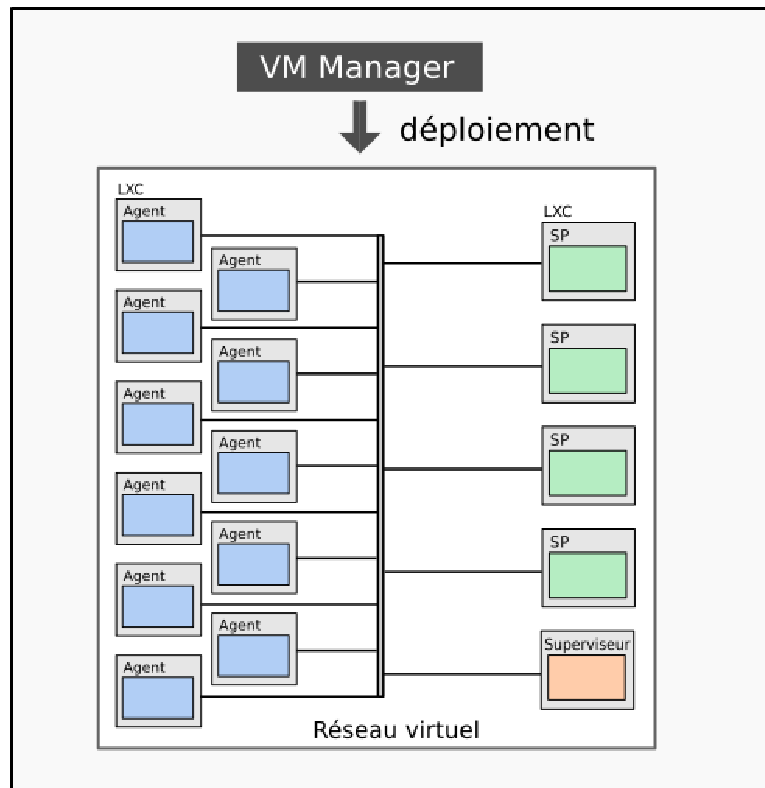


FIGURE 3.4 – Platerforme expérimentale afin de simuler l'environnement distribué. Le VM Manager est composé d'un ensemble de scripts capables de gérer l'activation, la mise en route et l'arrêt des différents containers LXC.

profond.

Au cours de l'exécution, la plateforme ajoute un temps d'attente aléatoire à chaque agent, entre le temps de calcul effectif et l'étape de communication avec le serveur de paramètres. Ce temps garantit que l'ordre des modifications des agents est bien aléatoire, permettant d'imiter la variété possible de puissance de calculs à la disposition des agents. En outre, nous effectuons une expérience sur l'hétérogénéité des machines participantes dans la Section 3.3. Le temps d'exécution sur cette plateforme est d'environ 1/2 journée par expérience (c'est-à-dire, pour atteindre $I = 250\ 000$ itérations).

Avec une telle configuration, la mesure intéressante à observer est la précision atteinte par le modèle final en fonction du nombre d'itérations effectuées. Cette précision reste inchangée, que l'on se trouve sur notre configuration d'émulation ou dans une configuration réelle. Notre plateforme a cependant l'avantage d'être facile à déployer et à superviser notamment pour la surveillance du trafic TCP entre les LXC lors de l'apprentissage.

3.3.2 Configuration expérimentale et compétiteurs

Nous expérimentons *AdaComp* ainsi que notre principale compétiteur [98] sur notre plateforme avec la base de données MNIST. Le but est de d'apprendre un classifieur d'images capable de reconnaître des chiffres écrits à la mains. Les chiffres sont séparés en 10 classes (de 0 à 9). Cette base est composée de $60\ 000$ images, soit $6\ 000$ images par classe, et un ensemble de test de $10\ 000$ images. Chaque image est représentée par une matrice 28×28.

Chaque compétiteur apprend un MLP à 2 couches (soit $109\ 386$ paramètres) et un CNN à 2 couches convolutives et à 2 couches toutes-connectées (soit $211\ 690$ paramètres). Ce dernier réseau de neurones est tiré de la librairie Keras [23] adapté à la base de données MNIST. Dans nos expériences, nous fixons le nombre d'agents $n = 200$, ce qui correspond à l'échelle des expériences reportées dans les travaux de Tensorflow [2]. Nous lançons également un serveur de paramètres (dont les capacités hardware ne sont pas limitées) et un superviseur.

Nous comparons les performances de *AdaComp* à 1) une version classique de la descente de gradient asynchrone (notée ASGD) [70] comme point de comparaison. 2) un algorithme que nous nommons *Comp-ASGD*, similaire à ASGD avec une mise-en-

oeuvre de la méthode de sélection de gradient telle que décrite dans Shokri *et al.* [98]. Pour ces deux compétiteurs, le taux d'apprentissage est adapté au délai de chaque gradient à la manière de Xiangru *et al.* [70].

3.3.3 Précision du modèle

Toutes les expériences sont exécutées jusqu'à ce que le serveur de paramètres ait reçu un total de $I = 250\,000$ gradients des agents (chaque compétiteur a ainsi le temps de faire converger son apprentissage). Chacune des expériences a été effectuée 3 fois. Les graphiques montrent les valeurs moyennes de ces expériences, ainsi que les valeurs minimales et maximales des scores de précision. La courbe a été lissée pour une meilleure compréhension. La précision finale donnée pour chaque méthode correspond à la moyenne des précisions maximales atteintes pour chaque expérience.

La Figure 3.5 (en haut) représente la précision du modèle en fonction du nombre d'itérations pour le modèle MLP. La Figure 3.5 (en bas) indique la précision en fonction du trafic entrant, mesurée à l'entrée du serveur de paramètres pour le modèle du MLP (à droite) et CNN (à gauche). De même, la Figure 3.6 représente la précision en fonction du nombre d'itérations et de la communication entrante du serveur de paramètres.

Les méthodes de ASGD, *Comp-ASGD* et *AdaComp* sont expérimentées dans la même configuration avec une taille de batch : $b = 10$. Pour *Comp-ASGD* et *AdaComp*, nous testons les taux de compressions $c = 0.1$ et $c = 0.01$, représentant respectivement 10% et 1% des paramètres du modèle (en local) envoyés par un agent au serveur de paramètres à chaque itération.

Les Figures 3.5 et 3.6 (en haut), montrent tout d'abord que la compression affecte les résultats de *Comp-ASGD*, alors que dans le cas de *AdaComp*, la précision du modèle n'est pas dégradée, et se trouve même meilleure que la ASGD. Pour le MLP (resp. pour le CNN), la précision finale pour la ASGD est de 95.43%(resp. 97.85%), alors qu'elle est de 94.80% (resp. 96.07%) avec $c = 0.1$ et 92.10% (resp. 95.11%) avec $c = 0.01$ pour *Comp-ASGD* . Nous remarquons que la perte de précision est d'autant plus importante que la compression est importante. Cela s'explique simplement par la perte d'information induite par cette compression. Pour la méthode *AdaComp*, la précision obtenue est de 97.61% pour $c = 0.1$ et 97.44% pour $c = 0.01$ (et 98.7% et 98.59% pour le CNN). Les niveaux de précision obtenus par *AdaComp* sont donc bien meilleur que ceux de *Comp-ASGD* et dépassent même ceux de ASGD. Cela montre

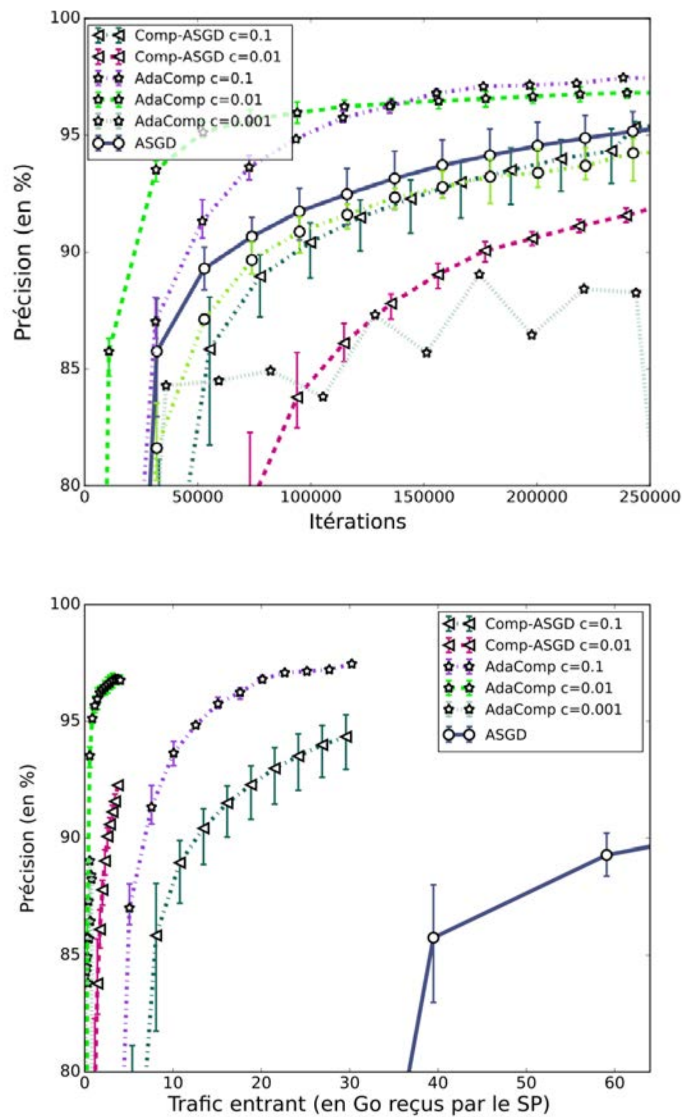


FIGURE 3.5 – Apprentissage du modèle MLP dans un contexte distribué : la précision du modèle en fonction du nombre d'itérations (haut) et en fonction du trafic entrant (bas).

que l'adaptation du taux d'apprentissage au délai de chaque paramètre du modèle est bénéfique pour l'apprentissage. Dans la Figure 3.5, nous avons également testé *AdaComp* avec un niveau de compression extrêment fort : $c = 0.001$. Nous pouvons voir que dans ce cas l'apprentissage n'est pas très stable mais la précision du modèle obtient un score proche de 90% (avant de chuter). Une compression trop importante fait donc baissé les performances d'*AdaComp* .

Nous avons également le résultat de chacune de ces méthodes en fonction du trafic entrant du serveur de paramètres nécessaire dans la Figure 3.6 (en bas). Comme attendu, la quantité de données entrantes dans le serveur de paramètres pour entraîner le réseau de neurones est considérable (jusqu'à 460Go après $250\ 000$ itérations pour le CNN) pour la méthode classique ASGD. Pour les méthodes d'*AdaComp* de *Comp-ASGD*, les effets de la compression du modèle sont particulièrement visibles lorsque c est faible. Cela permet de réduire considérablement la quantité de trafic à l'entrée du serveur de paramètres et de permettre ainsi la réduction du coût de l'apprentissage.

Afin de comparer la quantité de données entrantes nécessaires à l'apprentissage du modèle, nous avons choisi de fixer une précision à atteindre (à un niveau de 97%) par les compétiteurs avec le modèle du CNN. *AdaComp* génère 1.33Go de trafic entrant sur le serveur de paramètres (avec le paramètre $c = 0.01$), ce qui représente 191 fois moins de trafic que celui généré par la descente de gradient. La méthode *Comp-ASGD* quant à elle, n'est pas capable d'atteindre cette précision lors de l'apprentissage.

Nous pouvons conclure que *AdaComp* dépasse les scores de la descente de gradient et de *Comp-ASGD* en terme de précision et de trafic entrant. La compression des modifications par sélection, en plus de réduire le trafic entrant, a également pour effet d'augmenter la précision finale sur notre modèle, dépassant le score d'ASGD. Ce phénomène peut paraître contre-intuitif mais il peut s'expliquer par l'utilisation du taux d'apprentissage adaptatif à l'échelle des paramètres.

Les résultats obtenus nous permettent donc de considérer *AdaComp* comme une méthode capable de déployer l'apprentissage de réseaux de neurones profonds dans un système distribué tel que des machines d'utilisateurs avec des puissances de calculs hétérogènes. Par exemple, pour une *gateway* domestique d'utilisateur connectée 24h/24, les expérimentations testées nécessitent une capacité d'*upload* moyenne de seulement 5Ko/s environ pendant une semaine.

3.3.4 *AdaComp* face aux pannes

AdaComp est une méthode destinée à être utilisée dans les systèmes distribués. Il est donc important de l'expérimenter dans une configuration avec l'apparition de pannes potentielles chez les participants. Nous considérons ici le cas où les agents peuvent s'arrêter à tout moment dans l'apprentissage de manière définitive (les agents peuvent tomber en panne sans prévenir le serveur de paramètres, ni envoyer de mes-

Paramètres			Précision atteinte
Nombres de données dans B_{train}	Pannes	n	
$60,000$ images	non	200	97.44%
$60,000$ images	oui	200	97.17%
$12,000$ images	non	200	95.47%
$12,000$ images	oui	200	95.24%
$6,000$ images	non	100	94.35%

TABLE 3.1 – Précision maximale de *AdaComp*, avec $c = 0.01$ et un taux de pannes de $p = 0.0004$ durant l'expérience.

sage incomplet). De plus, une fois qu'un agent est tombé en panne, la totalité des données qu'il contient est perdue pour le système distribué. Nous fixons la probabilité de panne à $p = 0.0004$ pour chaque agent à chaque itération. Une fois une itération envoyée au serveur de paramètres, nous effectuons un tirage aléatoire entre O et 1 pour voir si une panne apparaît. Si le tirage est inférieur à p, le container contenant l'agent concerné est arrêté pour simuler la panne. Avec ce taux, environ la moitié de la population initiale des participants devrait tomber au panne durant une expérimentation. Dit autrement, 200 agents sont actifs à l'itération $t = 0$ et seulement 100 devraient survivre (en moyenne) jusqu'à l'itération $t = 250\,000$. Les résultats pour *AdaComp* avec $c = 0.01$ pour le modèle MLP sont présentés dans la Table 3.1.

La première observation à faire sur les résultats de *AdaComp* avec les pannes concerne l'impact très faible des pannes sur la précision finale (97.17%, soit seulement 0.27% de moins que la même expérience sans panne). Cela peut s'expliquer par le fait que la base de données d'apprentissage soit suffisamment "riche" pour compenser la perte des données due aux pannes des machines. L'apprentissage, bien que ralenti par la perte en puissance de calcul, finit par trouver des paramètres θ qui permettent d'obtenir un modèle précis.

Dans le but de voir l'impact des pannes dans une base de données d'apprentissage plus petite, nous faisons la même expérience avec seulement 20% des données de la base MNIST pour composer B_{train} (c'est-à-dire, 12 000 images). Dans ce cas, la perte en terme de précision est de 0.23% entre *AdaComp* avec des pannes ($p = 0.0004$, $c = 0.01$) et *AdaComp* sans pannes ($c = 0.01$). Cette perte est quasiment identique à celle de l'expérience précédente avec la base de données MNIST entière.

Afin de mieux mesurer ce phénomène, nous avons réalisé la même expérience avec 100 agents qui participent à la tâche d'apprentissage sans aucune panne et tou-

jours $c = 0.01$. Les 100 agents se partagent seulement 10% de la base de données MNIST, soit le même nombre de données par agent que l'expérience avec 200 agents et 20% de MNIST. Dans cette experience, la précision finale est de 94.35%, contre 95.24% pour l'expérience précédente avec les pannes. Cela signifie donc que les 100 agents qui sont tombés en panne pendant les expériences ont bien participé à l'apprentissage, de telle sorte que l'on remarque un gain dans la précision du modèle finale. Si cela n'avait pas été le cas, l'expérience avec 20% de MNIST et les pannes aurait dû obtenir une précision autour de 94.35%.

Cette expérimentation montre que les pannes potentielles des machines contenant les agents n'impactent que très peu le bon fonctionnement de l'apprentissage. Les effets sont visibles en terme de précision si la base de données d'apprentissage est relativement faible ou lorsqu'une trop grande majorité d'agents tombent en panne avant d'avoir pu contribuer suffisamment au modèle.

3.3.5 *AdaComp* dans un contexte d'agents hétérogènes

Concernant les machines mises à contribution, du fait de nombreux facteurs (contraintes réseaux, capacités de calculs, tâches concurrentes...), il est possible d'avoir des temps de réponse qui peuvent varier de manière significative pour le calcul et l'envoi d'une modification. Il est donc important de prendre en compte ce phénomène lors de nos expériences d'apprentissage distribué.

Pour ce faire, nous avons expérimenté *AdaComp* avec trois classes différentes d'agents : les agents rapides, les agents moyens et les agents lents. Les agents rapides envoient 10 fois plus de modifications au serveur de paramètres que les agents moyens et 100 fois plus que les agents lents. La proportion d'agents dans chacune des classes est fixée respectivement à 30%, 40% et 30%. Nous considérons que les agents ne changent pas de classe durant l'apprentissage. Nous expérimentons *AdaComp* avec $n = 200$ agents et $c = 0.01$ sur la base de données de MNIST en entière ainsi qu'avec la base de données de MNIST réduite (c'est-à-dire, 12 000 images).

La Figure 3.8 affiche la précision du modèle du CNN comme décrit dans la Section 3.3.2 (c'est-à-dire, avec une seule classe d'agents) et avec 3 classes d'agents. La première observation est qu'*AdaComp* avec seulement une classe d'agents obtient la meilleure précision finale (respectivement 98.59% contre 98.36% pour la base de données entière et 97.66% contre 97.00% pour la base de données réduite). Nous

pouvons noter que la différence est plus forte lorsque nous utilisons la base de données réduite. La Figure 3.8 montre cependant qu'*AdaComp* avec 3 classes a tendance à converger plus rapidement, dépassant le score d'*AdaComp* avec une seule classe sur les premières itérations. Cela montre que dans le cas des 3 classes, les agents les plus rapides contribuent plus souvent à l'apprentissage du modèle avec des délais de gradients relativement courts (ce qui améliore l'apprentissage). Les agents les plus lents quant à eux, participent moins à l'apprentissage. Leurs gradients ont des délais longs qui peuvent déstabiliser l'apprentissage dans certains cas (comme c'est le cas sur la courbe avec la base de données réduite). Les données de ces agents sont donc moins prises en compte pour l'apprentissage, ce qui explique la différence de précision finale. Notons cependant que cette différence reste relativement faible, surtout dans le cas où B_{train} contient beaucoup de données.

Cette expérimentation montre que la distribution de l'apprentissage sur des agents avec des puissances de calculs hétérogènes, comme nous pouvons nous y attendre sur un déploiement réel, est réalisable puisque le modèle converge. Cela tend à réduire les contributions des agents les plus lents face aux agents les plus rapides. Comme dans le cas de l'expérience précédente avec les pannes, l'impact de cette hétérogénéité sur la précision finale dépend de la richesse de l'ensemble des données d'apprentissage B_{train}.

3.4 Discussion sur *AdaComp* et les travaux connexes

Nous avons vu en Section 1.2.5 qu'il existait de nombreuses alternatives à la descente de gradient stochastique pour l'apprentissage profond tel que la méthode du momentum [91], Adagrad [30], ou encore Adam [58]. Ces méthodes sont cependant adaptées à l'apprentissage centralisé. De plus, le délai du gradient a déjà des effets proches de la méthode du momentum. Ces méthodes ne sont donc pas directement adaptables pour l'apprentissage distribué. Cependant, des travaux sur l'adaptation de Adam avec *AdaComp* pourraient potentiellement améliorer la vitesse d'apprentissage dans notre configuration de systèmes distribués.

Le serveur de paramètres utilisé par *AdaComp* est une méthode populaire pour distribuer l'apprentissage. Par exemple, DistBelief [28], Adam [22], et TensorFlow [2] utilisent généralement cette méthode à l'interieur d'un *datacenter*. Cependant, dans notre contexte d'apprentissage dans un système distribué avec de nombreux partici-

pants, les problèmes causés par les gradients avec délais sont amplifiés. Ils peuvent être résolus *1*) de manière algorithmique ou *2*) par le biais de plus de synchronisations.

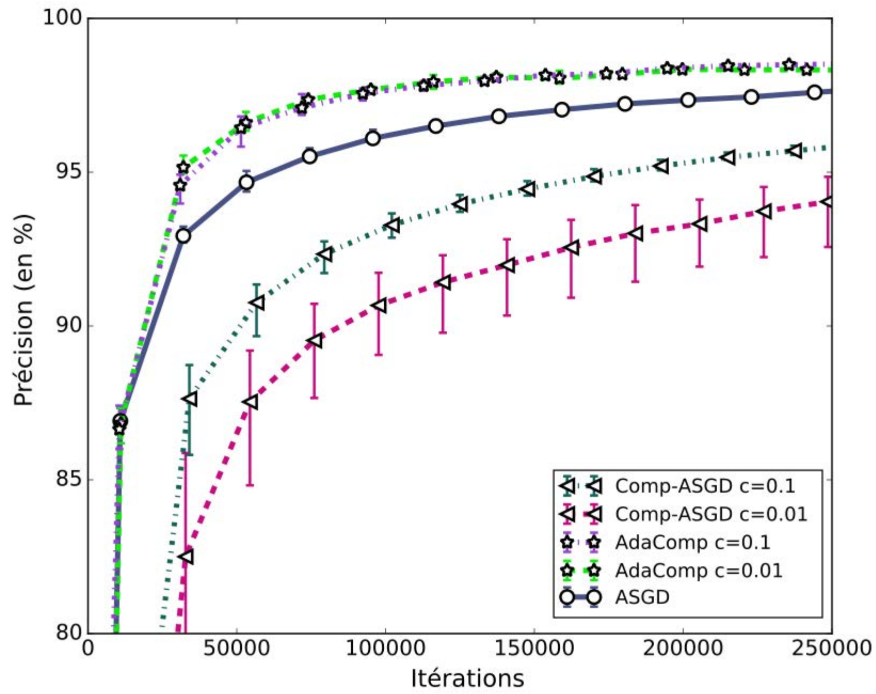
1) Dans leurs travaux [118], W. Zhang et *al.* proposent d'adapter le taux d'apprentissage en fonction du retard de l'agent pour chaque gradient. Le serveur de paramètres divise le taux d'apprentissage par le délai de la modification. Cette méthode limite donc l'impact des gradients avec un grand délai. Plus récemment, Odena [81] propose de maintenir une variance moyenne de chaque paramètre. Associé aux délais du gradient, cette variance permet de pondérer plus précisément les modifications de chaque paramètres. Le taux d'apprentissage est alors grand lorsque peu de changement ont été effectués sur un paramètre, et faible dans le cas contraire. Cette méthode est proche d'*AdaComp*, mais ne prend pas en compte l'utilisation de gradients compressés parcimonieux.

2) Une solution simple pour éviter les gradients avec délais est de synchroniser les agents à chaque itération avec une descente de gradient synchrone. Cependant, l'attente de l'ensemble des agents à chaque itération est très coûteuse en temps. W. Zhang *et al.* [118] proposent le protocole de la descente de gradient n-*softSync* dans lequel le serveur de paramètres attend une fraction de l'ensemble des gradients à chaque itération avant de calculer une modification. Un gradient plus précis est alors obtenu et le retard moyen des agents est réduit. Un autre article de Chen *et al.* [20] montre qu'une descente de gradient synchrone peut être très efficace si le serveur de paramètres n'attend pas les k derniers agents à chaque itération. Cette méthode n'est cependant pas intéressante si la base de données est distribuée sur les agents : les données des agents les plus lents ne sont jamais utilisées dans ce cas. Dans notre configuration de système distribué, nous n'avons pas de garantie sur une borne supérieure du temps de réponse d'un agent. Cela motive l'utilisation de méthodes asynchrones comme *AdaComp* .

En parallèle de nos recherches, la méthode du Federated Learning [61, 60, 76] a été proposée comme un bon compromis entre communications et performances. Contrairement à *AdaComp*, cette méthode utilise la moyenne de modèles comme technique de mise en commun de l'apprentissage (et non une descente de gradient asynchrone). Nous verrons dans les Chapitres 4 et 5 comment cette méthode est utile à l'apprentissage et comment nous y comparer.

3.5 Conclusion

Dans ce chapitre nous avons montré quelles étaient les implications les plus importantes pour l'apprentissage d'un réseau de neurones profond sur un système distribué composé de machines d'utilisateurs. Du fait des contraintes matérielles et réseaux des participants, la descente de gradient asynchrone est une solution naturelle pour distribuer la tâche d'apprentissage sur celles-ci. Elle demande cependant un trafic considérable si l'algorithme n'est pas adapté. C'est pourquoi nous proposons *AdaComp*, un nouvel algorithme basé sur l'architecture du serveur de paramètres. Celui-ci permet de compresser les modifications tout en adaptant le taux d'apprentissage pour chaque paramètre afin d'apprendre un réseau de neurones avec un trafic réduit entre agents et serveur de paramètres. Nous avons montré dans nos expériences une réduction de 191 fois le trafic entrant nécessaire à l'apprentissage comparé à la méthode classique de descente de gradient asynchrone pour la base de données MNIST. De plus, *AdaComp* obtient une précision finale 1% à 2% meilleure que la descente de gradient asynchrone classique sur les modèles étudiés. Enfin, nous avons montré qu'*AdaComp* était capable de faire face à des pannes des machines participantes lors de l'apprentissage mais également à des puissances de calculs très hétérogènes. Ces résultats nous permettent de penser que *AdaComp* peut être utilisée comme technique pour distribuer l'apprentissage d'un réseau de neurones profonds sur des machines d'utilisateurs avec un moindre coût.



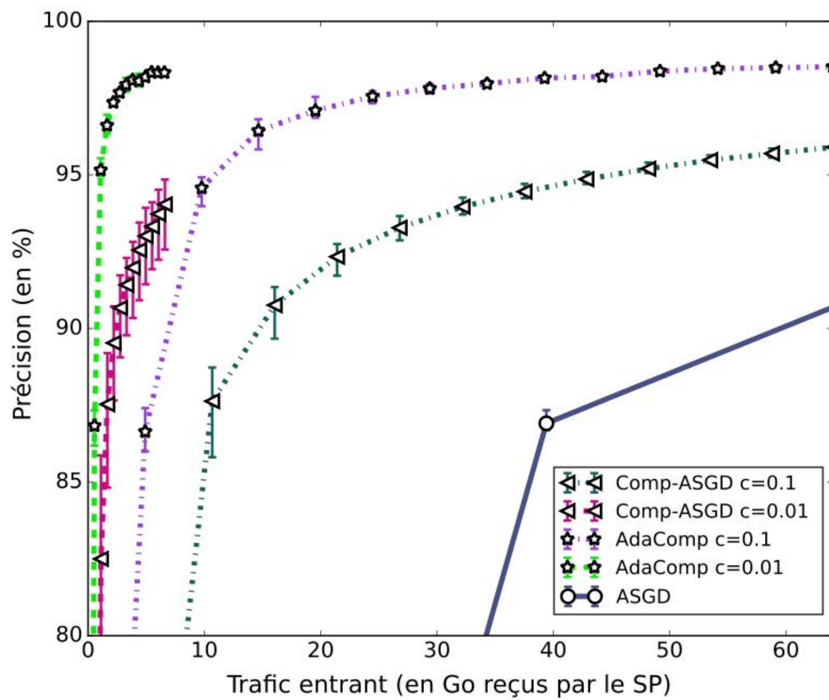


FIGURE 3.6 – Apprentissage du modèle CNN dans un contexte distribué : La précision du modèle en fonction du nombre d'itérations (Haut) et en fonction du trafic entrant (Bas).

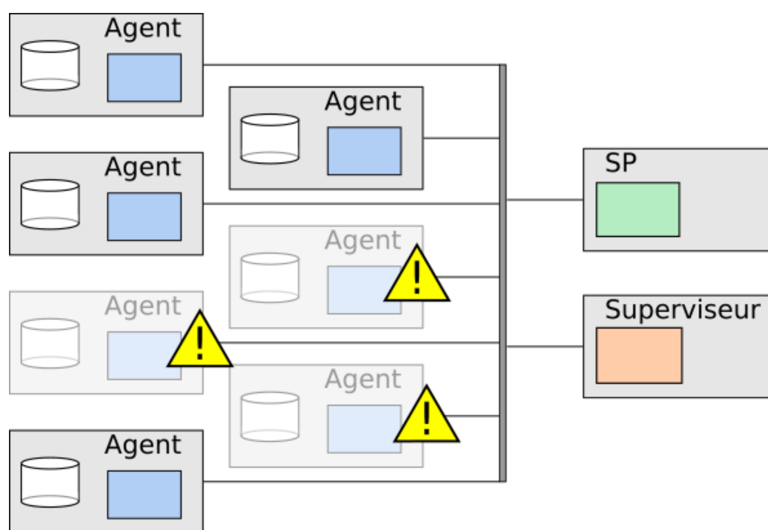


FIGURE 3.7 – Exemple d'environnement avec des pannes durant l'apprentissage. Les machines indiquées avec des points d'exclamation sont déconnectées du réseau et ne peuvent donc apporter aucune modification au serveur de paramètres.

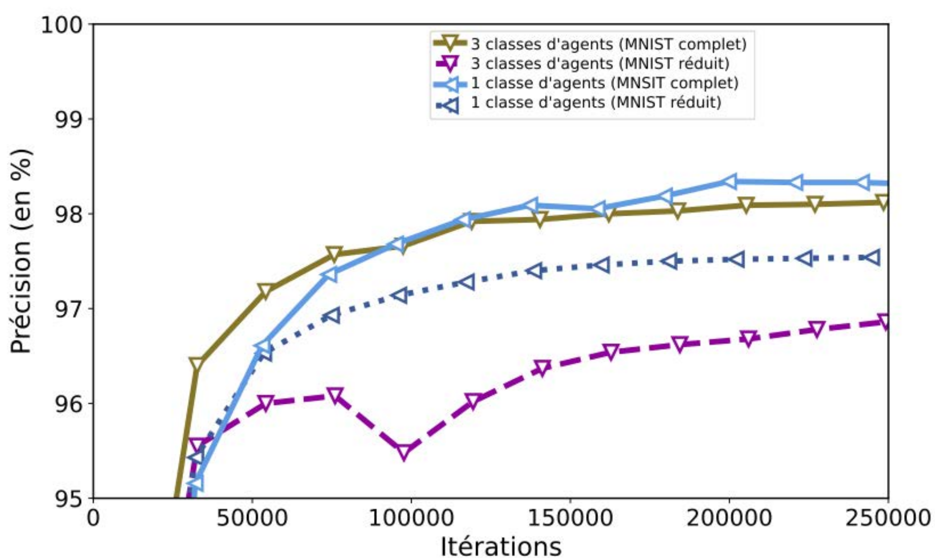


FIGURE 3.8 – Apprentissage du modèle du CNN avec 3 classes d'agents (machines hétérogènes en puissance de calcul).

CHAPITRE 4

PROTOCOLE DE RUMEUR POUR L'APPRENTISSAGE DES RÉSEAUX ANTAGONISTES GÉNÉRATIFS

Dans ce chapitre, nous nous intéressons à l'apprentissage d'un type de réseau de neurones appelé réseaux antagonistes génératifs (ou GAN pour *Generative Adversarial Network*), présenté dans la Section 1.3. Ce type de réseau de neurones est particulièrement intéressant car il permet de générer de nouvelles données à partir d'une distribution inconnue. De plus, les GAN sont utilisés pour de nombreuses applications, principalement en image, comme la génération d'images à partir d'un texte [90], l'édition d'images existantes [87], ou la création de vidéo à partir d'images [109]. Ils peuvent également être utilisés pour de la compression [6] ou l'augmentation de la résolution [68] d'images. Il est donc intéressant de pouvoir entraîner ce type de réseaux de neurones de manière collaborative comme décrit dans le Chapitre 2.

Nous avons vu dans le chapitre précédent une méthode pour effectuer une descente de gradient asynchrone sur un ensemble de machines. Bien que cette méthode montre de bons résultats et puisse s'appliquer à tout type de réseaux de neurones (comme les GAN), elle nécessite un serveur central et des communications fréquentes entre serveur et machines participantes. Les techniques de mise en commun par modèle moyen, comme dans le cas du *Federated Learning* [61] nécessitent moins de communications (le modèle est envoyé après avoir vu plusieurs fois toutes les données en local). Cependant le *Federated Learning* demande toujours un serveur de paramètres afin de centraliser les modèles lors de la mise en commun. Les protocoles de rumeur [16] (*Gossip* en anglais) sont également utilisés pour mettre en commun l'apprentissage d'un réseau de neurones profond sur plusieurs machines [13]. Ces méthodes ont l'avantage de pouvoir fonctionner de manière totalement décentralisée (c'est-à-dire, sans serveur de paramètres). De plus, les communications peuvent se

faire de manière asynchrone et les protocoles de rumeur sont résistants aux pannes.

Dans ce chapitre, nous évaluons la capacité des méthodes utilisant un protocole de rumeur afin d'apprendre un GAN. Nous comparons ces méthodes à la méthode du *Federated Learning* qui nécessite un serveur central.

4.1 Apprentissage collaboratif d'un GAN

4.1.1 Modèle du GAN

Comme nous l'avons vu à la Section 1.3, les réseaux antagonistes génératifs que nous appelons GAN, font partie des réseaux de neurones profonds. La particularité des GAN, tels qu'ils sont présentés dans les travaux de I. Goodfellow [39], est que leur phase d'apprentissage est non-supervisée, c'est-à-dire qu'aucun label (ou description) n'est requis pour l'apprentissage à partir des données. Un GAN classique est composé de deux éléments : un *générateur* $\mathcal{G}$ et un *discriminateur* $\mathcal{D}$. Les deux sont des réseaux de neurones profonds. Le générateur prend en entrée un signal bruité (par exemple, des vecteurs aléatoires de taille k où chaque entrée suit une distribution normale $\mathcal{N}(0,1)$) et génère des données au même format que les données d'apprentissage en entrée (par exemple, une image de 128x128 pixels et 3 canaux de couleurs). Le discriminateur est un classifieur binaire qui reçoit en entrée des données de ces deux sources : des données générées du générateur ou des données réelles de la base de données d'apprentissage. L'objectif du discriminateur est de déterminer de quelle source proviennent les données (Voir Figure 1.21).

Formellement, considérons une base de données d'apprentissage B_{train} inclues dans un espace de données X, où $\boldsymbol{x} \in B_{train}$ suit la distribution probabiliste P_{data}. Un GAN, composé d'un générateur $\mathcal{G}$ et d'un discriminateur $\mathcal{D}$, a pour objectif d'apprendre cette distribution. Comme proposé dans les travaux de I. Goodfellow [39], nous modélisons le générateur par la fonction $\mathcal{G}_{\boldsymbol{\lambda}} : \mathbb{R}^{\eta} \longrightarrow X$ où $\boldsymbol{\lambda}$ contient les paramètres du réseau de neurones $\mathcal{G}$ et η est la dimension de l'espace latent (c'est-à-dire, l'espace d'entrée du générateur). De la même manière, nous modélisons le discriminateur par la fonction $\mathcal{D}_{\boldsymbol{\theta}} : X \longrightarrow [0,1]$ où $\mathcal{D}_{\boldsymbol{\theta}}(\boldsymbol{x})$ représente la probabilité que $\boldsymbol{x}$ appartienne à la base de données d'apprentissage, et $\boldsymbol{\theta}$ contient les paramètres du discriminateur $\mathcal{D}_{\boldsymbol{\theta}}$. Considérant le fonction $\log$ comme étant le logarithme en base 2, l'apprentissage

consiste à trouver les paramètres $\boldsymbol{\lambda}^*$ pour le générateur :

$$\boldsymbol{\lambda}^* = \arg\min_{\boldsymbol{\lambda}} \max_{\boldsymbol{\theta}} (A_{\boldsymbol{\theta}} + B_{\boldsymbol{\theta},\boldsymbol{\lambda}}), \text{ avec}$$

$$A_{\boldsymbol{\theta}} = \mathbb{E}_{\boldsymbol{x}\sim P_{\text{data}}} [\log \mathcal{D}_{\boldsymbol{\theta}} (\boldsymbol{x})] \text{ et}$$

$$B_{\boldsymbol{\theta},\boldsymbol{\lambda}} = \mathbb{E}_{\boldsymbol{z}\sim\boldsymbol{\mathcal{N}}_\eta} [\log (1 - \mathcal{D}_{\boldsymbol{\theta}} (\mathcal{G}_{\boldsymbol{\lambda}}(\boldsymbol{z}))] ,$$

où $\boldsymbol{z} \sim \boldsymbol{\mathcal{N}}_\eta$ signifie que les entrées du vecteur $\boldsymbol{z}$ de dimension η sont indépendantes et identiquement distribuées suit une distribution normale avec des paramètres fixes. Dans cette équation, $\mathcal{D}$ ajuste ses paramètres θ pour maximiser $A_{\boldsymbol{\theta}}$, c'est-à-dire pour maximiser la bonne classification des données réelles (avec "1" en sortie de $\mathcal{D}$) et maximiser $B_{\boldsymbol{\theta},\boldsymbol{\lambda}}$, c'est-à-dire maximiser la bonne classification des données générées (avec "0" en sortie de $\mathcal{D}$). $\mathcal{G}$ ajuste ses paramètres $\boldsymbol{\lambda}$ pour minimizer $B_{\boldsymbol{\theta},\boldsymbol{\lambda}}$ ($\boldsymbol{\lambda}$ n'ayant pas d'impact sur A), ce qui signifie que $\mathcal{G}$ essaie de maximiser l'erreur faite par $\mathcal{D}$ sur les données générées. L'apprentissage est réalisé en itérant deux étapes, appelées l'étape d'apprentissage du discriminateur et l'étape d'apprentissage du générateur, comme décrit ci-après.

Apprentissage du discriminateur

La première étape consiste à apprendre $\boldsymbol{\theta}$ pour un $\mathcal{G}_{\boldsymbol{\lambda}}$ fixe. Le but est d'approximer les paramètres $\boldsymbol{\theta}$ qui maximisent $A_{\boldsymbol{\theta}} + B_{\boldsymbol{\theta},\boldsymbol{\lambda}}$ avec le $\boldsymbol{\lambda}$ actuel. Cette étape est effectuée par quelques itérations d'une descente de gradient (généralement avec la méthode d'optimisation Adam [58]) de la fonction d'erreur du discriminateur J_{disc} sur les paramètres $\boldsymbol{\theta}$:

$$J_{disc}(X_r, X_g) = \widetilde{A}(X_r) + \widetilde{B}(X_g), \text{ avec}$$

$$\widetilde{A}(X_r) = \frac{1}{b} \sum_{\boldsymbol{x}\in X_r} \log(\mathcal{D}_{\boldsymbol{\theta}}(\boldsymbol{x})); \widetilde{B}(X_g) = \frac{1}{b} \sum_{\boldsymbol{x}\in X_g} \log(1 - \mathcal{D}_{\boldsymbol{\theta}}(\boldsymbol{x})),$$

où X_r est un batch de b données réelles tirées au hasard de B_{train} et X_g un batch de b données générées à partir du générateur $\mathcal{G}$. Dans le document original [39], les auteurs proposent d'effectuer plus d'itérations d'apprentissage sur les paramètres $\boldsymbol{\theta}$ de $\mathcal{D}$ par rapport au paramètres $\boldsymbol{\lambda}$ de $\mathcal{G}$. Nous notons ce rapport L (c'est-à-dire, $\mathcal{D}$ effectue L fois plus d'itérations que $\mathcal{G}$).

Apprentissage du générateur

La deuxième étape consiste à adapter $\boldsymbol{\lambda}$ aux nouveaux paramètres $\boldsymbol{\theta}$. Comme pour la première étape, elle est effectuée par une descente sur gradient de la fonction d'erreur J_{gen} sur les paramètres du générateur $\boldsymbol{\lambda}$:

$$\begin{aligned} J_{gen}(Z_g) &= \widetilde{B}\left(\{\mathcal{G}_{\boldsymbol{\lambda}}(\boldsymbol{z}) | \boldsymbol{z} \in Z_g\}\right) \\ &= \frac{1}{b} \sum_{\boldsymbol{x} \in \{\mathcal{G}_{\boldsymbol{\lambda}}(\boldsymbol{z}) | \boldsymbol{z} \in Z_g\}} \log(1 - \mathcal{D}_{\boldsymbol{\theta}}(\boldsymbol{x})) \\ &= \frac{1}{b} \sum_{\boldsymbol{z} \in Z_g} \log(1 - \mathcal{D}_{\boldsymbol{\theta}}(\mathcal{G}_{\boldsymbol{\lambda}}(\boldsymbol{z}))) \end{aligned}$$

où Z_g est un échantillon de b vecteurs aléatoires de dimension η, générés à partir de $\boldsymbol{\mathcal{N}}_\eta$. Contrairement à l'étape d'apprentissage discriminatoire, cette étape n'est effectuée qu'une fois par itération.

En itérant de nombreuses fois ces deux étapes avec des données différentes (voir par exemple, [39] ou [8] pour les questions relatives à la convergence), le GAN fini par trouver un $\boldsymbol{\lambda}$ qui approxime $\boldsymbol{\lambda}^*$. En dépit de cette avancée très récente, il existe de nombreuses propositions alternatives pour apprendre un GAN (plus de détails peuvent être trouvés dans [9, 82], et [93]).

4.1.2 GAN et systèmes distribués

Dans le cas d'un apprentissage collaboratif entre différents utilisateurs, le GAN final doit être capable de générer des données réalistes qui sembleraient appartenir à la base de données globale $B_{train} = \bigcup_{i=0}^{n} B_i$ (avec B_i la base de données locale à l'utilisateur i). L'avantage des GAN dans notre contexte est de pouvoir être appris de manière non-supervisée. Cela signifie que les données n'ont pas être labellisée sur les machines des participants. S'il est cependant possible de labelliser les données, ou d'y ajouter des informations, les modèles de GAN tels que ceux décrits dans les travaux de [82, 90] peuvent être utilisés. Ainsi le générateur contiendra toutes les connaissances possibles sur ces données. Si l'apprentissage s'est bien effectué (sans sur-apprentissage), le générateur ne devrait pas être capable de générer des données réelles appartenant à un participant en particulier (les connaissances sur chaque

donnée réelle sont agrégées et simplifiées dans son modèle). Il n'est cependant pas impossible que des données générées soient proches de données personnelles. Des travaux de recherche récents[119, 114] ont proposé des méthodes permettant de préserver la vie privée concernant les données des participants par rapport aux données générées.

Nous allons ici présenter deux méthodes d'apprentissage dans le contexte de base de données distribuée. La première, basée sur le *Federated Learning*, nécessite un serveur de paramètres. La seconde peut être utilisée de manière décentralisée, permettant un apprentissage collaboratif sans serveur central.

FL-GAN : *Federated Learning* pour GAN

De par la conception des GAN, un générateur et un discriminateur sont deux éléments distincts qui sont pourtant étroitement couplés. Ce fait permet néanmoins d'envisager d'adapter une méthode de calcul connue, généralement utilisée pour la formation d'un seul réseau de neurones profond. Le *Federated Learning* [61] propose de former un modèle d'apprentissage, et en particulier un réseau de neurones profond, sur un ensemble d'agents (voir Section 2.3.2). Il utilise la méthode du serveur de paramètres, avec la particularité que les agents effectuent de nombreuses itérations localement entre chaque communication avec le serveur central (c'est-à-dire, un round), au lieu d'envoyer des gradients régulièrement. De plus, tous les agents ne sont pas nécessairement actifs à chaque round. Pour mettre en commun leur modèle, tous les agents actifs calculent un modèle moyen sur le serveur au début de chaque round.

Afin de comparer les méthodes de rumeur à une configuration de type *Federated Learning*, nous proposons une version adaptée du *Federated Learning* aux GAN. Cette adaptation considère le discriminateur $\mathcal{D}$ et le générateur $\mathcal{G}$ de chaque opérateur comme un seul modèle à traiter de manière atomique (l'ensemble des paramètres est donc composé des paramètres $\boldsymbol{\theta}$ et $\boldsymbol{\lambda}$. Chaque agent i est considéré comme actif (c'est-à-dire, qu'il participe à l'apprentissage) et possède les mêmes architectures $\mathcal{G}$ et $\mathcal{D}$ avec leurs propres paramètres $\boldsymbol{\theta}_i$ et $\boldsymbol{\lambda}_i$ en local ainsi qu'une base de données d'apprentissage B_i. Les agents effectuent des itérations sur leurs données et à chaque période de E *époques* (c'est-à-dire, lorsque chaque agent a utilisé E fois la totalité de ses données pour l'apprentissage de son GAN), il envoie ses paramètres au serveur de paramètres. À son tour, le serveur calcule la moyenne des paramètres $\mathcal{G}$ et $\mathcal{D}$ de tous les agents, afin d'envoyer le modèle mis à jour à ces agents pour commen-

cer un nouveau round. Nous nommons cette version FL-GAN ; elle est illustrée par la Figure 4.1 b).

***Gossip* GAN : Protocol de rumeur pour GAN**

Le protocole de rumeur permet de faire circuler une information parmi un grand nombre de machines interconnectées par des communications pair-à-pair. Cette méthode a été utilisée par M. Blot *et al.* [13] afin de diffuser les valeurs moyennes des paramètres d'un réseau de neurones entre les agents lors d'un apprentissage. Nous adaptons ici cette méthode afin de la comparer au *Federated Learning* sur l'apprentissage de GAN. De même que pour le FL-GAN, nous considérons dans cette première version, le modèle $\mathcal{G}$ et $\mathcal{D}$ comme un unique modèle. Chaque agent i possède la même architecture de $\mathcal{G}$ et $\mathcal{D}$ avec ses propres paramètres $\boldsymbol{\theta}_i$ et $\boldsymbol{\lambda}_i$ en local ainsi qu'une base de données d'apprentissage B_i. Les agents effectuent plusieurs rounds composés des étapes suivantes :

— les agents effectuent une descente de gradient stochastique (ou une variante) jusqu'à avoir fait E époques en local,
— chaque agent i envoit ses paramètres $\boldsymbol{\theta}_i$ et $\boldsymbol{\lambda}_i$ à un de ses voisins, choisi de manière aléatoire,
— chaque agent i fait la moyenne entre ses paramètres en local $\boldsymbol{\theta}_i$ et $\boldsymbol{\lambda}_i$ et ceux reçus par ses voisins

Afin de simplifier nos expérimentations, nous considérons par la suite que chaque agent ne reçoit les paramètres que d'un seul de ses voisins, ou de lui-même. Les communications de cette méthode sont illustrées dans la Figure 4.1. c).

Variante : *Gossip_ind* GAN Dans cette variante, nous considérons $\mathcal{G}$ et $\mathcal{D}$ comme deux modèles indépendants lors de la communication pair-à-pair. L'idée est de pouvoir diversifier les couples d'adversaires à la manière des travaux de J. Im *et al* [56]. À chaque communication, l'agent i reçoit donc des paramètres $\boldsymbol{\theta}_j$ de l'agent j et les paramètres $\boldsymbol{\lambda}_{j'}$ de l'agent j'. De la même manière que *Gossip* GAN, les agents font la moyenne entre leurs modèles en local et les modèles reçus dans la variante *Gossip_ind* GAN. Cette méthode est illustrée sur la Figure 4.1 d).

4.2 FL-GAN vs Gossip GAN : une comparaison expérimentale

4.2.1 Configuration des expériences

Nous expérimentons sur une plateforme composée de multiple CPUs et GPUs. Nous simulons n machines, chacune contenant une part égale (i.i.d.) de la base de données d'apprentissage, de sorte qu'un GAN par machine soit formé et que les partages de données ne quittent jamais leur machine d'origine. Nous utilisons la base de données MNIST, composée de $60\ 000$ images de chiffres manuscrits de taille 28×28 pixels. L'ensemble de données est composé de 10 classes (nous n'utilisons pas les labels de chaque classe dans les expériences car l'apprentissage des GAN est non-supervisé).

Le modèle de GAN utilisé provient des modèles open source de la plateforme Tensorflow : $\mathcal{G}$ est composé de deux couches toutes-connectées de tailles 1024 et 6272, suivies de 2 couches de déconvolution avec respectivement un nombre de filtres de 64 et 32, et une couche finale de convolution utilisant 1 filtre. $\mathcal{D}$ est composé de 2 couches de convolution avec respectivement 64 et 128 filtres, suivies de deux couches toutes connectées de tailles 1024 et 1. Nous utilisons la même configuration d'entraînement que dans le cas centralisé. $\mathcal{G}$ prend en entrée des batchs de vecteurs aléatoires de taille 128 où chaque élément est généré à partir d'une distribution normale $\mathcal{N}(0, 1)$. Nous fixons le nombre total d'itérations à $I = 20000$ pour $\mathcal{G}$ dans toutes les expériences.

Les différents compétiteurs *(i)* Une version dite *standalone* (Figure 4.1 *a*), destinée à illustrer l'apprentissage de la base de données distribuée, mais sans collaboration entre les machines. Dans cette configuration, un GAN est formé sur chaque machine mais aucune communication entre eux n'est possible. Chaque agent i utilise uniquement les données locales B_i pour apprendre son GAN.
(ii) L'approche FL-GAN décrite en Section 4.1.2 (Figure 4.1 *b*). Chaque machine contient un agent avec sa base de données B_i et un GAN. Tous les GAN sont initialisés avec les mêmes paramètres au début de l'apprentissage.
(iii) La méthode basée sur le protocole de rumeur et présentée en Section 4.1.2 appelée *Gossip* GAN (Figure 4.1 c). Contrairement au FL-GAN, cette méthode ne nécessite pas de serveur central, ce qui réduit le phénomène de goulot d'étranglement.

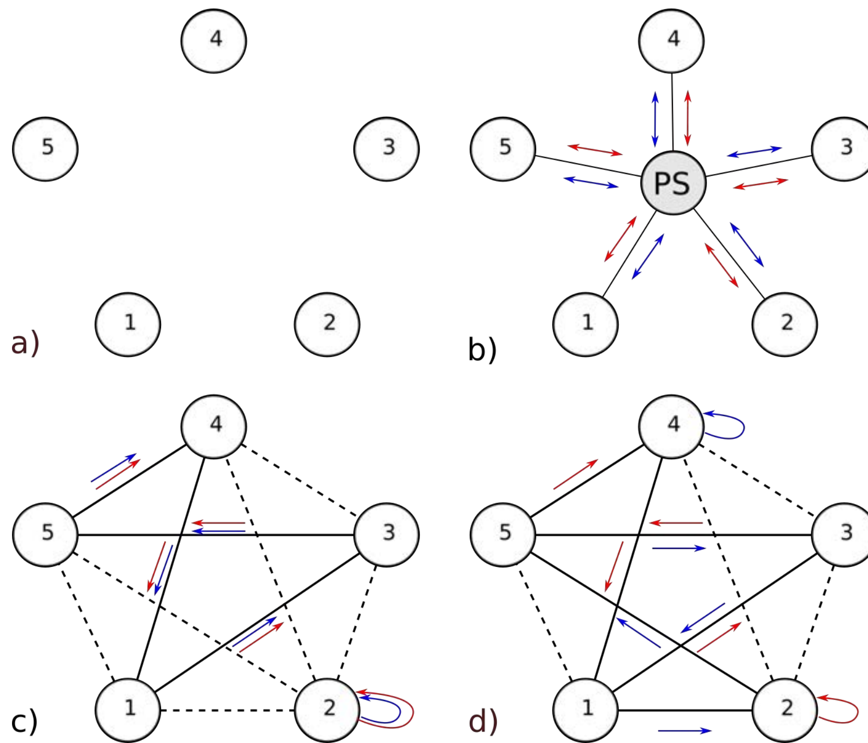


FIGURE 4.1 – Les différentes configurations de communications considérées : *a) Standalone* (c'est-à-dire, pas de collaboration entre les agents), *b)* le FL-GAN, *c) Gossip* GAN, et *d) Gossip_ind* GAN. Les flèches rouges et bleues représentent respectivement les mouvements des générateurs $\mathcal{G}$ et des discriminateurs $\mathcal{D}$.

(iv) La variante de *Gossip* GAN, nommée *Gossip_ind* GAN présentée également dans la Section 4.1.2 (Figure 4.1d). Celle-ci a les mêmes caractéristiques que *Gossip* GAN à la seule différence que les paramètres de $\mathcal{G}$ et de $\mathcal{D}$ sont envoyés à des machines différentes à chaque round de communication.

Mesures de performances des GAN Évaluer les modèles générateurs comme les GAN est une tâche difficile. Idéalement, cela requiert une validation humaine pour juger de la qualité des données générées. Heureusement, dans le domaine des GAN, des méthodes intéressantes sont proposées pour simuler ce jugement humain. La principale est appelée *Inception Score* (ou score d'Inception), que l'on notera IS. Ce score a été proposée par Salimans *et al.* [93]. Il a été montré qu'elle correspondait à un jugement humain sur le réalisme des images qu'elle évalue. La méthode IS consiste à appliquer un classifieur Inception [102] pré-entraîné sur les données générées. Le score d'Inception évalue la confiance sur la classification des données générées et sur

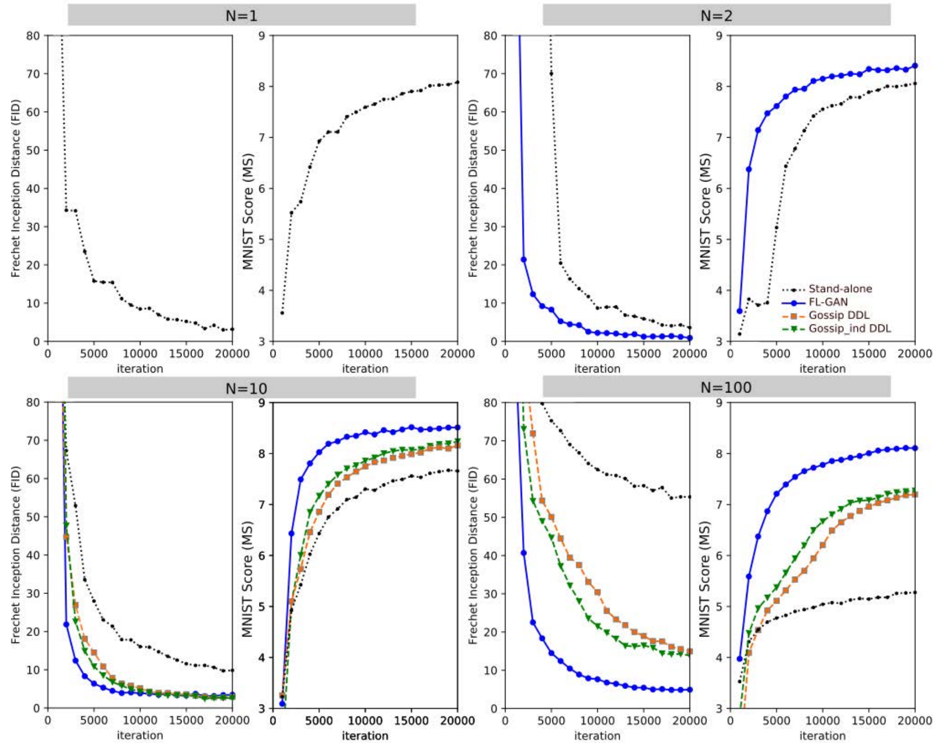


FIGURE 4.2 – Score MNIST (MS) et distance *Inception* de Fréchet (FID) pour les quatre compétiteurs avec $n \in \{1, 2, 10, 100\}$. Plus MS est grand et meilleur est le score. Inversement, plus la FID est grand, moins le score est bon.

la diversité des données en sortie. Un bon score sur des données générées assure donc que celles-ci sont bien réalistes et qu'elles se sont pas toutes les mêmes. Pour évaluer les concurrents sur MNIST, nous utilisons le score MNIST (que l'on nommera MS), similaire à le score d'Inception, mais qui utilise un classifieur adapté aux données du MNIST plutôt que le réseau de neurones Inception. Heusel *et al.* proposent une deuxième métrique appelée la distance Inception de Fréchet (Fréchet Inception Distance - FID) dans leur travaux [47]. Le FID mesure la distance entre la distribution des données générées par $\mathcal{G}$ ($P_{\mathcal{G}}$) et la distribution des données réelles P_{data}. Il applique le réseau Inception sur un échantillon de données générées et sur un échantillon de données réelles et suppose que les résultats de ces deux types de données ont des distributions gaussiennes. Le FID calcule la distance de Fréchet entre la distribution gaussienne obtenue en utilisant les données générées et la distribution gaussienne obtenue à partir des données réelles. De même que pour le score d'Inception, nous

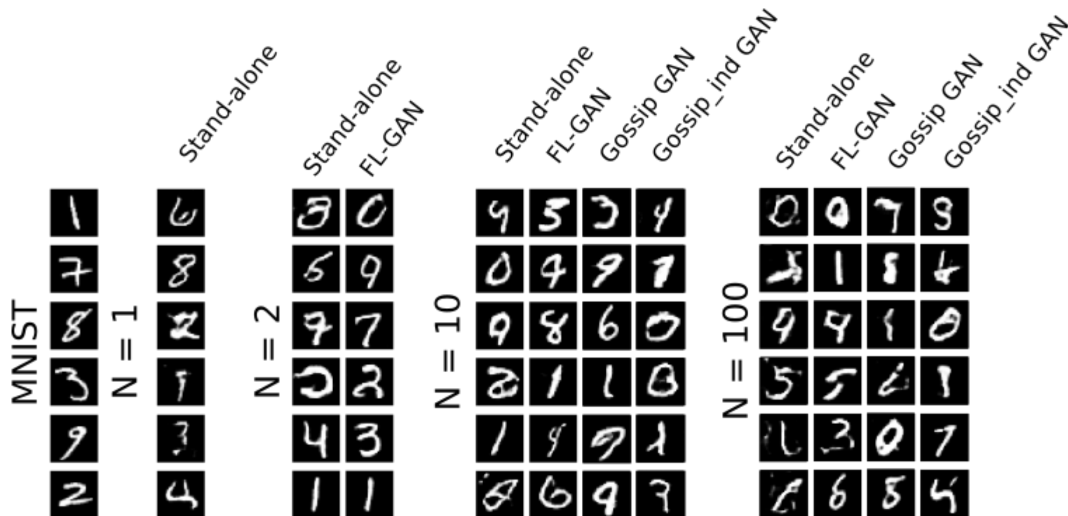


FIGURE 4.3 – Échantillons d'images générées par le meilleur GAN de chaque méthode concurrente.

utilisons un classifieur mieux adapté pour calculer le FID sur l'ensemble de données MNIST. Nous utilisons l'implémentation de MS et de FID disponible dans Tensorflow [1].

4.2.2 Passage à l'échelle et performances

Nous considérons que chaque machine héberge $1/n$ de l'ensemble de données MNIST, divisé de manière aléatoire (avec une distribution i.i.d.). La taille de la base de données totale B_{train} reste constant ($60\ 000$ images). L'objectif est d'évaluer les performances des différents concurrents lorsque la base de données est de plus en plus distribuée. Nous exécutons chacune des approches avec 200 itérations locales entre deux étapes de communication (non applicable pour la méthode *standalone*) et $I = 20\ 000$ itérations de descente de gradient (avec la méthode d'optimisation Adam [58]) au niveau du générateur. Le nombre de agents pour l'apprentissage varie avec $n \in \{1, 2, 10, 100\}$. Notez que $n = 1$ correspond au cas centralisé et que pour $n = 2$, FL-GAN et les deux méthodes de rumeur sont équivalentes, c'est-à-dire, que les paramètres des deux agents sont moyennés à chaque étape de la communication. Seule la méthode de FL-GAN est donc représentée.

1. Le code disponible à l'adresse https://github.com/tensorflow/models/blob/master/research/gan/mnist/util.py.

Le MS et le FID sont mesurés après chaque round de communication. Le score moyen des GAN sur toutes les machines est indiqué dans la figure 4.2 pour chaque méthode concurrente. Afin de mieux comprendre les performances des GAN, nous traçons des données générées par ceux-ci à la fin de chaque exécution (c'est-à-dire, après $20\ 000$ d'itérations) sur la figure 4.3. Enfin, la figure 4.4 présente la distribution des scores obtenus par chaque GAN à la fin de l'apprentissage.

Résultats Nous observons d'abord que la méthode *standalone* présente des écarts de score plus importants dus à une non-collaboration des GAN au cours de l'apprentissage (Figure 4.4). Très clairement, l'apprentissage autonome souffre de la petite taille des bases de données locales B_i dans le cas où $n = 100$ et ne peut donc pas fournir de résultats d'apprentissage satisfaisants (Figure 4.3 pour $n = 100$). A partir de cette observation, la nécessité d'une approche collaborative est évidente pour des raisons de performance du GAN final.

Comme on pouvait s'y attendre, FL-GAN obtient les meilleurs scores MS et FID avec $n = 10$ et $n = 100$, qui sont similaires ou supérieurs aux résultats de l'exécution centralisée ($n = 1$). Ces performances sont permises au prix de l'envoi de tous les paramètres $\mathcal{G}$ et $\mathcal{D}$ de chaque machine à un serveur de paramètres pour les calculer en moyenne à chaque étape de la communication. Comme nous l'avons aperçu dans le chapitre précédent, les serveurs de paramètres peuvent être un point bloquant pour l'apprentissage. Ils nécessitent une bande passante importante afin de collecter tous les modèles de chacun des agents.

La troisième observation est que les méthodes d'apprentissage avec un protocole de rumeur pour les GAN peuvent aussi donner de bons résultats. En effet, les méthodes *Gossip* obtiennent des résultats proches de FL-GAN pour $n = 10$ et $n = 100$. Nous observons que la variante *Gossip_ind* GAN n'améliore que marginalement les résultats. Pour $n = 10$, *Gossip* GAN obtient des résultats similaires à FL-GAN pour la métrique FID et suffisamment proches pour la métrique MS. Pour $n = 100$, les résultats de *Gossip* GAN diminuent par rapport à FL-GAN, mais sont néanmoins bien meilleurs que ceux de la méthode *standalone*. Les écarts de scores pour les méthodes de rumeur, dans la figure 4.5, sont plus importants que pour le *Federated Learning* ; cela peut justifier une dernière étape d'agrégation à la fin des exécutions, où toutes les machines convergent de manière distribuée pour ne conserver que le meilleur couple $\mathcal{G}$ et $\mathcal{D}$. La figure 4.3 montre que les échantillons générés par les méthodes Gossip sont

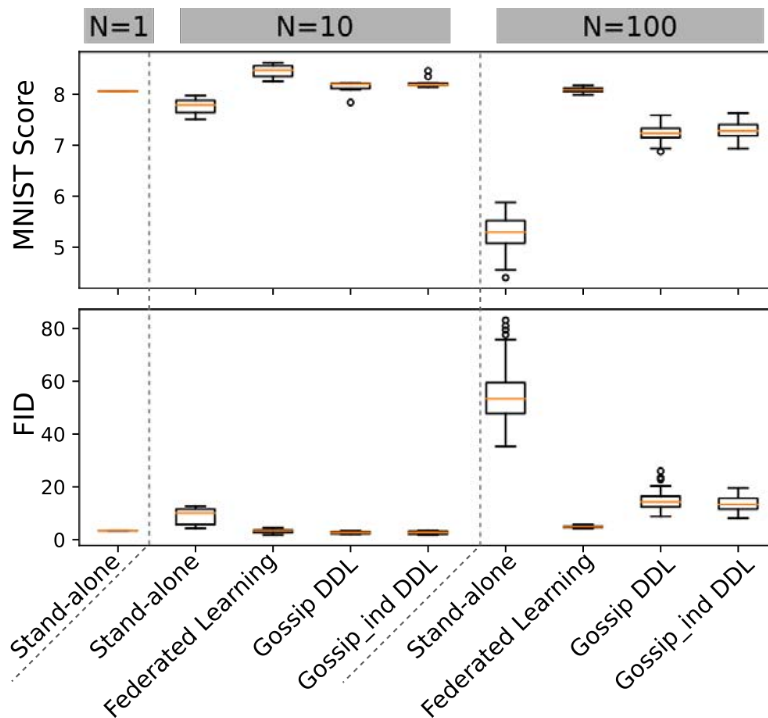


FIGURE 4.4 – Distribution des scores finaux des GAN (FID et MS) de chaque compétiteur, avec $n \in \{1, 10, 100\}$.

également réalistes pour la plupart d'entre eux.

Nous avons finalement mis en place un scénario dans lequel les données sur les machines ne sont pas i.i.d. : chacune des 10 machines héberge des images d'un même chiffre. Nous observons sur la figure 4.5 qu'aucun des concurrents ne parvient à atteindre un score FID décent, soulignant la non convergence des approches d'apprentissage. Ce dernier point montre une faiblesse de l'apprentissage des GAN par une méthode de moyenne de modèle. Entre chaque round de communication, chaque couple de GAN se sur-spécialise aux types de données en local sur leur machine. Ce phénomène empêche les GAN de converger vers un optimum.

4.3 Conclusion

Les conclusions de cette évaluation sont les suivantes : *(i)* le processus de rumeur des GAN obtient des performances d'apprentissage proches de celles du *Federated Learning*, avec l'avantage d'une absence de serveur central. Ce fait empirique n'a pas été rapporté dans [13] même pour l'utilisation du protocole de rumeur dans l'apprentis-

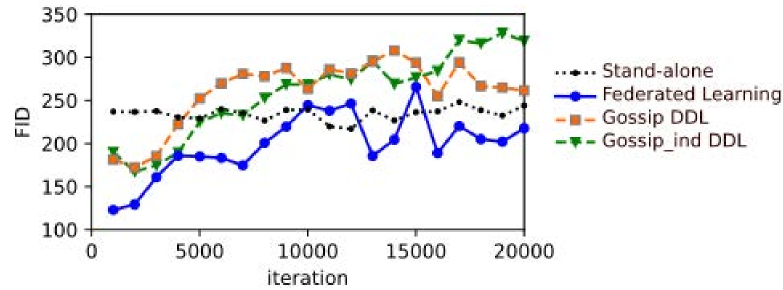


FIGURE 4.5 – FID de chaque compétiteur avec une base de données B_{train} non i.i.d. et $n = 10$ (chaque machine contient uniquement les images d'un seul chiffre).

sage d'un modèle réseau de neurones classique. Cette observation, tirée de la base de données MNIST, doit cependant être vérifiée sur d'autres jeux de données et d'architectures de GAN. *(ii)* pour les GAN, la distribution initiale des données (c'est-à-dire, i.i.d. ou non) est cruciale pour la convergence. *(iii)* l'intuition (dirigée par les travaux de [56]) que le changement d'adversaires entre $\mathcal{G}$ et $\mathcal{D}$ au cours de l'apprentissage, afin d'apporter plus de diversité, n'apporte que des résultats légèrement supérieurs à ceux de la méthode de base du protocole de rumeur *Gossip* GAN [13].

Ces résultats motivent l'améliorer les techniques d'apprentissage dans un contexte de système distribué dans le cas d'apprentissage de GAN. Nous présentons dans le prochain chapitre une solution pour répondre à cette problématique. Nous avons revu le concept des GAN [39] dans l'introduction. Les spécificités de la distribution efficace des GAN pour des bases de données distribuées restent un sujet de recherche ouvert, car le couplage étroit des générateurs et des discriminateurs doit être pris en compte. Le chapitre suivant propose une nouvelle méthode afin de distribuer l'apprentissage d'un GAN sans utiliser la technique de mise en commun par modèle moyen. Nous verrons que cette méthode permet d'atteindre des scores de performances plus intéressants que le *Federated Learning*.

CHAPITRE 5

MD-GAN : GAN AVEC DE MULTIPLES DISCRIMINATEURS POUR DES BASES DE DONNÉES DISTRIBUÉES

Les réseaux antagonistes génératifs (ou GAN) sont des modèles d'apprentissage profond prometteurs. Comme nous l'avons vu dans le chapitre précédent, ils ont de nombreuses applications et sont particulièrement intéressants dans notre contexte d'apprentissage collaboratif. En effet, ils peuvent être entraînés par un apprentissage non-supervisé, c'est-à-dire, sans nécessiter de labels sur les données d'apprentissage. Cela permet aux participants de ne pas avoir à annoter par eux-mêmes leurs données utilisées pour l'apprentissage. Dans ce chapitre, nous allons présenter une nouvelle contribution afin d'apprendre des GAN dans le contexte d'une base de données distribuée. Cette méthode, appelée MD-GAN (pour *Multi-Discriminator Generative Adversarial Network*) se base sur un modèle avec un serveur central. Contrairement à une descente de gradient asynchrone, comme avec notre méthode *AdaComp*, ce serveur central est utilisé pour réduire la charge de calcul sur les machines des participants en plus de mettre en commun l'apprentissage du modèle. Ce chapitre est organisé de la manière suivante :

— Dans la Section 5.1, nous présentons le contexte de système distribué dans lequel nous nous plaçons.
— La Section 5.2 présente en détail notre méthode MD-GAN.
— Nous expérimentons cette méthode dans la Section 5.3.
— Dans la Section 5.4, nous arborderons des travaux connexes, et en particulier ceux des GAN avec adversaires multiples utilisés dans des cas centralisé.
— Finalement, nous discutons des perspectives de cette méthode en Section 5.5.

5.1 Contexte distribué

Avant de présenter MD-GAN, nous présentons la configuration de système distribué, considérée dans ce chapitre.

Modèle du GAN Nous avons vu dans le chapitre précédent que l'apprentissage d'un GAN dans le contexte d'un apprentissage distribué pouvait être intéressant car il permettait d'apprendre la distribution des données d'une base d'apprentissage de manière non supervisée. De plus, avec l'utilisation de certains types de GAN [82], il est également possible de rajouter certaines connaissances (comme des labels) si elles sont disponibles. Nous reprenons la définition du GAN décrite dans la Section 4.1.1 avec les mêmes notations.

Apprentissage sur une base de données distribuée Nous considérons une configuration avec w machines contenant chacune un agent. Les w agents ont accès à une base de données locale composée de m données (chacune des données est de taille d) avec la même distribution de probabilité P_{data} (par exemple, des requêtes à un assistant vocal ou des photos de vacances). Ces bases de données locales restent en place au cours de tout le processus d'apprentissage (c'est-à-dire que leurs données ne seront pas envoyées sur le réseau). On note $B_{train} = \bigcup_{i=1}^{w} B_i$, l'ensemble de la base de données d'apprentissage, avec B_i l'ensemble des données locales de l'agent i.

Modèle avec un serveur central Malgré les progrès des techniques de calculs distribuées vers l'utilisation de méthodes décentralisées, utilisées également dans les *datacenters* (par exemple, l'utilisation du protocole de rumeur comme dans Dynamo [29] en 2007), l'apprentissage profond fait souvent exception avec une architecture centralisée (comme nous l'avons vu dans le Chapitre 2).

En effet, la quantité de données nécessaire pour former un réseau de neurones profond et la nature très itérative des tâches d'apprentissage obligent à opérer de manière parallèle, avec généralement l'utilisation d'un serveur central. Par exemple, nous avons vu dans le Chapitre 4 que l'utilisation de ce serveur central permettait à la méthode du *Federated Learning* d'obtenir de meilleurs résultats que la méthode décentralisée proposée. Pour ces raisons, nous supposons la possibilité d'utiliser un serveur central connecté à tous les participants dans notre configuration.

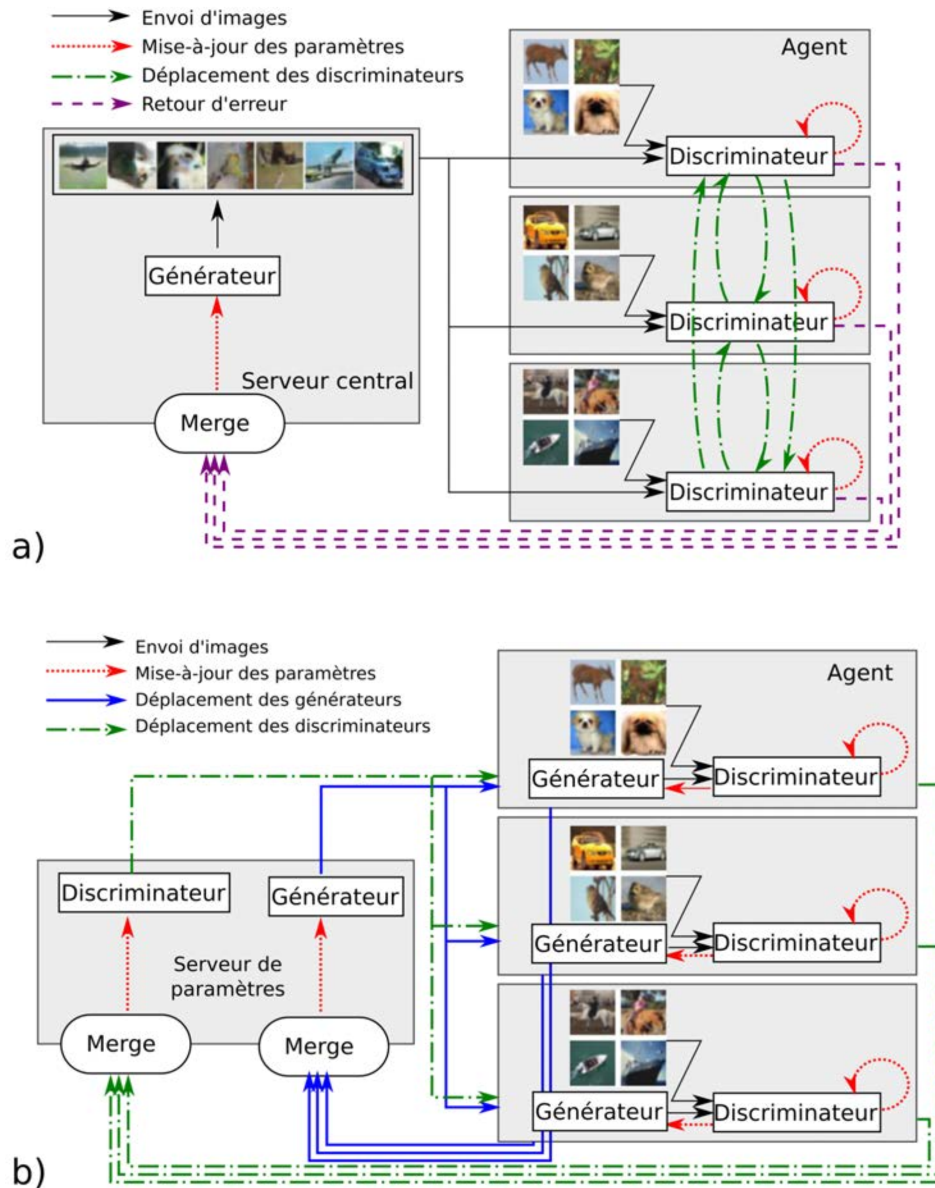


FIGURE 5.1 – Les deux concurrents proposés pour la distribution des GAN : a) Le modèle MD-GAN, comparé à b) FL-GAN (*Federated Learning* adapté aux GAN). MD-GAN utilise un seul générateur, placé sur le serveur central ; FL-GAN utilise des générateurs et des discriminateurs sur chaque agent. De plus, il utilise le serveur central pour mettre en commun l'apprentissage. Pour MD-GAN, seul l'apprentissage du générateur est centralisé avec la mise en commun des retours d'erreur. Dans cette exemple, seuls les agents actifs sont montrés pour MD-GAN et FL-GAN.

Le concurrent FL-GAN Dans la configuration décrite, il est possible d'entraîner une méthode de *Federated Learning* afin d'apprendre le GAN. Nous reprenons la méthode FL-GAN, vue dans la Section 4.1.2 du chapitre précédent afin de comparer les performances de MD-GAN avec celles du *Federated Learing*. Elle est illustrée sur la Figure 5.1.

Nous allons maintenant détailler notre contribution, appelée MD-GAN, qui permet l'apprentissage d'un GAN sur un ensemble d'agents contenant leurs propres données en local. Nous comparons ensuite MD-GAN à FL-GAN.

5.2 La méthode du MD-GAN

5.2.1 Raisonnement

Afin de réduire la charge de calcul des agents, nous proposons de faire l'apprentissage avec un unique générateur $\mathcal{G}$, situé sur le serveur central et de multiples discriminateurs, chacun situé sur un agent qui sera alors dans un état "actif" (de la même manière que pour FL-GAN). Cette solution a également l'avantage de tirer partie d'un affrontement entre adversaires multiples (le générateur contre n discriminateurs). Ce genre d'affrontement a montré de meilleures capacités du GAN dans le cas d'un apprentissage centralisé [56, 49, 31].

Le serveur central contient donc $\mathcal{G}$ avec ses paramètres $\boldsymbol{\lambda}$ tandis que la base de données d'apprentissage est distribuée sur les w agents. n discriminateurs $\mathcal{D}_1, \ldots, \mathcal{D}_n$ sont initialisés sur n agents choisis au hasard parmi les w agents. Chacun de ces discriminateurs est initialisé de manière indépendante (avec potentiellement des architectures différentes). Nous appellerons $\boldsymbol{\theta}_i$ les paramètres du discriminateur $\mathcal{D}_i$. Au cours de l'apprentissage, ces discriminateurs vont se déplacer entre les différents agents. Cette architecture est présentée dans la Figure 5.1 a).

Le but de l'apprentissage consiste donc à apprendre le générateur $\mathcal{G}$ (situé sur le serveur central) en utilisant la base de données distribuée $B_{train} = \bigcup_{i=1}^{w} B_i$. Pour ce faire, le générateur $\mathcal{G}$ doit s'entraîner contre les n discriminateurs, c'est-à-dire, essayer de générer des données qui peuvent passer pour réelles pour chacun des $\mathcal{D}_1, \ldots, \mathcal{D}_n$. Les discriminateurs, quant à eux, doivent apprendre à différencier les données générées par $\mathcal{G}$ des données réelles à l'aide des données de $B_1, \ldots, B_w$ situées sur les agents. L'apprentissage du GAN est un processus itératif ; dans MD-GAN, une *itération*

globale est composée de 4 étapes différentes :

— Le serveur central génère un ensemble K de κ batches $K = \{X^{(1)}, \ldots, X^{(\kappa)}\}$, avec $\kappa \leq n$. Chaque batch est composé de b données générées par $\mathcal{G}$. Le serveur se charge de sélectionner deux batches distincts pour chaque discriminateur j, qu'il envoie à l'agent actif correspondant. Ces batches sont renommés localement $X_j^{(g)}$ et $X_j^{(d)}$. La manière de sélectionner ces deux batches pour chaque discriminateur est discutée dans la Section 5.2.2.

— Chaque agent actif i, qui contient un discriminateur $\mathcal{D}_j$, effectue L itérations d'apprentissage de $\mathcal{D}_j$ en utilisant le batch reçu $X_j^{(d)}$ ainsi qu'un nouveau batch constitué localement de b données provenant de B_i, nommé $X_j^{(r)}$.

— Chaque agent actif calcule ensuite un retour sur l'erreur produite par le générateur sur les données $X_j^{(g)}$ par rapport au discriminateur $\mathcal{D}_j$. Ce retour d'erreur est noté F_j. Nous détaillons le calcul de F_j dans la Section 5.2.2. Une fois le calcul effectué, ce retour F_j est envoyé au serveur central.

— Le serveur central calcule le gradient de sa fonction de coût J_{gen} pour ses paramètres $\boldsymbol{\lambda}$ à partir des retours d'erreur $F_1, \ldots, F_n$. Il peut ensuite mettre à jour les paramètres de $\mathcal{G}$ avec ce gradient à l'aide d'un optimiseur tel que Adam [58].

De plus, après chaque période de E époques, les discriminateurs se déplacent d'agent en agent en utilisant la fonction MOVE(). Le pseudo-code de MD-GAN, incluant ces étapes, est présenté dans les Algorithmes 3 et 4. L'ensemble des notations est également indiqué sur la Table 5.1.

Nous pouvons noter que des nouveaux agents peuvent entrer dans ce processus d'apprentissage sans aucune perturbation. Ces agents doivent juste se faire connaître de leurs pairs et du serveur central pour pouvoir accueillir un discriminateur. Cet ajout d'agents permet donc d'augmenter les données d'apprentissage et la limite du nombre de discriminateurs. Si le nombre de discriminateurs n'est pas suffisant, l'ajout d'un nouveau discriminateur peut se faire sur un agent inactif. Afin de ne pas perturber l'apprentissage, il est important que ce discriminateur soit au même niveau que les autres. La méthode la plus simple consiste donc à dupliquer un discriminateur déjà présent sur un agent actif.

Ce chapitre présente l'entraînement d'un GAN qui a pour but de générer des images, avec la méthode MD-GAN. Cette exemple n'implique aucune perte de généralité pour d'autres applications.

Notation	Signification
$\mathcal{G}$	Générateur
$\mathcal{D}$	Discriminateur
n	Nombre de discriminateurs
w	Nombre d'agents
C	Server central
W_i	Agent i
P_{data}	Distribution des données réelles
$P_{\mathcal{G}}$	Distribution des données générées par $\mathcal{G}$
$\boldsymbol{\lambda}$ (resp. $\boldsymbol{\theta}$)	Paramètres de $\mathcal{G}$ (resp. $\mathcal{D}$)
λ_k (resp. θ_k)	k-ième paramètres de $\mathcal{G}$ (resp. $\mathcal{D}$)
B_{train}	Base de données d'apprentissage
B_i	Base de données d'apprentissage locale à l'agent i
m	Nombre d'objets dans une base de données locale B_i
d	Taille d'une donnée (Par exemple, taille d'une image en Mb)
b	Taille des batches
I	Nombre d'itérations globales lors d'un apprentissage
K	L'ensemble de tous les batches $X^{(1)}, \ldots, X^{(\kappa)}$ générés par $\mathcal{G}$ lors d'une itération
F_j	Le retour de l'erreur calculé par l'agent actif contenant le discriminateur j
E	Nombre d'époques locales avant un déplacement des discriminateurs

TABLE 5.1 – Table des notations pour MD-GAN

5.2.2 Procédure d'apprentissage du générateur

Le serveur contient le générateur $\mathcal{G}$ avec les paramètres associés $\boldsymbol{\lambda}$. Le serveur central envoie des images générées par $\mathcal{G}$ aux agents. Ceux-ci utilisent ces données pour entraîner leurs discriminateurs et envoyer un retour au générateur. Une fois l'apprentissage terminé, le GAN appris sur le serveur central doit être capable de générer des images comme celles présentes sur les agents.

Distribution des batches générés

À chaque itération globale, $\mathcal{G}$ génère un ensemble de κ batches $K = \{X^{(1)}, \ldots, X^{(\kappa)}\}$ (avec $\kappa \leq n$) de taille b. Chaque agent contenant un discriminateur j reçoit 2 des batches, parmi K, nommés $X_j^{(g)}$ et $X_j^{(d)}$. Le choix d'envoyer deux batches à chaque discriminateur est nécessaire pour pouvoir calculer le gradient pour $\mathcal{D}$ et $\mathcal{G}$ sur des

données distinctes (comme dans le cas du GAN classique [39]). Une solution proposée pour distribuer les $X^{(a)}$ (avec $a = 1, \ldots, \kappa$) aux n différents agents actifs peut être de fixer $X_j^{(g)} = X^{((j \mod \kappa)+1)}$ et $X_j^{(d)} = X^{(((j+1) \mod \kappa)+1)}$ pour $j = 1, \ldots, n$.

Mise à jour des paramètres du générateur

À chaque itération globale, le serveur reçoit le retour d'erreur F_j de chaque discriminateur j, ce qui correspond à l'erreur faite par $\mathcal{G}$ sur $X_j^{(g)}$. Plus formellement, F_j est composé de b vecteurs $\{\mathbf{e}_{j_1}, \ldots, \mathbf{e}_{j_b}\}$, où $\mathbf{e}_{j_q}$ est donné par

$$\mathbf{e}_{j_q} = \frac{\partial \widetilde{B}(X_j^{(g)})}{\partial \boldsymbol{x}_q},$$

avec $\boldsymbol{x}_q$ la q-ième donnée du batch $X_j^{(g)}$. Le gradient $\Delta\boldsymbol{\lambda} = \partial \widetilde{B} \left(\cup_{j=1}^{n} X_j^{(g)} \right) / \partial \boldsymbol{\lambda}$ est calculé à partir de tous les F_j tel que :

$$\Delta\lambda_k = \frac{1}{nb} \sum_{j=1}^{n} \sum_{\boldsymbol{x}_q \in X_j^{(g)}} \mathbf{e}_{j_q} \frac{\partial \boldsymbol{x}_q}{\partial \lambda_k},$$

avec $\Delta\lambda_k$ le k-ième élément de $\Delta\boldsymbol{\lambda}$. Le terme $\partial \boldsymbol{x}_j / \partial \lambda_j$ est calculé sur le serveur. Notons que $\cup_{n=1}^{N} X_n^{(g)} = \{\mathcal{G}_\lambda(\boldsymbol{z}) | \boldsymbol{z} \in Z_g\}$. Minimiser $\widetilde{B} \left(\cup_{j=1}^{n} X_j^{(g)} \right)$ revient donc à minimiser $J_{gen}(Z_g)$ (voir Section 4.1.1). Une fois les gradients calculés, le serveur peut mettre à jour ses paramètres $\boldsymbol{\lambda}$. En utilisant l'optimiseur Adam [58], le paramètre $\lambda_k \in \boldsymbol{\lambda}$ à l'itération t, indiqué par $\lambda_k(t)$ ici, est égale à

$$\lambda_k(t) = \lambda_k(t-1) + \text{Adam}(\Delta\lambda_k),$$

où la fonction Adam représente l'optimiseur qui calcule la mise à jour en fonction du gradient $\Delta\lambda_k$.

Charge de calcul du serveur

Le fait de placer le générateur sur le serveur augmente sa charge de travail. Il doit générer et envoyer k batches de b données avec $\mathcal{G}$ lors de la première étape d'une itération globale, puis recevoir n retours d'erreur de taille bd à la troisième étape. Il les utilise ensuite pour calculer le gradient de $\mathcal{G}$. La génération de batches nécessite kbG_{op} opérations en virgule flottante (où G_{op} est le nombre d'opérations flottantes permettant

de générer une donnée avec $\mathcal{G}$) et une mémoire de kbG_a (avec G_a la mémoire utilisée pour le calcul d'une donnée avec $\mathcal{G}$). Pour simplifier, supposons que $G_{op} = O(|\boldsymbol{\lambda}|)$ et que $G_a = O(|\boldsymbol{\lambda}|)$. Par conséquent, la complexité de la génération de batches est $O(kb|\boldsymbol{\lambda}|)$. L'opération de fusion de tous les retours F_n et les calculs de gradient impliquent une complexité de mémoire et de calcul de $O(b(dn + k|\boldsymbol{\lambda}|))$.

Compromis entre complexité et diversité des données

À chaque itération globale, le serveur central génère k batches, avec $k < n$. Si $k = 1$, tous les agents reçoivent et calculent leur retour d'erreur F_i sur le même batch d'entraînement. Cela réduit la diversité des commentaires reçus par le générateur, mais également la charge de travail du serveur. Si $k = n$, chaque agent calcule un retour d'erreur sur un batch différent, ainsi les retours d'erreur seront plus diversifiés. La charge de calcul du serveur central se trouve cependant trop élevée. Par conséquent, nous choisissons comme compromis $k = \lfloor \log(n) \rfloor$. Dans nos expériences, nous évaluons $k = 1$ er $k = \lfloor \log(n) \rfloor$ pour mesurer l'impact de ces valeurs sur les performances du modèle final.

5.2.3 Procédure d'apprentissage des discriminateurs

Dans notre configuration, nous supposons un ensemble de n discriminateurs répartis sur w agents, de telle sorte qu'il ne puisse y avoir deux discriminateurs sur un agent (ce qui implique que $w \geq n$). Notons que si $n = w$, chaque agent héberge un discriminateur.

L'ensemble des agents (actifs ou non) contient la base de données d'apprentissage $B_{train} = \bigcup_{i=1}^{w} B_i$ avec B_i la base de données placée sur l'agent i. Les agents actifs, c'est-à-dire, qui contiennent un discriminateur j, reçoivent des batches d'images générées, divisées en deux parties : $X_j^{(d)}$ et $X_j^{(g)}$. Notons que l'indice d'un agent est notée i alors que l'indice d'un discriminateur est noté j.

Les images générées $X_j^{(d)}$ sont utilisées pour l'apprentissage de $\mathcal{D}_j$ afin de distinguer les images générées des images réelles. Cet apprentissage est effectué comme dans le cas classique d'un système centralisé [39]. Chaque agent actif calcule le gradient $\Delta\boldsymbol{\theta}_j$ de la fonction d'erreur J_{disc} appliquée au batch reçu $X_j^{(d)}$, et à un batch d'images réelles $X_j^{(r)}$ provenant de $\mathcal{B}_i$. Comme indiqué dans la Section 4.1.1, cette opération est itérée L fois.

Le deuxième batch $X_j^{(g)}$ d'images générées est utilisé pour calculer le terme d'erreur F_j du générateur $\mathcal{G}$. Une fois calculé, F_j est envoyé au serveur pour le calcul des gradients $\Delta\boldsymbol{\lambda}$.

Le déplacement des discriminateurs pair-à-pair

Chaque discriminateur j placé sur un agent actif i utilise uniquement B_i pour entraîner ses paramètres $\boldsymbol{\theta}_j$. Si trop d'itérations sont effectuées sur la même base de données local, le discriminateur a tendance à se spécialiser (ce qui diminue sa capacité de généralisation). Cet effet de sur-apprentissage (voir Section 1.2.4), est évité dans MD-GAN en déplaçant les n discriminateurs avec leurs paramètres $\boldsymbol{\theta}_j$ (avec $j = 1, \ldots, n$) entre les w agents après E époques effectuées en local. Lorsque $w > n$, ce déplacement permet de varier les agents actifs et ainsi utiliser une plus grande variété de données. Le déplacement est mis en œuvre par communication pair-à-pair, en choisissant de manière aléatoire le nouveau destinataire parmi les agents pour chaque discriminateur.

Le principe d'agents actifs est repris du système du *Federated Learning* pour être capable de faire face à des bases de données largement distribuées, sans pour autant avoir un trop grand nombre de modèles à entraîner en parallèle. Cela évite ainsi les risques de divergence de l'apprentissage et une surconsommation inutile des ressources des utilisateurs.

Charge de calcul au niveau des agents

L'objectif de MD-GAN est de réduire la charge de calcul des agents sans déplacer les données locales hors de leur emplacement initial. Par rapport à la méthode du FL-GAN, la charge du générateur est déportée sur le serveur. Les agents actifs doivent uniquement gérer les paramètres de leur discriminateur θ_j et calculer les retours d'erreur après L itérations d'apprentissage du discriminateur effectuées en local. À chaque itération globale, un agent effectue $2LbD_{op}$ opérations en virgule flottante (où D_{op} correspond au nombre d'opérations en virgule flottante pour le calcul d'un gradient de $\mathcal{D}$ pour une donnée (générée ou non). La mémoire utilisée par un agent est de l'ordre de $O(|\boldsymbol{\theta}|)$.

	FL-GAN	MD-GAN
Calculs C	$O(Ibn(\|\boldsymbol{\lambda}\| + \|\boldsymbol{\theta}\|)/(mE))$	$O(Ib(dn + k\|\boldsymbol{\lambda}\|))$
Mémoire C	$O(n(\|\boldsymbol{\lambda}\| + \|\boldsymbol{\theta}\|))$	$O(b(dn + k\|\boldsymbol{\lambda}\|)$
Calculs W	$O(Ib(\|\boldsymbol{\lambda}\| + \|\boldsymbol{\theta}\|))$	$O(Ib\|\boldsymbol{\theta}\|))$
Mémoire W	$O(\|\boldsymbol{\lambda}\| + \|\boldsymbol{\theta}\|)$	$O(\|\boldsymbol{\theta}\|)$

TABLE 5.2 – Complexité en calculs et en mémoire pour le MD-GAN et le *Federated Learning* adapté au GAN au niveau des agents (W) et du serveur (C).

5.2.4 Caractéristiques de MD-GAN

Complexité de la communication

L'algorithme MD-GAN comprend trois types de communications :

— Communications du serveur à agent : au début de chaque itération globale, le serveur envoie ses batches de k images générées aux agents. Le nombre d'images générées est kb (avec $k \leq n$), mais seuls deux batches sont envoyés à chaque agent. La communication totale du serveur est donc $2bdn$ (c'est-à-dire, $2bd$ par agent).

— Communications des agents au serveur : après avoir calculé l'erreur du générateur sur $X_j^{(g)}$, tous les agents actifs envoient leur retour d'erreur F_j au serveur. La taille du retour d'erreur est de bd par agent actif, car un seul flottant est requis pour chaque élément d'une donnée.

— Communications entre agents : après les E époques locales, les paramètres de chaque discriminateur sont déplacés d'un agent actuellement actif à un nouvel agent (précédemment actif ou non). En terme de communication, chaque agent actif envoie un message de taille $|\boldsymbol{\theta}_j|$ à un autre agent. Donc n agents pris aléatoirement reçoivent un message de cette même taille (pour des raisons de simplicité, nous supposons que les modèles discriminants sur les agents ont la même architecture).

Les complexités de la communication sont résumées dans le tableau 5.3, pour MD-GAN et FL-GAN. Le tableau 5.4 présente un exemple de ces complexités avec les quantités réelles de données estimées pour l'expérience sur la base de données CIFAR10. La première observation est que MD-GAN requiert une communication serveur-agent à chaque itération, tandis que FL-GAN effectue des communications toutes les mE/b itérations (c'est-à-dire, pendant une période de E époques). Notons que la taille des communications agents-serveur dépend des paramètres du GAN ($\boldsymbol{\theta}$ et $\boldsymbol{\lambda}$) pour FL-

GAN, alors qu'elle dépend de la taille d des données et de b pour MD-GAN. Il est donc particulièrement intéressant de choisir une petite taille de batches. Ceci, d'autant plus que Gupta *et al.* [41] ont montré que, pour espérer de bonnes performances sur un apprentissage parallèle d'un modèle (tel que le générateur dans MD-GAN), b doit être inversement proportionnel au nombre d'agents n. Cependant, lorsque la taille des données est grande (comme dans les applications d'images avec de grandes résolutions), les communications MD-GAN peuvent devenir coûteuses.

Type de communication	FL-GAN	MD-GAN
C→W (C)	$n(\boldsymbol{\theta} + \boldsymbol{\lambda})$	bdn
C→W (W)	$\boldsymbol{\theta} + \boldsymbol{\lambda}$	bd
W→C (W)	$\boldsymbol{\theta} + \boldsymbol{\lambda}$	bd
W→C (C)	$n(\boldsymbol{\theta} + \boldsymbol{\lambda})$	bdn
# C↔D total	$Ib/(mE)$	I
W→W (W)	-	$\boldsymbol{\theta}$
# W↔W total	-	$Ib/(mE)$

TABLE 5.3 – Complexité de communication pour MD-GAN et FL-GAN. C et W représentent respectivement le serveur central un agent.

Type de communication	FL-GAN $b = 10$	FL-GAN $b = 100$	MD-GAN $b = 10$	MD-GAN $b = 100$
C→W (C)	175 Mo	175 Mo	2.30 Mo	23.0 Mo
C→W (W)	17.5 Mo	17.5 Mo	0.23 Mo	2.30 Mo
W→C (W)	17.5 Mo	17.5 Mo	0.23 Mo	2.30 Mo
W→C (C)	175 Mo	175 Mo	2.30 Mo	23.0 Mo
Total # C↔W	100	1,000	50,000	50,000
W→W (W)	-	-	6.34 Mo	6.34 Mo
Total # W↔W	-	-	100	1,000

TABLE 5.4 – Exemple du coût de communication avec la base de données CIFAR10 (et le modèle associé dans les expériences) avec $n = w = 10$.

Nous avons tracé sur la Figure 5.2 une estimation du trafic maximal en entrée d'un agent et en entrée du serveur central (axe des y) pour FL-GAN et MD-GAN, pour une seule itération, en fonction de la taille de batch choisie (axe x). Cela correspond pour FL-GAN à une communication agents-serveur. Pour MD-GAN, cela correspond à une communication agents-serveur et agents-agents au sein d'une même itération. Les

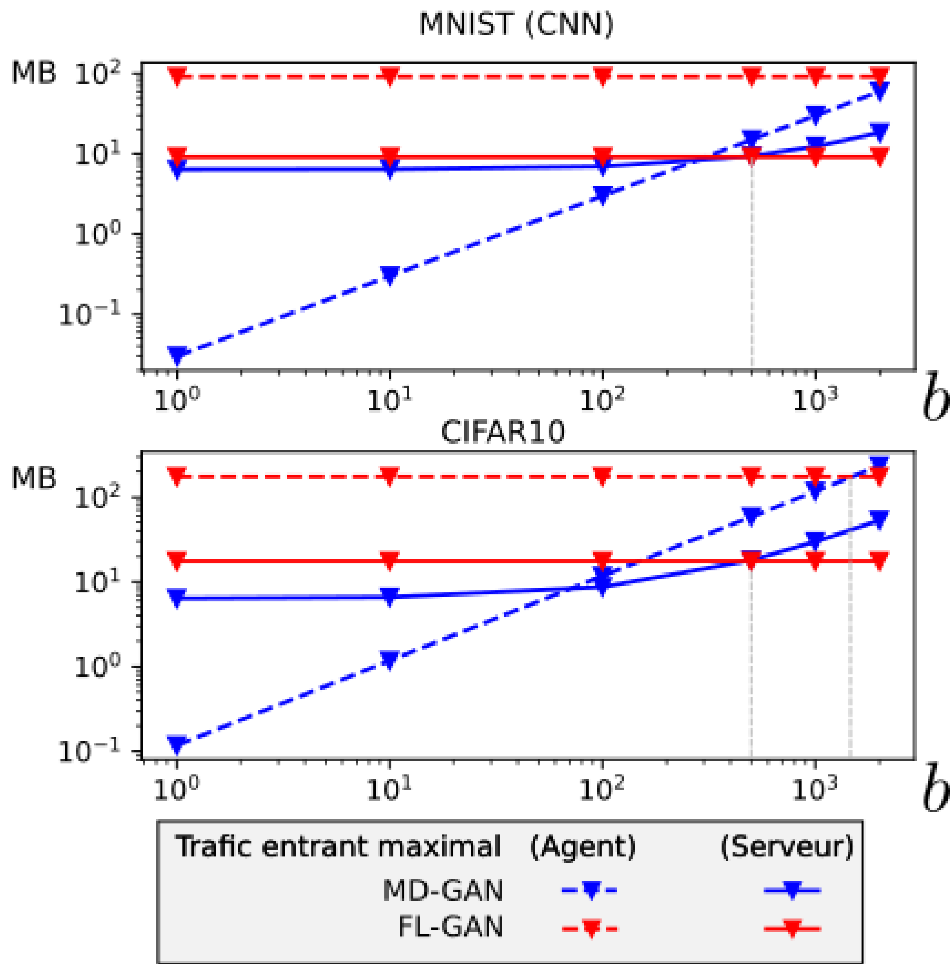


FIGURE 5.2 – Estimation de la borne supérieure du trafic entrant sur le serveur central (en trait plein) et pour chaque agent (en pointillé) pour MD-GAN et FL-GAN en fonction de la taille de batch b.

courbes pleines décrivent le trafic entrant sur les agents, tandis que les courbes pointillées représentent le trafic sur le serveur ; ces quantités peuvent aider à dimensionner les capacités du réseau nécessaires au processus d'apprentissage. Notons que les figures ont une échelle logarithmique sur les deux axes.

Comme prévu, le trafic FL-GAN est constant, car les communications ne dépendent que de la taille du GAN. Ce trafic indique une limite supérieure après laquelle le MD-GAN est donc plus coûteux. MD-GAN est donc compétitif pour des tailles de batch réduites, de l'ordre d'une centaines d'images dans les deux exemples (environ $b < 550$ pour MNIST et $b < 400$ pour CIFAR10).

Complexité du calcul

L'objectif de MD-GAN est de supprimer les tâches liées au générateur sur les agents en n'en conservant qu'un seul sur le serveur central. Pendant la phase d'apprentissage MD-GAN, le trafic entre les agents et le serveur est raisonnable (Table 5.3). Le gain en complexité pour les agents en terme de mémoire et de calcul dépend de l'architecture de $\mathcal{D}$; cela représente en général la moitié de la complexité totale, du fait que les complexités de $\mathcal{G}$ et $\mathcal{D}$ sont souvent similaires. La conséquence de cet algorithme basé sur un seul générateur est une fréquence plus grande d'interactions entre les agents et le serveur central, et la création d'un trafic pair-à-pair entre les agents. Les complexités globales des opérations sont résumées et comparées dans la Table 5.2, pour MD-GAN et FL-GAN ; la Table indique une charge moitié moins importante que FL-GAN pour les agents.

5.3 Évaluation expérimentale

Nous allons maintenant analyser empiriquement la convergence de MD-GAN et ses approches concurrentes.

5.3.1 Configuration expérimentale

Nos expérimentations utilisent le *framework* Keras basé sur le *backend* Tensorflow. Nous simulons les agents et le serveur central sur des serveurs équipés de deux processeurs Intel Xeon Gold 6132, 260 GB de RAM et quatre GPUs NVIDIA Tesla M60

ou quatre GPUs NVIDIA Tesla P100. Cette configuration permet l'apprentissage d'un GAN qui serait identique à celui appris dans un cas réel de système distribué. En effet, l'ordre d'exécution de l'algorithme 5.2 est préservé. Le choix de cette configuration a été fait afin de faciliter le déploiement de l'expérience.

Base de données d'apprentissage Nous expérimentons des approches concurrentes sur deux bases de données classiques pour l'apprentissage profond : MNIST, la base de données de chiffres écrits à la main, et CIFAR10, une base de données de petites images. MNIST est composée de $60\ 000$ images noir et blanc en 28x28 pixels, représentant les chiffres manuscrits, et un ensemble de test de $10\ 000$ images. CIFAR10 est composée d'un ensemble d'apprentissage de $50\ 000$ images RGB en 32x32 pixels, représentant les 10 classes suivantes : avion, automobile, oiseau, chat, cerf, chien, grenouille, cheval, bateau, camion. La base de test de CIFAR10 comporte également $10\ 000$ images non présentes sur la base de données d'apprentissage.

Architectures des GAN Dans nos expérimentations, nous avons entraîné un type de GAN appelé ACGAN [82]. Nous avons expérimenté trois architectures différentes de $\mathcal{G}$ et $\mathcal{D}$: une architecture basée sur des couches toutes connectées (MLP), une architecture basée sur des réseaux de neurones à convolution (CNN) pour MNIST et autre une architecture basée sur des CNN pour CIFAR10. Leurs caractéristiques sont les suivantes :

— dans l'architecture MLP pour MNIST, $\mathcal{G}$ et $\mathcal{D}$ sont composées de trois couches toutes-connectées. Les couches de $\mathcal{G}$ contiennent respectivement 512, 512 et 784 neurones, et les couches de $\mathcal{D}$ contiennent 512, 512 et 11 neurones. Le nombre total de paramètres est de $716\ 560$ pour $\mathcal{G}$ and $670\ 219$ pour $\mathcal{D}$.

— dans l'architecture CNN pour MNIST, $\mathcal{G}$ est composée d'une couche toute connectée de $6\ 272$ neurones et de deux couches de déconvolution de respectivement 32 et 1 filtres de dimensions 5×5. $\mathcal{D}$ est composée de six couches de convolution de respectivement 16, 32, 64, 128, 256 et 512 filtres de dimensions 3×3, une couche de discrimination de mini-batch reprise des travaux de Salimans *et al.* [93], et une couche totalement interconnectée de 11 neurones. Le nombre total de paramètres est de $628\ 058$ pour $\mathcal{G}$ et $286\ 048$ pour $\mathcal{D}$

— dans l'architecture basée CNN pour CIFAR10, $\mathcal{G}$ est composée d'une couche totalement interconnectée de 6144 neurones et de trois couches de "déconvo-

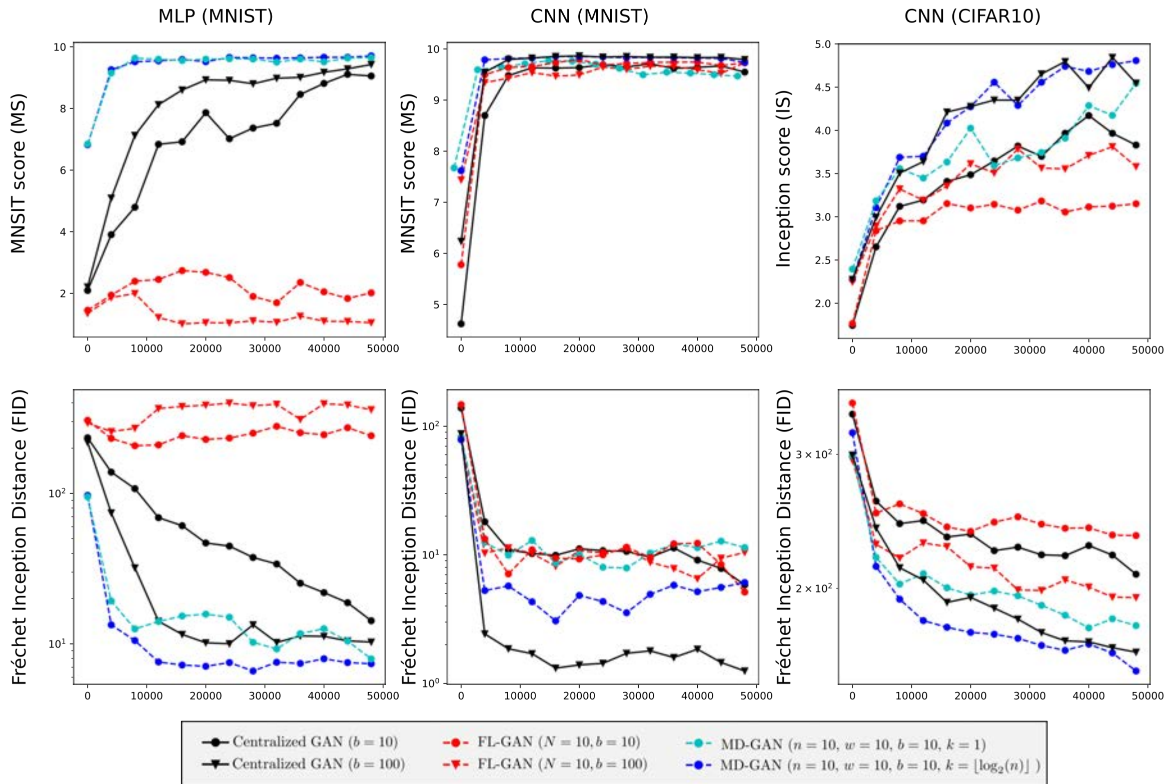


FIGURE 5.3 – Scores MS (ou IS pour CIFAR10) et FID (axes y) obtenus pour les trois compétiteurs en fonction du nombre d'itérations (axes x).

lution" de respectivement de 192, 96, et 3 noyaux de dimensions 5×5. $\mathcal{D}$ est composée de six couches de convolution de respectivement 16, 32, 64, 128, 256 et 512 noyaux de dimensions 3×3, une couche de discrimination de mini-batch (voir les travaux de [93]) et une couche toute-connectée de 11 neurones. Le nombre total de paramètres est de $628\ 110$ pour $\mathcal{G}$ et $100\ 203$ for $\mathcal{D}$.

Mesures des performances De même que dans le chapitre précédent, nous utilisons les mesures du score d'Inception (noté IS pour *Inception Score*) proposée par Salimans *et al.* [93] et de la distance Inception de Fréchet proposée dans les travaux de Heusel *et al.* [47]. Dans le cas de la base de données MNIST, nous utilisons également un réseau de neurones adapté à la classification des données de MNIST pour le calcul des scores. Le score d'Inception est alors renommé score de MNIST (MS). Pour rappel, le score d'Inception représente le niveau de réalisme des images générées tandis que la distance Inception de Fréchet représente une distance entre les données

générées et les données de la base de données de test.

Configurations de MD-GAN et des compétiteurs Pour comparer le MD-GAN au GAN centralisé, nous entraînons la même architecture de GAN sur un serveur unique (qui a accès à l'ensemble des données B_{train}). Nous nommons cette méthode *centralized-GAN*. Elle est testée avec des batches de dimensions $b = 10$ et $b = 100$. Nous exécutons FL-GAN avec pour paramètres $E = 1$ et $b = 10$ ou $b = 100$. MD-GAN s'exécute également avec $E = 1$ afin de comparer notre méthode avec FL-GAN. Pour MD-GAN et FL-GAN, la base de données d'apprentissage est partagée entre les agents de manière i.i.d. Nous travaillons sur deux configurations de MD-GAN, dont une avec $k = 1$ et l'autre avec $k = [\log(n)]$, afin d'évaluer l'impact de la diversité des données envoyées aux agents. La configuration $k = n$ n'est pas évaluée car nous l'estimons trop coûteuse pour le serveur central et donc pas réaliste. Toutes les expérimentations sont réalisées avec $I = 50000$, c'est-à-dire, que le générateur (ou les n générateurs pour FL-GAN) a été soumis à $50\ 000$ itérations d'apprentissage. Nous calculons les scores FID, MS et IS toutes les $1\ 000$ itérations en utilisant un échantillon de 500 données générées. Le FID est calculé en utilisant la même taille d'échantillon de 500 données, sélectionnées de manière aléatoire dans B_{test}. Pour FL-GAN et MD-GAN, les scores sont calculés à partir du générateur sur le serveur central.

5.3.2 Résultats des expériences

Nous rapportons ici les résultats pour toutes les configurations concurrentes, en fonction du nombre d'itérations sur la Figure 5.3. Les courbes sont lissées pour plus de lisibilité.

Comparaison des différents compétiteurs

La version *centralized-GAN* obtient de meilleurs résultats avec $b = 100$ qu'avec $b = 10$. Cela s'explique du fait que le GAN voit plus de données (réelles et générées) par itération lorsque b augmente. Lorsque $b = 10$ pour MD-GAN, le nombre total de données réelles vues de B_{train} est 100 avec $n = 10$. Ceci explique pourquoi MD-GAN obtient des scores très similaires à la version GAN autonome avec $b = 100$ (à l'exception de CNN sur MNIST). Nous notons que, comme indiqué dans la discussion de la

Section 5.2.2, l'hyper-paramètre k a un impact significatif sur le processus d'apprentissage. Plus les données envoyées par le serveur aux agents sont diversifiées, plus hauts sont les scores du générateur. Pour les expérimentations sur MLP, FL-GAN ne converge pas, tandis que MD-GAN a de meilleurs scores (FID et MS) que la version *centralized-GAN*. Ceci peut s'expliquer du fait que, avec MD-GAN, nous proposons un jeu avec des "discriminateurs multiples" contre un "générateur unique" ; des travaux récents [31] ont montré que l'apprentissage, dans un cas centralisé, basé sur un générateur et de multiple discriminateurs, ou un mixte de générateurs et un discriminateur [49], peuvent dépasser les performances du GAN classique. La compétition multiple est donc bénéfique pour l'apprentissage. Dans l'expérimentation du CNN sur MNIST, les scores FID et MS obtenus par MD-GAN et FL-GAN sont presque équivalents. Dans l'expérimentation CNN sur CIFAR10, MD-GAN obtient un meilleur IS et MS que FL-GAN sur cette tâche d'apprentissage plus complexe. Ces trois expérimentations montrent que MD-GAN exploite l'avantage d'avoir un seul générateur à entraîner face à plusieurs discriminateurs.

Mise à l'échelle et impact des communications pair-à-pair des agents

Nous présentons en Figure 5.4, l'évolution des scores pour MD-GAN (après $20\ 000$ itérations) en fonction du nombre d'agents (tous actifs, c'est-à-dire, $w = n$) en utilisant le modèle MLP. Il faut noter que lorsque le nombre d'agents w augmente, la taille des base de données locales diminue (car $|B_i| = |B_{train}|/w$). Deux variantes de MD-GAN sont exécutées. La première est la version de l'algorithme MD-GAN discutée, et la seconde est représentée par les courbes MD-GAN en pointillées, pour lesquelles il n'y a pas de permutations entre les agents. Les courbes bleues représentent les scores de MD-GAN lorsque la charge sur les agents (c'est-à-dire, le nombre d'images à traiter par itération) reste constante, tandis que les courbes oranges représentent une charge constante sur le nœud central. La Figure 5.4 illustre aussi une variation des tailles de batches b utilisées par les agents sur les courbes correspondantes à une charge constante sur le serveur central : plus grande est la valeur de n, plus faible est la valeur de b, pour conserver la même charge sur le serveur. Nous voyons que les phénomènes intéressants apparaissent après $n = 10$; pour des valeurs plus faibles de n, les agents semblent avoir assez de données localement pour atteindre des scores satisfaisants. Nous observons qu'en considérant une charge constante sur les agents, on améliore les résultats. Mais cela s'obtient au prix d'un coût plus élevé sur le serveur

(cf Table 5.3 et 5.2). Nous observons également que le processus de déplacement des discriminateurs améliore les résultats. Cependant, cette amélioration reste marginale pour la FID avec une charge constante sur le serveur.

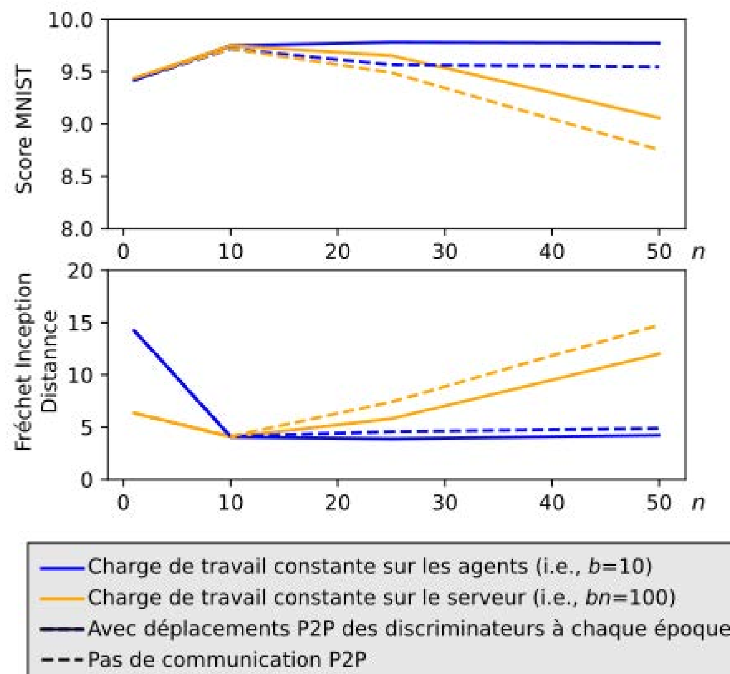


FIGURE 5.4 – Les résultats du score MNIST et de la distance Inception de Fréchet en fonction du nombre de discriminateurs (et d'agents car $w = n$) pour MD-GAN sur le modèle MLP. Ces expériences comparent également l'impact des déplacements entre agents.

Afin de mesurer l'impact de la distribution des données, nous réalisons l'apprentissage du MLP avec MD-GAN en variant le nombre d'agents $w \in \{10, 50, 100, 200, 500\}$ et le paramètre $E \in [1, 10, 50]$. Le nombre de discriminateurs reste cependant constant à $n = 10$. Le résultat de ces expériences est reporté sur la Figure 5.5 avec les scores de MS et FID atteints à la fin de l'apprentissage.

Nous observons que les résultats du modèle se dégradent lorsque E augmente. La dégradation est d'autant plus flagrante que le nombre d'agents est grand. Ceci peut s'expliquer par le fait que les 10 discriminateurs passent en phase de sur-apprentissage lorsqu'ils restent trop longtemps sur le même agent (c'est-à-dire, E est grand). Cette phase de sur-apprentissage arrive d'autant plus rapidement que le nombre d'agent est grand (car la taille des B_i diminue). Cependant, nous pouvons observer que, en

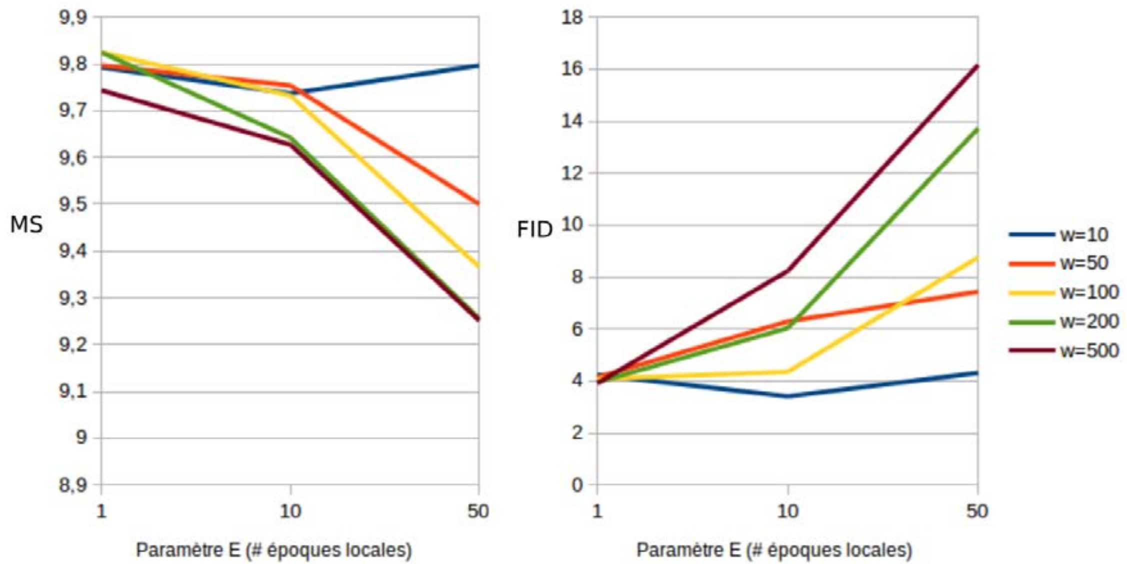


FIGURE 5.5 – Évolution de MS et FID en fonction du nombre d'agents et du paramètre E avec le modèle MLP et $n = 10$

terme de capacité de mise à l'échelle, MD-GAN obtient des scores similaires quel que soit la valeur de w lorsque E est petit. Cette capacité de passage à l'échelle est rendu possible au prix d'une communication plus fréquente entre les agents. En effet le nombre d'itérations faite pendant une époque est réduit lorsque w est grand. Ce coût de communication est cependant partagé entre les agents et reste donc constant en moyenne pour chaque agent. Cette différenciation entre le nombre d'agents et le nombre de discriminateurs permet donc à MD-GAN de pouvoir passer à l'échelle de la même manière que le *Federated Learning*.

Tolérance aux pannes de MD-GAN

De la même manière qu'*AdaComp*, il est important de mesurer la capacité de MD-GAN à faire face aux pannes des agents. Dans le cas où $w >> n$, le risque de pannes sur un agent actif reste faible et donc l'impact sur l'apprentissage peut être réduit. Cependant, lorsque w et n sont proches, ce risque peut s'avérer élevé (selon les machines de notre système distribué). Les pannes sont d'autant plus problématique que les machines fonctionnent de manière synchrone (tout comme le *Federated Learning*). Il faut donc instaurer une borne de temps maximale d'attente pour le serveur central avant de considérer un agent actif comme en panne.

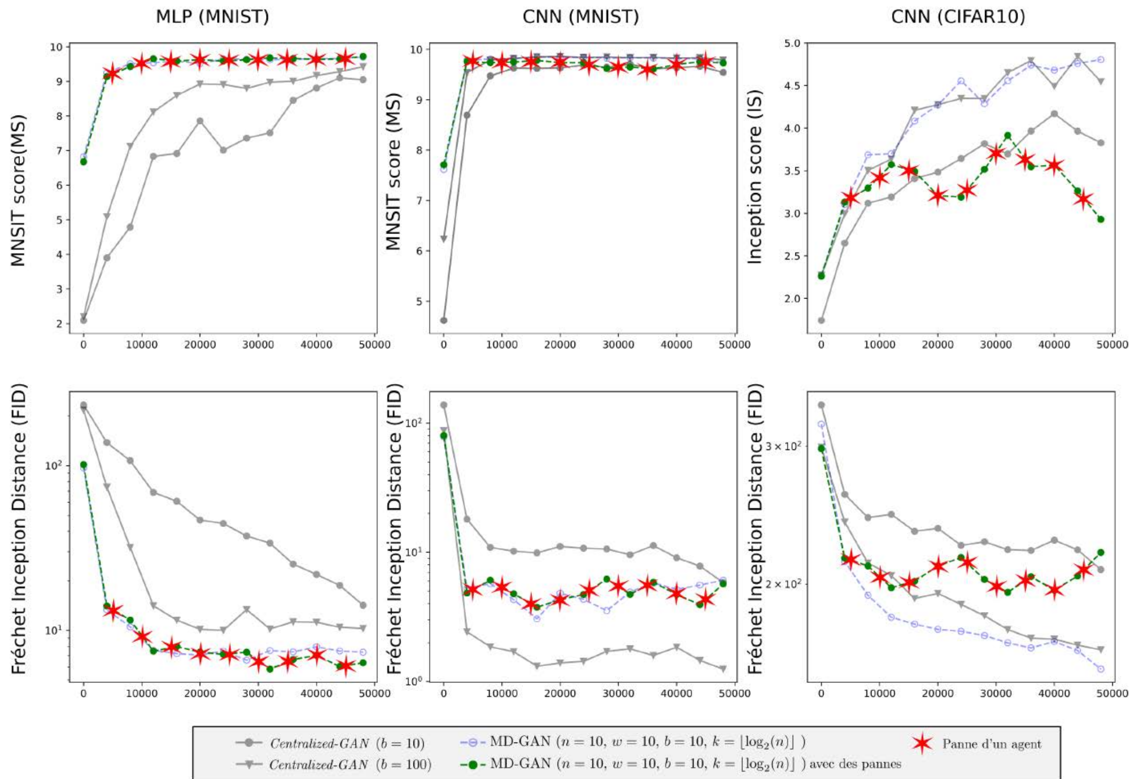


FIGURE 5.6 – Score MNIST ou score Inception (le cas échéant) et la distance Inception de Fréchet par rapport au nombre d'itérations pour MD-GAN avec des pannes, comparé à MD-GAN sans panne.

Afin de tester la tolérance du modèle face aux pannes, nous testons une expérience durant laquelle les agents peuvent s'arrêter au cours de l'apprentissage avec $n = w$. Le serveur central n'attend pas indéfiniment les retours d'erreur des agents actifs en panne. Nous opérons suivant le même scénario que pour la sous-section précédente avec le modèle du MLP. Les résultats sont présentés en Figure 5.6, avec la configuration la plus performante de MD-GAN (avec $k = \lfloor \log(n) \rfloor$), mais cette fois, nous programmons la panne d'un agent toutes les I/n itérations (apparaissant sur la courbe verte). La conséquence est que lorsque $I = 50\ 000$, tous les agents sont en panne.

Pour comparer, nous gardons les scores de *centralized-GAN* (c'est-à-dire, apprentissage sur une unique machine contenant B_{train}) avec deux tailles différentes de batch ($b \in \{10, 100\}$), ainsi que MD-GAN sans panne, avec les mêmes paramètres. La première observation est que le modèle avec des pannes n'est pas impacté significative-

ment sur ses performances (MS et FID) dans le cas du MLP sur MNIST. L'architecture MLP indique même la plus petite valeur de FD à la fin de l'expérimentation. De même qu'*AdaComp*, cette expérience démontre que MD-GAN est capable d'apprendre malgré les pannes (avec les pertes de données et de discriminateurs engendrées par celles-ci) tout en obtenant de bonnes performances. Les deux mesures sont affectées dans le cas de CIFAR10 : nous observons une divergence due aux pannes, et cela apparaît tôt dans la phase d'apprentissage (autour de $I = 5\ 000$, ce qui correspond à la première panne d'un agent). Cette expérimentation montre la sensibilité aux pannes précoces de MD-GAN. Les GAN n'ont pas eu assez de temps pour estimer précisément la distribution des données. La perte des discriminateurs ainsi que des données ne permettent pas aux agent survivants d'être suffisamment performants. Ainsi, le générateur ne peut atteindre un score compétitif. Les scores obtenus restent encore comparables au *centralized-GAN* jusqu'à la 8ème panne. Nous noterons cependant que, dans le cas où $w >> n$, le serveur central peut détecter l'absence de réponse d'agents actifs et donc garder n constant en relançant un nouveau discriminateur sur un agent inactif.

Validation sur une base de données plus grande

Dans cette expérimentation, nous validons la convergence de MD-GAN, et son intérêt vis-à-vis du *centralized-GAN* et de FL-GAN. L'objectif est d'entraîner un GAN sur la base de données CelebA [71] qui est composée de 200 000 images de célébrités (au dimension 128×128 pixels). Nous utilisons 10K images comme ensemble de test, et le reste des images est partagé entre les $n \in [1, 5]$ agents de manière i.i.d. L'architecture GAN est une variante de celle utilisée pour le dataset CIFAR10 : $\mathcal{G}$ est composé d'une couche totalement interconnectée de $16\ 384$ neurones et de deux couches de déconvolution, de respectivement 128 et 3 filtres de dimensions 5 × 5 ; $\mathcal{D}$ est composé de six couches de convolution de respectivement 16, 32, 64, 128, 256 et 512 filtres de dimensions 3×3, et d'une couche toute connectée d'un neurone. La taille du batch pour le *centralized-GAN* et le FL-GAN est $b = 200$, tandis que la taille du batch de MD-GAN est $b = 40$ (ce qui correspond à 200 images à traiter pour le calcul d'une modification du générateur). Dans cette expérimentation, nous utilisons deux paramètrages distincts de l'optimiseur Adam, ce qui donne de meilleurs résultats pour chaque compétiteur. Le *centralized-GAN* et le FL-GAN utilisent un taux d'apprentissage de $\alpha = 0.003$ pour $\mathcal{G}$ (resp. $\alpha = 0.002$ pour $\mathcal{D}$), et $\beta_1 = 0.5$, $\beta_2 = 0.999$ pour $\mathcal{G}$ et $\mathcal{D}$, tandis que MD-GAN uti-

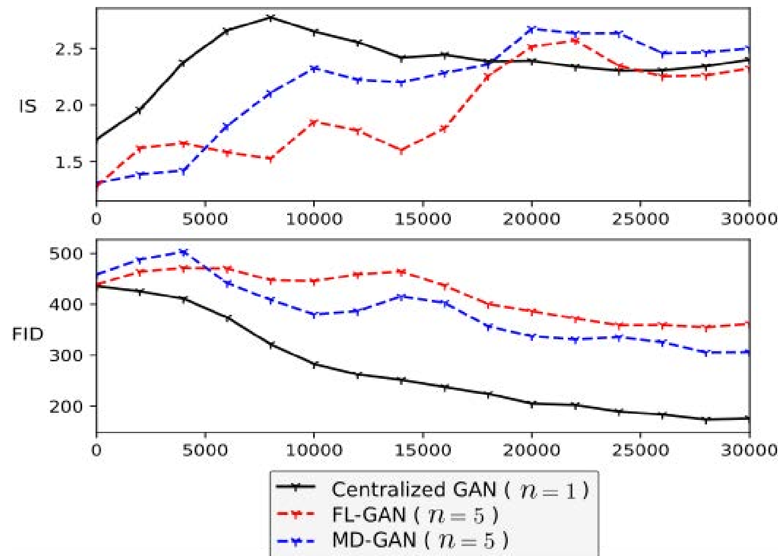


FIGURE 5.7 – IS et FID des trois compétiteurs pour la base de données CelebA.

lise un taux d'apprentissage de $\alpha = 0.001$ pour $\mathcal{G}$ (resp. $\alpha = 0.004$ pour $\mathcal{D}$), et $\beta_1 = 0.0$, $\beta_2 = 0.9$ pour $\mathcal{G}$ et $\mathcal{D}$. Le FID et le score d'Inception résultants pendant les $30\ 000$ itérations que nous avons considérées sont reportés en Figure 5.6.

Nous observons que tous les scores d'Inception sont comparables (MD-GAN est légèrement au-dessus) ; alors qu'au regard du score FID, MD-GAN (tout comme FL-GAN) est distancé par le *centralized-GAN* (comme c'est le cas du CNN avec MNIST).

5.4 MD-GAN et les travaux connexes

L'apprentissage distribué des GAN n'est pas un cas très étudié dans la littérature. Ceux-ci pouvant être vus comme des réseaux de neurones classiques, nous pouvons citer les travaux vus dans le Chapitre 2 afin de distribuer l'apprentissage profond sur de multiples machines, tels que [28, 41, 118, 117]. Nous avons vu dans le chapitre précédent que les méthodes de mise en commun par modèle moyen tel que le *Federated Learning* ne sont pas toujours efficaces.

Dans le cas d'apprentissage de GAN en centralisé, Im *et al.* proposent de multiplier le nombre de discriminateurs et de générateurs [56] : les auteurs entraînent plusieurs couples de GAN en parallèle et échangent les couples au cours de l'apprentissage afin de varier les adversaires de chacun. Durugkar *et al.* [31] proposent un autre exemple

d'architecture multi-discriminateur dans un cas centralisé afin d'augmenter la qualité de retour sur les données générées. De la même manière, Hoang *et al.* [49] étudient le cas où un discriminateur fait face à de multiples générateurs. Chacun d'eux peut apprendre un type de données différent afin d'obtenir un ensemble de générateurs variés à la fin de l'apprentissage. Les travaux de [112] et [104] améliorent ce modèle de *mixture* de générateurs avec des méthodes d'ensemble tel que le *boosting* (voir Section 2.3.1).

Notons que tous ces travaux ont été proposés afin d'améliorer l'apprentissage des GAN. Ils ne sont pas directement applicables dans le cas d'un apprentissage distribué. Notre contribution est une méthode utilisant l'avantage des adversaires multiples pour l'apprentissage de GAN dans un contexte de système distribué.

5.5 Perspectives et conclusion sur MD-GAN

Avant de conclure sur MD-GAN, nous souhaitons mettre en avant les pistes d'améliorations importantes à réfléchir pour l'apprentissage de GAN dans un environnement distribué.

Fonctionnement asynchrone

Au lieu d'attendre tous les retours d'erreur F_j à chaque itération globale, le serveur peut calculer un gradient $\Delta\boldsymbol{\lambda}$ et l'appliquer chaque fois qu'il reçoit un seul F_j à la manière d'une descente de gradient asynchrone. De nouveaux batches de données peuvent être générés fréquemment, de sorte qu'ils puissent être envoyés aux agents actifs qui ont envoyé leur retour. Ainsi, tous les agents peuvent fonctionner sans synchronisation globale, contrairement aux méthodes de *Federated Learning* telle que FL-GAN. Avec cette méthode, le temps d'attente des agents et du serveur est considérablement réduit à chaque itération. Cependant, en raison des modifications asynchrones, l'apprentissage se retrouverait dans une configuration proche du gradient avec délai de la descente de gradient asynchrone. Cependant, si ce délai est bien pris en compte, il peut ne pas impacter l'apprentissage du modèle comme nous l'avons vu au Chapitre 3.

Le goulot d'étranglement des communications du serveur central

La structure du serveur de paramètres, malgré sa simplicité, présente l'inconvénient évident de créer un goult d'étranglement en matière de communication vers le serveur central. Cela a été quantifié par plusieurs travaux [69, 42], et des solutions de réduction du trafic entre les agents et le serveur ont été proposées, à la manière de *AdaComp* dans le chapitre précédent.

Dans le contexte des réseaux GAN, ces méthodes peuvent être appliquées aux données générées avant leur envoi aux agents et aux messages de retours d'erreur F_i envoyés par les agents au serveur. En particulier, en ce qui concerne les données d'images, il existe de nombreuses techniques de compression (avec ou sans perte d'informations, voir par exemple, [113]).

Conclusion

Ce chapitre a présenté une méthode, MD-GAN, afin d'entraîner des réseaux antagonistes génératifs (GAN) dans le contexte de système distribué. Nous avons proposé un algorithme (MD-GAN) qui supprime la moitié de la complexité de calcul du coté des agents par rapport à la méthode du *Federated Learning*. Cette réduction de charge sur les agents est particulièrement utile dans le cas où ceux-ci ont des puissances de calcul limitées. MD-GAN combine les techniques de descente de gradient synchrone avec une utilisation des méthodes d'adversaires multiples afin de faire un apprentissage efficace du générateur final. Notre méthode obtient de meilleurs résultats que FL-GAN sur MNIST et CIFAR10. Les GAN nécessitent beaucoup de calculs et de données d'apprentissage, ce qui peut être problématique dans notre contexte de système distribué pour les machines participantes. Nous pensons que ce travail a apporté une première solution viable dans ce domaine.

Algorithm 3 Algorithme de MD-GAN côté agent

```
1: procedure WOKER(C, B_i, I, L, b, E, Statut)
2:     if Statut = "actif" then
3:         Initialisation de θ_j pour D_j
4:     else
5:         Attente d'un nouveau discriminateur.
6:         D_j ← RECEIVED                          ▷ Réception du nouveau discriminateur.
7:         Statut ← "actif"
8:     end if
9:     while itération I non atteinte do
10:        for t ← 1 to mE/b do
11:            X_j^(r) ← SAMPLES(B_i, b)
12:            X_j^(g), X_j^(d) ← RECEIVEBATCHES(C)
13:            for l ← 0 to L do
14:                D_j ← DISCLEARNINGSTEP(J_disc, D_j)
15:            end for
16:            F_j ← {∂B̃(X_n^(g))/∂x | x ∈ X_n^(g)}
17:            SEND(C, F_j)                        ▷ Envoi de F_j au serveur
18:        end for
19:        D_j ← MOVED(D_j)
20:    end while
21: end procedure
22:
23: procedure MOVED(D_j)
24:     W_l ← GETRANDOMWORKER
25:     SEND(W_l, D_j)                             ▷ Envoie de D_j à l'agent W_l.
26:     Statut ← "inactif"
27:     Attente d'un nouveau discriminateur.
28:     D_j ← RECEIVED                             ▷ Réception du nouveau discriminateur.
29:     Statut ← "actif"
30:     Return D_j
31: end procedure
```

Algorithm 4 Algorithme de MD-GAN coté serveur

1: **procedure** SERVER(k,I,b)
2: Initialisation de $\boldsymbol{\lambda}$ pour $\mathcal{G}$
3: **for** $t \leftarrow 1$ **to** I **do**
4: **for** $k \leftarrow 1$ **to** κ **do**
5: Génération de Z_j à partir d'une loi normale
6: $X^{(k)} \leftarrow \{\mathcal{G}_{\boldsymbol{\lambda}}(\boldsymbol{z}) | \boldsymbol{z} \in Z_k\}$
7: **end for**
8: $X_1^{(d)}, \ldots, X_n^{(d)} \leftarrow$ SPLIT($X^{(1)}, \ldots, X^{(\kappa)}$)
9: $X_1^{(g)}, \ldots, X_n^{(g)} \leftarrow$ SPLIT($X^{(1)}, \ldots, X^{(\kappa)}$)
10: $i_1, \ldots, i_n \leftarrow$ Indices des agents actifs
11: **for** $i_j \in \{i_1, \ldots, i_n\}$ **do**
12: SEND($W_{i_j}, (X_j^{(d)}, X_j^{(g)})$)
13: **end for**
14: $F_1, \ldots, F_n \leftarrow$ GETFEEDBACKFROMWORKERS
15: Calcul de $\Delta\boldsymbol{\lambda}$ en fonction de $F_1, \ldots, F_n$
16: **for** $\lambda_k \in \boldsymbol{\lambda}$ **do**
17: $\lambda_k \leftarrow \lambda_k +$ADAM($\Delta\lambda_k$)
18: **end for**
19: **end for**
20: **end procedure**

PERSPECTIVES ET CONCLUSION

Perspectives

L'apprentissage de réseaux de neurones profonds dans un contexte de système distribué est un domaine relativement vaste. De nombreux travaux ont déjà été réalisés concernant l'apprentissage de modèles plus simples distribués sur différents systèmes. Bien que l'apprentissage profond se déroule à une large échelle (les modèles sont complexes, nécessitant de grandes bases de données d'apprentissage et donc de nombreux contributeurs), il est intéressant de regarder les travaux effectués sur des modèles plus simples afin trouver des pistes à nos différentes problématiques. Durant cette thèse, nous avons principalement discuté :

- des contraintes de bande passante réseau (Chapitres 3, 4 et 5),
- des capacités de passage à l'échelle des algorithmes proposés (Chapitres 3 et 5),
- des problématiques dues aux pannes (Chapitres 3 et 5),
- des vitesses de calcul sur les machines utilisateurs hétérogènes (Chapitre 3),
- de la distribution des données des utilisateurs non nécessairement i.i.d. (Chapitre 4)

Nous allons voir ici d'autres problèmatiques concernant le cas d'utilisateurs curieux ou malveillants. Nous présentons ensuite un autre type de réseau de neurones peu connu sur lequel nous avons travaillé durant ma thèse. Nous nous sommes intéressés à son potentiel en tant que réseau de neurones profond.

Participants curieux ou malveillants

Pour rappel, notre motivation première est de permettre l'apprentissage d'un réseau de neurones profond à l'aide de la collaboration des futurs utilisateurs de ce modèle. Il est donc évident que certains d'entre eux peuvent être malveillants : c'est-à-dire, qu'ils cherchent à tirer profit du système ou le compromettre pour diverses raisons.

Faille concernant la vie privée Une garantie permise par notre système de collaboration est de pouvoir garder les données des contributeurs uniquement sur leurs machines. Le but étant de protéger la vie privée des utilisateurs. Cependant les communications lors de l'apprentissage (gradients ou modèles) peuvent être utilisées par des utilisateurs ou opérateurs curieux afin d'avoir des informations sur les données de certains utilisateurs. Le domaine de confidentialité différentielle s'intéresse à ce genre de problématique [32, 46]. L'idée est de mesurer (et réduire) les informations privées qu'il est possible d'obtenir en interrogeant une base de données anonyme. Dans notre contexte, il est intéressant de comprendre quelles informations privées peuvent être extraites des gradients ou des modèles envoyés sur le réseau. Les travaux de Shokri *et al.* utilisent la compression de gradients afin de réduire cette information lors d'une descente de gradient asynchrone. Cette méthode n'est cependant pas suffisante d'après les travaux de B. Hitaj' *et al.* [48]. Ces derniers ont appris un GAN capable de générer des données personnelles en influençant l'apprentissage et en observant le gradient compressé envoyé par l'agent victime de l'attaque. Dans les travaux de K. Bonawitz *et al.* [14], les auteurs proposent un mécanisme d'agrégation sécurisé pour regrouper les modèles appris sur différentes machines lors d'un apprentissage avec la méthode du *Federated Learning*.

Résistances aux erreurs byzantines Dans le cas d'utilisateurs malveillants, les contributions apportées par les agents de ces derniers ont pour but de détourner l'apprentissage du modèle. S'il est seul, un utilisateur malveillant doit être capable de faire une modification suffisament importante pour avoir un impact final sur le modèle. Par exemple, dans le cas d'une descente de gradient synchrone, il doit envoyer un gradient avec une norme élevée et une direction très différente de celles des autres agents pour perturber l'apprentissage. Ce genre d'attaque est facilement détectable en analysant les contributions de chacun des participants. Cependant, si plusieurs agents malveillants se coordonnent pour changer les paramètres du réseau de neurones profond dans la même direction, cette attaque devient beaucoup plus difficile à détecter. Des travaux ont été proposés par P. Blanchard *et al.* [12] pour effectuer une descente de gradient capable de résister à de nombreuses fautes byzantines. Leur méthode se base sur une règle d'agrégation des gradients appelée *Krum*. Les auteurs de ces travaux montrent qu'ils sont capables d'apprendre un MLP avec 20 agents dont jusqu'à 33% d'entre eux font des fautes byzantines. Cette algorithme se limite au cadre d'une

descente de gradient synchrone, mais il serait pertinent de l'adapter à une descente de gradient asynchrone, à une mise en commun par modèle moyen avec le *Federated Learning* ou encore à notre méthode MD-GAN sur les retours d'erreurs.

Utilisation d'un type de réseau de neurones différent : les réseaux de neurones aléatoires.

Durant nos travaux, nous avons étudié un type de réseau de neurones particuliers appelé réseau de neurones aléatoires. D'après nos expériences, ces réseaux de neurones peuvent être utilisés dans des architectures profondes classiques (tels que des CNN). Une piste de recherche peut donc s'intéresser à l'utilisation de réseaux de neurones aléatoires dans des systèmes distribués. Nous allons présenter ce type de réseaux de neurones et ses particularités.

Voir le neurone comme une file d'attente

Dans [37] et [33], E. Gelenbe a proposé d'utiliser les systèmes de file d'attente en tant que réseaux de neurones. Dans un RNA (Réseau de neurones aléatoires), les neurones sont considérés comme des files d'attente recevant deux catégories de clients : les clients positifs (libellés en $+$) et les clients négatifs (libellés en $-$). Les files d'attente sont des systèmes dynamiques modélisant l'utilisation des ressources par les clients dans des systèmes distribués. Comme pour les réseaux de neurones classiques, les files d'attente sont interconnectées et peuvent former différents types d'architectures.

À un temps t, le nombre N_t de clients d'une file d'attente s'appelle le *potentiel* du neurone et la *valeur d'activation* d'un neurone, indiquée par ϱ, est la probabilité que $N_t \geq 1$ lorsque le système dynamique atteint un état stable (c'est-à-dire, quand $t \to \infty$). La durée de service d'un client positif est distribuée selon la loi exponentielle avec le taux μ. La figure 5.8 est une représentation graphique d'une telle file.

Lorsqu'un client positif entre dans la file d'attente à l'instant t, le potentiel neuronal N_t augmente de 1. Lorsqu'un client négatif entre dans la file d'attente à l'instant t, alors si le potentiel $N_t \geq 1$, il est réduit de 1, et si $N_t = 0$, rien ne se produit. Notez que les clients négatifs n'ont pas de temps de service et qu'ils se suppriment toujours. Ils ne sont utilisés que pour supprimer les clients positifs de la file d'attente, le cas échéant.

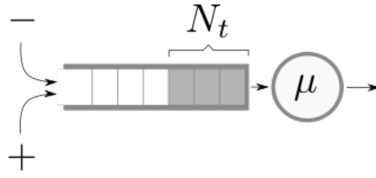


FIGURE 5.8 – Un neurone représenté par une file d'attente.

Lorsque son potentiel est $N_t \geq 1$, un neurone est dit "excité".

Un réseau de K neurones (ou files d'attente) se comporte comme suit. Si un client positif quitte le neurone i à l'instant t, le potentiel N_t diminue de un et il est envoyé au neurone j en tant que client positif avec une probabilité p_{ij}^{+}, ou en tant que client négatif avec une probabilité p_{ij}^{-}, et vers l'extérieur avec une probabilité $d_i = 1 - \sum_{j=1}^{K}(p_{ij}^{+} + p_{ij}^{-})$.

Au lieu de simuler le système jusqu'à ce qu'il atteigne l'état d'équilibre, E. Gelenbe a montré qu'il était possible de calculer la valeur d'activation des neurones de tout type de réseaux de neurones aléatoires. Soit T_j^{+} (resp. T_j^{-}), le débit des clients positifs (resp. négatifs) sur le neurone j, la valeur d'activation ϱ_j du neurone j est donnée par

$$\varrho_j = \begin{cases} \dfrac{T_j^{+}}{T_j^{-} + \mu_j} & \text{si } T_j^{+} < T_j^{-} + \mu_j, \\ 1 & \text{sinon.} \end{cases}$$

Le cas où $\varrho_j = 1$ se produit lorsque la file d'attente n'est pas stable. Le débit des clients positifs sur le neurone j est alors

$$T_j^{+} = \lambda_j^{+} + \sum_{i=1}^{K} \varrho_i w_{ij}^{+},$$

où λ_j^{+} est le débit moyen des clients positifs arrivant de l'extérieur du réseau sous forme de processus de Poisson (par exemple, l'entrée du réseau de neurones) et $w_{ij}^{+} = \mu_i p_{ij}^{+}$ est appelé le poids positif entre les neurones i et j, avec μ_i le taux de service du neurone i. De la même manière, le débit moyen T_j^{-} de clients négatifs au neurone j est égal à

$$T_j^{-} = \lambda_j^{-} + \sum_{i=1}^{K} \varrho_i w_{ij}^{-},$$

où λ_j^{-} est le débit moyen des clients négatifs arrivant de l'extérieur du réseau (c'est-à-dire, l'entrée du réseau de neurones), et $w_{ij}^{-} = \mu_i p_{ij}^{-}$.

Considérer un tel réseau de files d'attente conduit à un modèle d'apprentissage

automatique non linéaire où les poids w_{ij}^+ et w_{ij}^- sont les paramètres ajustables (au lieu des w_{ij} dans un réseau de neurone classique). Comme dans les réseau de neurones profond classiques, il est possible de former un RNA avec une fonction de coût par des méthodes de rétro-propagation et de descente de gradient (GD) [33] ou d'autres types d'optimiseurs [10].

Couche de neurones aléatoires (RNL)

Dans nos travaux, nous avons introduit la couche de neurones aléatoires (RNL pour *Random Neural Layer*), inspirée des couches de RNA : une couche de neurones prend en entrée la valeur d'activation de la couche de neurones précédente, ou l'entrée du réseau, pour calculer leurs valeurs d'activation.

Valeur d'activation d'une couche de neurones aléatoire. Soit un RNL avec un vecteur d'entrée $X \in \mathbb{R}_+^n$ (c'est-à-dire, l'entrée du réseau ou la sortie d'une couche précédente) et une matrice de poids positifs (resp. négatifs) $W^+ = \left(w_{ij}^+\right) \in \mathbb{R}_+^{m\times n}$ (resp. $W^- = \left(w_{ij}^-\right) \in \mathbb{R}_+^{m\times n}$). Pour $i = 1, \ldots, n$ et $j = 1, \ldots, m$, w_{ij}^+ et w_{ij}^- sont les poids entre la i-ème entrée de X et le neurone j du RNL. Rappelons que $1/\mu_j$ est le temps de service moyen du neurone j du RNL. On note M le vecteur de dimension m dont les entrées sont les μ_j. La valeur d'activation a_j du j-ième neurone de la couche RNL est calculée comme suit :

$$a_j = n_j/d_j,$$

où n_j et d_j sont respectivement les j-èmes entrées des vecteurs $N = W^+X$ et $D = W^-X + M$.

Rappelons que dans une couche neuronale toute connectée classique (FCL pour *Full-Connected Layer*) avec un vecteur d'entrée $X \in \mathbb{R}_+^n$ et l'activation Relu [79], dans laquelle $W = (w_{ij}) \in \mathbb{R}^{m\times n}$ sont les poids et $\beta = (\beta_j)$ est le vecteur de biais de la couche, la valeur d'activation a_j du neurone j de la couche est calculée comme

$$a_j = \max(0, z_j),$$

où z_j est la j-ième entrée du vecteur $Z = WX + \beta$.

Évaluation expérimentale

Nous avonc comparé la couche RNL face à des couches toutes connectés classique avec une fonction d'activation ReLU, notées FCL, sur des réseaux de l'état-de-l'art. Pour ce faire, nous avons seulement changé ces couches FCL par des couches RNL et comparé la différence en terme performances des différents modèles. Le Tableau 5.5 présente les scores de précision finale des modèles avec toutes les combinaisons possibles FCL/RNL d'un MLP entrainé sur MNIST. De même, le Tableau 5.6, présente les précisions obtenues sur un CNN entrainé sur MNIST et le Tableau 5.7 présente les précisions obtenus par un CNN sur la base de donnée CIFAR10. Le détail de ces expériences est disponible dans notre évaluation [44].

$1^{ère}$ couche	2nd couche	3rd couche	Précision finale
RNL	RNL	RNL	96.39%
RNL	RNL	FCL	97.16%
RNL	FCL	RNL	97.22%
RNL	FCL	FCL	97.31%
FCL	RNL	RNL	94.78%
FCL	RNL	FCL	98.06%
FCL	FCL	RNL	**98.35**%
FCL	FCL	FCL	98.31%

TABLE 5.5 – Précision finale du MLP sur MNIST après 20 époques ; le meilleur score est marqué en gras.

$1^{ère}$ couche	2e couche	3e couche	4e couche	Précision finale
CNN	CNN	RNL	RNL	98.12%
CNN	CNN	RNL	FCL	98.68%
CNN	CNN	FCL	RNL	98.96%
CNN	CNN	FCL	FCL	**99.03%**

TABLE 5.6 – Précision finale du CNN sur MNIST après 12 époques ; le meilleur score est marqué en gras. La notation CNN représente des couches convolutives.

Le potentiel des RNL

Dans les expérimentations que nous avons faites (voir Tableaux 5.5, 5.6 et 5.7) les architectures de réseaux de neurones hybrides avec une couche de RNL en dernière couche obtiennent généralement des scores presque aussi bons voir meilleurs que les

$1^{ère}$ couche	2e couche	3e couche	4e couche	5e couche	6e couche	Précision finale
CNN	CNN	CNN	CNN	RNL	RNL	75.57%
CNN	CNN	CNN	CNN	RNL	FCL	50.83%
CNN	CNN	CNN	CNN	FCL	RNL	**83.92%**
CNN	CNN	CNN	CNN	FCL	FCL	81.17%

TABLE 5.7 – Précision finale d'un CNN sur CIFAR10 après 50 époques ; le meilleur score est marqué en gras. La notation CNN représente des couches convolutives.

architectures classiques. Ces résultats nous questionnent donc sur l'intérêt de ce type de couches de manière générale dans les réseaux de neurones profonds.

Conclusion

Dans cette thèse, nous avons montré l'intérêt de l'apprentissage profond dans des systèmes distribués avec l'application de l'apprentissage collaboratif d'un réseau de neurones profond. Cette application a l'avantage de simplifier la création de la base de données d'apprentissage et de réduire les coûts en terme de structure de calculs tout en gardant les données privées localisées chez les utilisateurs.

Dans un premier temps, nous avons présenté les principales contraintes et difficultés de cette apprentissage profond dans un système distribué. Nous avons ensuite proposé différentes méthodes afin de répondre en partie au contraintes exposées. *AdaComp* permet de réaliser une descente de gradient asynchrone afin d'apprendre des réseaux profonds sur de nombreuses machines interconnectées tout en réduisant les coûts en terme de communications. Nous avons montré que cette méthode était même capable d'obtenir de meilleurs résultats que la descente de gradient asynchrone standard et qu'elle était capable de faire face à des pannes et à la participation ma machines hétérogène (en terme de puissance de calcul). Nous avons ensuite motivé, puis étudié, la possibilité d'un apprentissage de modèle tel que les GAN dans un système distribué de manière décentralisée. Nous avons montré que les méthodes qui utilisait des protocoles de rumeur se rapprochait mais ne dépassait pas les scores de la méthode du *Federated Learning*. Finalement, nous avons proposé une nouvelle méthode appelée MD-GAN qui utilise un serveur central pour réduire la charge de travail des machines participantes tout en gardant les données localisées sur ces dernières. Cette méthode utilise une architecture avec des adversaires multiples afin de dépasser les performances du *Federated Learning* de manière significative.

Contributions

— Corentin Hardy, Erwan Le Merrer and Bruno Sericola, “Distributed deep learning on edge-devices : feasibility via adaptive compression”, *16th IEEE International Symposium on Network Computing and Applications (NCA)*, Cambridge, MA, USA, 2017 (Best Paper Award).

— Corentin Hardy, Erwan Le Merrer and Bruno Sericola, “MD-GAN :Multi-Discriminator Generative Adversarial Networks for Distributed Datasets”, *33rd International Parallel and Distributed Processing Symposium (IPDPS’19)*, Rio de Janeiro, Brazil, 2019.

— Corentin Hardy, Erwan Le Merrer and Bruno Sericola, “Gossiping GANs”, *Second Workshop on Distributed Infrastructures for Deep Learning (DIDL)*, Rennes, France, 2018.

— C. Hardy, E. Le Merrer, G. Rubino, B. Sericola. “Evaluation of Random Neural Layers in Deep Neural Networks”. *Poster in workshop NIPS on Deep Learning :Bridging Theory and Practice*, Long Beach, CA, USA, 2017.

— C. Hardy, E. Le Merrer, B. Sericola. “Distributed deep learning on edge-devices in the Parameter Server Model”. *Poster in workshop on Decentralized Machine Learning, Optimization and Privacy*, Lille, France, 2017.

Code

Le code Python de certaines contributions est disponible en ligne sur GitHub.

— La méthode *AdaComp* présentée dans le Chapitre 3 :
https://github.com/Hardy-c/AdaComp.

— Les couches de neurones aléatoires présentés dans les perspectives :
https://github.com/Hardy-c/DNN-with-RNL.

BIBLIOGRAPHIE

[1] Martín ABADI et David G. ANDERSEN, « Learning to Protect Communications with Adversarial Neural Cryptography », in : *CoRR* abs/1610.06918 (2016), arXiv : 1610.06918, URL : http://arxiv.org/abs/1610.06918.

[2] Martín ABADI et al., « Tensorflow : a system for large-scale machine learning. », in : *OSDI*, t. 16, 2016, p. 265-283.

[3] Martín ABADI et al., « TensorFlow : Large-Scale Machine Learning on Heterogeneous Distributed Systems », in : *CoRR* abs/1603.04467 (2016), URL : http://arxiv.org/abs/1603.04467.

[4] Alekh AGARWAL et John C DUCHI, « Distributed Delayed Stochastic Optimization », in : *Advances in Neural Information Processing Systems 24*, sous la dir. de J. SHAWE-TAYLOR et al., Curran Associates, Inc., 2011, p. 873-881, URL : http://papers.nips.cc/paper/4247-distributed-delayed-stochastic-optimization.pdf.

[5] Jose AGUILAR et Cristhian MOLINA, « The Multilayer Random Neural Network », in : *Neural Processing Letters* 37.*2* (2013), p. 111-133, ISSN : 1573-773X, DOI : 10.1007/s11063-012-9237-x.

[6] Eirikur AGUSTSSON et al., « Generative Adversarial Networks for Extreme Learned Image Compression », in : *CoRR* abs/1804.02958 (2018), arXiv : 1804.02958, URL : http://arxiv.org/abs/1804.02958.

[7] Dario AMODEI et al., « Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin », in : *Proceedings of The 33rd International Conference on Machine Learning*, sous la dir. de Maria Florina BALCAN et Kilian Q. WEINBERGER, t. 48, Proceedings of Machine Learning Research, New York, New York, USA : PMLR, 2016, p. 173-182, URL : http://proceedings.mlr.press/v48/amodei16.html.

[8] M. ARJOVSKY et L. BOTTOU, « Towards Principled Methods for Training Generative Adversarial Networks », in : *ArXiv e-prints* (jan. 2017), arXiv : 1701.04862 [stat.ML].

[9] M. ARJOVSKY, S. CHINTALA et L. BOTTOU, « Wasserstein GAN », in : *ArXiv e-prints* (jan. 2017), arXiv : 1701.07875 [stat.ML].

[10] Sebastián BASTERRECH et Gerardo RUBINO, « A Tutorial about Random Neural Networks in Supervised Learning », in : *CoRR* abs/1609.04846 (2016).

[11] Kevin BEYER et al., « When Is “Nearest Neighbor” Meaningful ? », in : *Database Theory — ICDT'99*, sous la dir. de Catriel BEERI et Peter BUNEMAN, Berlin, Heidelberg : Springer Berlin Heidelberg, 1999, p. 217-235, ISBN : 978-3-540-49257-3.

[12] Peva BLANCHARD et al., « Machine Learning with Adversaries : Byzantine Tolerant Gradient Descent », in : *Advances in Neural Information Processing Systems 30*, sous la dir. d'I. GUYON et al., Curran Associates, Inc., 2017, p. 119-129, URL : http://papers.nips.cc/paper/6617-machine-learning-with-adversaries-byzantine-tolerant-gradient-descent.pdf.

[13] M. BLOT et al., « Gossip training for deep learning », in : *ArXiv e-prints* (nov. 2016), arXiv : 1611.09726 [cs.CV].

[14] Keith BONAWITZ et al., « Practical Secure Aggregation for Privacy-Preserving Machine Learning », in : *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, Dallas, Texas, USA : ACM, 2017, p. 1175-1191, ISBN : 978-1-4503-4946-8, DOI : 10.1145/3133956.3133982, URL : http://doi.acm.org/10.1145/3133956.3133982.

[15] Léon BOTTOU, « Online Algorithms and Stochastic Approximations », in : *Online Learning and Neural Networks*, sous la dir. de David SAAD, revised, oct 2012, Cambridge, UK : Cambridge University Press, 1998, URL : http://leon.bottou.org/papers/bottou-98x.

[16] Stephen BOYD et al., « Randomized gossip algorithms », in : *IEEE transactions on information theory* 52.*6* (2006), p. 2508-2530.

[17] Leo BREIMAN, « Random forests », in : *Machine learning* 45.*1* (2001), p. 5-32.

[18] Ignacio CANO et al., « Towards Geo-Distributed Machine Learning », in : *CoRR* abs/1603.09035 (2016), arXiv : 1603.09035, URL : http://arxiv.org/abs/1603.09035.

[19] Olivier CHAPELLE, Jason WESTON et Bernhard SCHÖLKOPF, « Cluster Kernels for Semi-Supervised Learning », in : *Advances in Neural Information Processing Systems 15*, sous la dir. de S. BECKER, S. THRUN et K. OBERMAYER, MIT Press, 2003, p. 601-608, URL : http://papers.nips.cc/paper/2257-cluster-kernels-for-semi-supervised-learning.pdf.

[20] Jianmin CHEN et al., « Revisiting Distributed Synchronous SGD », in : *International Conference on Learning Representations Workshop Track*, 2016, URL : https://arxiv.org/abs/1604.00981.

[21] Muthuraman CHIDAMBARAM et Yanjun QI, « Style Transfer Generative Adversarial Networks : Learning to Play Chess Differently », in : *CoRR* abs/1702.06762 (2017).

[22] Trishul CHILIMBI et al., « Project Adam : Building an Efficient and Scalable Deep Learning Training System », in : *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO : USENIX Association, 2014, p. 571-582, ISBN : 978-1-931971-16-4, URL : https://www.usenix.org/conference/osdi14/technical-sessions/presentation/chilimbi.

[23] François CHOLLET, *Keras*, https://github.com/fchollet/keras, 2015.

[24] Ronan COLLOBERT et Jason WESTON, « A unified architecture for natural language processing : Deep neural networks with multitask learning », in : *Proceedings of the 25th international conference on Machine learning*, ACM, 2008, p. 160-167.

[25] Paul COVINGTON, Jay ADAMS et Emre SARGIN, « Deep Neural Networks for YouTube Recommendations », in : *Proceedings of the 10th ACM Conference on Recommender Systems*, RecSys '16, Boston, Massachusetts, USA : ACM, 2016, p. 191-198, ISBN : 978-1-4503-4035-9, DOI : 10.1145/2959100.2959190, URL : http://doi.acm.org/10.1145/2959100.2959190.

[26] Jeff DAILY et al., « GossipGraD : Scalable Deep Learning using Gossip Communication based Asynchronous Gradient Descent », in : *CoRR* abs/1803.05880 (2018), arXiv : 1803.05880, URL : http://arxiv.org/abs/1803.05880.

[27] Jeffrey DEAN et Sanjay GHEMAWAT, « MapReduce : Simplified Data Processing on Large Clusters », in : *Commun. ACM* 51.*1* (jan. 2008), p. 107-113, ISSN : 0001-0782, DOI : 10.1145/1327452.1327492, URL : http://doi.acm.org/10.1145/1327452.1327492.

[28] Jeffrey DEAN et al., « Large Scale Distributed Deep Networks », in : *Advances in Neural Information Processing Systems 25*, sous la dir. de F. PEREIRA et al., Curran Associates, Inc., 2012, p. 1223-1231, URL : http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf.

[29] Giuseppe DECANDIA et al., « Dynamo : Amazon's Highly Available Key-value Store », in : *SOSP*, 2007.

[30] John DUCHI, Elad HAZAN et Yoram SINGER, « Adaptive subgradient methods for online learning and stochastic optimization », in : *Journal of Machine Learning Research* 12.*Jul* (2011), p. 2121-2159.

[31] I. DURUGKAR, I. GEMP et S. MAHADEVAN, « Generative Multi-Adversarial Networks », in : *5th International Conference on Learning Representations (ICLR 2017)* (nov. 2016), arXiv : 1611.01673.

[32] Cynthia DWORK, « Differential Privacy : A Survey of Results », in : *Theory and Applications of Models of Computation*, sous la dir. de Manindra AGRAWAL et al., Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, p. 1-19, ISBN : 978-3-540-79228-4.

[33] E. GELENBE, « Learning in the Recurrent Random Neural Network », in : *Neural Computation* 5.*1* (1993), p. 154-164, DOI : 10.1162/neco.1993.5.1.154.

[34] E. GELENBE et J. M. FOURNEAU, « Random Neural Networks with Multiple Classes of Signals », in : *Neural Computation* 11.*4* (1999), p. 953-963, ISSN : 0899-7667, DOI : 10.1162/089976699300016520.

[35] E. GELENBE et K. F. HUSSAIN, « Learning in the multiple class random neural network », in : *IEEE Transactions on Neural Networks* 13.*6* (2002), p. 1257-1267, ISSN : 1045-9227, DOI : 10.1109/TNN.2002.804228.

[36] E. GELENBE et Y. YIN, « Deep learning with Random Neural Networks », in : *2016 International Joint Conference on Neural Networks (IJCNN)*, 2016, p. 1633-1638, DOI : 10.1109/IJCNN.2016.7727393.

[37] Erol GELENBE, « Random Neural Networks with Negative and Positive Signals and Product Form Solution », in : *Neural Comput.* 1.4 (déc. 1989), p. 502-510, ISSN : 0899-7667, DOI : 10.1162/neco.1989.1.4.502.

[38] Michael GEORGIOPOULOS, Cong LI et Taskin KOCAK, « Learning in the feed-forward Random Neural Network : A Critical Review », in : *Computer and Information Sciences*, sous la dir. d'Erol GELENBE et al., Dordrecht : Springer Netherlands, 2010, p. 155-160, ISBN : 978-90-481-9794-1.

[39] I. J. GOODFELLOW et al., « Generative Adversarial Networks », in : *ArXiv e-prints* (juin 2014), arXiv : 1406.2661 [stat.ML].

[40] Mihajlo GRBOVIC et al., « Scalable Semantic Matching of Queries to Ads in Sponsored Search Advertising », in : *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '16, Pisa, Italy : ACM, 2016, p. 375-384, ISBN : 978-1-4503-4069-4, DOI : 10.1145/2911451.2911538, URL : http://doi.acm.org/10.1145/2911451.2911538.

[41] Suyog GUPTA, Wei ZHANG et Fei WANG, « Model Accuracy and Runtime Tradeoff in Distributed Deep Learning : A Systematic Study », in : *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, IJCAI'17, Melbourne, Australia : AAAI Press, 2017, p. 4854-4858, ISBN : 978-0-9992411-0-3, URL : http://dl.acm.org/citation.cfm?id=3171837.3171972.

[42] C. HARDY, E. LE MERRER et B. SERICOLA, « Distributed deep learning on edge-devices : Feasibility via adaptive compression », in : *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, 2017, p. 1-8, DOI : 10.1109/NCA.2017.8171350.

[43] Corentin HARDY, Erwan LE MERRER et Bruno SERICOLA, « Distributed deep learning on edge-devices : feasibility via adaptive compression », in : *CoRR* abs/1702.04683 (2017), URL : https://arxiv.org/abs/1702.04683.

[44] Corentin HARDY et al., *Evaluation of Random Neural Layers in Deep Neural Networks*, NIPS 2017 - Workshop Deep Learning : Bridging Theory and Practice, Poster, déc. 2017, URL : https://hal.inria.fr/hal-01657644.

[45] Kaiming HE et al., « Deep Residual Learning for Image Recognition », in : *arXiv e-prints*, arXiv :1512.03385 (déc. 2015), arXiv :1512.03385, arXiv : 1512.03385 [cs.CV].

[46] I. HEGEDUS et M. JELASITY, « Distributed Differentially Private Stochastic Gradient Descent : An Empirical Study », in : *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, 2016, p. 566-573, DOI : 10.1109/PDP.2016.19.

[47] M. HEUSEL et al., « GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium », in : *ArXiv e-prints* (juin 2017), arXiv : 1706.08500 [cs.LG].

[48] Briland HITAJ, Giuseppe ATENIESE et Fernando PEREZ-CRUZ, « Deep Models Under the GAN : Information Leakage from Collaborative Deep Learning », in : *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, Dallas, Texas, USA : ACM, 2017, p. 603-618, ISBN : 978-1-4503-4946-8, DOI : 10.1145/3133956.3134012, URL : http://doi.acm.org/10.1145/3133956.3134012.

[49] Q. HOANG et al., « Multi-Generator Generative Adversarial Nets », in : *ArXiv e-prints* (août 2017), arXiv : 1708.02556.

[50] Sepp HOCHREITER et Jürgen SCHMIDHUBER, « Long Short-Term Memory », in : *Neural Computation* 9.*8* (1997), p. 1735-1780, DOI : 10.1162/neco.1997.9.8.1735, eprint : https://doi.org/10.1162/neco.1997.9.8.1735, URL : https://doi.org/10.1162/neco.1997.9.8.1735.

[51] Kurt HORNIK, « Approximation capabilities of multilayer feedforward networks », in : *Neural Networks* 4.*2* (1991), p. 251 -257, ISSN : 0893-6080, DOI : "https://doi.org/10.1016/0893-6080(91)90009-T", URL : "http://www.sciencedirect.com/science/article/pii/089360809190009T".

[52] Kevin HSIEH et al., « Gaia : Geo-Distributed Machine Learning Approaching LAN Speeds », in : *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, Boston, MA : USENIX Association, 2017, p. 629-647, ISBN : 978-1-931971-37-9, URL : https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/hsieh.

[53] Jie HU et al., « Squeeze-and-Excitation Networks », in : *arXiv e-prints*, arXiv :1709.01507 (sept. 2017), arXiv :1709.01507, arXiv : 1709.01507 [cs.CV].

[54] Sergey IOFFE et Christian SZEGEDY, « Batch Normalization : Accelerating Deep Network Training by Reducing Internal Covariate Shift », in : *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, Lille, France : JMLR.org, 2015, p. 448-456, URL : http://dl.acm.org/citation.cfm?id=3045118.3045167.

[55] Yanghua JIN et al., « Towards the Automatic Anime Characters Creation with Generative Adversarial Networks », in : *arXiv e-prints*, arXiv :1708.05509 (août 2017), arXiv :1708.05509, arXiv : 1708.05509 [cs.CV].

[56] D. JIWOONG IM et al., « Generative Adversarial Parallelization », in : *ArXiv e-prints* (déc. 2016), arXiv : 1612.04021.

[57] Leslie Pack KAELBLING, Michael L LITTMAN et Andrew W MOORE, « Reinforcement learning : A survey », in : *Journal of artificial intelligence research* 4 (1996), p. 237-285.

[58] Diederik P. KINGMA et Jimmy BA, « Adam : A Method for Stochastic Optimization », in : *CoRR* abs/1412.6980 (2014), URL : http://arxiv.org/abs/1412.6980.

[59] Durk P KINGMA et al., « Semi-supervised Learning with Deep Generative Models », in : *Advances in Neural Information Processing Systems 27*, sous la dir. de Z. GHAHRAMANI et al., Curran Associates, Inc., 2014, p. 3581-3589, URL : http://papers.nips.cc/paper/5352-semi-supervised-learning-with-deep-generative-models.pdf.

[60] Jakub KONECNÝ, Brendan MCMAHAN et Daniel RAMAGE, « Federated Optimization : Distributed Optimization Beyond the Datacenter », in : *CoRR* abs/1511.03575 (2015), URL : http://arxiv.org/abs/1511.03575.

[61] J. KONEČNÝ et al., « Federated Learning : Strategies for Improving Communication Efficiency », in : *CoRR* abs/1610.05492 (oct. 2016).

[62] Alex KRIZHEVSKY, *Learning Multiple Layers of Features from Tiny Images*, 2009.

[63] Alex KRIZHEVSKY, Ilya SUTSKEVER et Geoffrey E HINTON, « ImageNet Classification with Deep Convolutional Neural Networks », in : *Advances in Neural Information Processing Systems 25*, sous la dir. de F. PEREIRA et al., Curran Associates, Inc., 2012, p. 1097-1105, URL : http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[64] Leslie LAMPORT, Robert SHOSTAK et Marshall PEASE, « The Byzantine Generals Problem », in : *ACM Trans. Program. Lang. Syst.* 4.*3* (juil. 1982), p. 382-401, ISSN : 0164-0925, DOI : 10.1145/357172.357176, URL : http://doi.acm.org/10.1145/357172.357176.

[65] Y. LECUN et al., « Efficient BackProp », in : *Neural Networks : Tricks of the trade*, sous la dir. de G. ORR et Muller K., Springer, 1998, p. 9-50.

[66] Y. LECUN et al., « Gradient-Based Learning Applied to Document Recognition », in : *Proceedings of the IEEE* 86.*11* (1998), p. 2278-2324.

[67] Yann LECUN, Corinna CORTES et Christopher JC BURGES, *The MNIST database of handwritten digits*, http://yann.lecun.com/exdb/mnist, 1998, URL : http://yann.lecun.com/exdb/mnist/.

[68] C. LEDIG et al., « Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network », in : *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, p. 105-114, DOI : 10.1109/CVPR.2017.19.

[69] Mu LI et al., « Scaling Distributed Machine Learning with the Parameter Server », in : *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO : USENIX Association, 2014, p. 583-598, ISBN : 978-1-931971-16-4, URL : https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu.

[70] Xiangru LIAN et al., « Asynchronous Parallel Stochastic Gradient for Nonconvex Optimization », in : *Advances in Neural Information Processing Systems 28*, sous la dir. de C. CORTES et al., Curran Associates, Inc., 2015, p. 2737-2745, URL : http://papers.nips.cc/paper/5751-asynchronous-parallel-stochastic-gradient-for-nonconvex-optimization.pdf.

[71] Ziwei LIU et al., « Deep Learning Face Attributes in the Wild », in : *Proceedings of International Conference on Computer Vision (ICCV)*, 2015.

[72] D. G. LOWE, « Object recognition from local scale-invariant features », in : *Proceedings of the Seventh IEEE International Conference on Computer Vision*, t. 2, 1999, 1150-1157 vol.2, DOI : 10.1109/ICCV.1999.790410.

[73] Liqian MA et al., « Pose Guided Person Image Generation », in : *Advances in Neural Information Processing Systems 30*, sous la dir. d'I. GUYON et al., Curran Associates, Inc., 2017, p. 406-416, URL : http://papers.nips.cc/paper/6644-pose-guided-person-image-generation.pdf.

[74] T. MAGER, E. BIERSACK et P. MICHIARDI, « A measurement study of the Wuala on-line storage service », in : *2012 IEEE 12th International Conference on Peer-to-Peer Computing (P2P)*, 2012, p. 237-248, DOI : 10.1109/P2P.2012.6335804.

[75] H. Brendan MCMAHAN et al., « Ad Click Prediction : A View from the Trenches », in : *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, Chicago, Illinois, USA : ACM, 2013, p. 1222-1230, ISBN : 978-1-4503-2174-7, DOI : 10.1145/2487575.2488200, URL : http://doi.acm.org/10.1145/2487575.2488200.

[76] H. Brendan MCMAHAN et al., « Federated Learning of Deep Networks using Model Averaging », in : *CoRR* abs/1602.05629 (2016), URL : http://arxiv.org/abs/1602.05629.

[77] Tomas MIKOLOV et al., « Distributed Representations of Words and Phrases and their Compositionality », in : *Advances in Neural Information Processing Systems 26*, sous la dir. de C. J. C. BURGES et al., Curran Associates, Inc., 2013, p. 3111-3119, URL : http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf.

[78] Tom MITCHELL, *Machine learing*, McGraw-Hill Science/Engineering/Math, 1997.

[79] Vinod NAIR et Geoffrey E. HINTON, « Rectified Linear Units Improve Restricted Boltzmann Machines », in : *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, Haifa, Israel : Omnipress, 2010, p. 807-814, ISBN : 978-1-60558-907-7, URL : http://dl.acm.org/citation.cfm?id=3104322.3104425.

[80] Mathias NIEPERT, Mohamed AHMED et Konstantin KUTZKOV, « Learning Convolutional Neural Networks for Graphs », in : *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, New York, NY, USA : JMLR.org, 2016, p. 2014-2023, URL : http://dl.acm.org/citation.cfm?id=3045390.3045603.

[81] A. ODENA, « Faster Asynchronous SGD », in : *CoRR* abs/1601.04033 (jan. 2016).

[82] Augustus ODENA, Christopher OLAH et Jonathon SHLENS, « Conditional Image Synthesis with Auxiliary Classifier GANs », in : *Proceedings of the 34th International Conference on Machine Learning*, sous la dir. de Doina PRECUP et Yee Whye TEH, t. 70, Proceedings of Machine Learning Research, International Convention Centre, Sydney, Australia : PMLR, 2017, p. 2642-2651.

[83] X. PAN et al., « CYCLADES : Conflict-free Asynchronous Machine Learning », in : *ArXiv e-prints* (mai 2016), arXiv : 1605.09721 [stat.ML].

[84] N. PAPERNOT et al., « The Limitations of Deep Learning in Adversarial Settings », in : *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, 2016, p. 372-387, DOI : 10.1109/EuroSP.2016.36.

[85] Omkar M. PARKHI, Andrea VEDALDI et Andrew ZISSERMAN, « Deep Face Recognition », in : *Proceedings of the British Machine Vision*, 2015.

[86] Pitch PATARASUK et Xin YUAN, « Bandwidth optimal all-reduce algorithms for clusters of workstations », in : *Journal of Parallel and Distributed Computing* 69.*2* (2009), p. 117-124.

[87] G. PERARNAU et al., « Invertible Conditional GANs for image editing », in : *ArXiv e-prints* (nov. 2016), arXiv : 1611.06355 [cs.CV].

[88] Lutz PRECHELT, « Automatic early stopping using cross validation : quantifying the criteria », in : *Neural Networks* 11.*4* (1998), p. 761 -767, ISSN : 0893-6080, DOI : https://doi.org/10.1016/S0893-6080(98)00010-0, URL : http://www.sciencedirect.com/science/article/pii/S0893608098000100.

[89] Benjamin RECHT et al., « Hogwild : A Lock-Free Approach to Parallelizing Stochastic Gradient Descent », in : *Advances in Neural Information Processing Systems 24*, sous la dir. de J. SHAWE-TAYLOR et al., Curran Associates, Inc., 2011, p. 693-701, URL : http://papers.nips.cc/paper/4390-hogwild-a-

lock-free-approach-to-parallelizing-stochastic-gradient-descent.pdf.

[90] S. REED et al., « Generative Adversarial Text to Image Synthesis », in : *ArXiv e-prints* (mai 2016), arXiv : 1605.05396.

[91] David E RUMELHART, Geoffrey E HINTON et Ronald J WILLIAMS, « Learning representations by back-propagating errors », in : *Cognitive modeling* 5.*3* (1988).

[92] Olga RUSSAKOVSKY et al., « ImageNet Large Scale Visual Recognition Challenge », in : *International Journal of Computer Vision (IJCV)* 115.*3* (2015), p. 211-252, DOI : 10.1007/s11263-015-0816-y.

[93] T. SALIMANS et al., « Improved Techniques for Training GANs », in : *ArXiv e-prints* (juin 2016), arXiv : 1606.03498 [cs.LG].

[94] Tom SCHAUL, Sixin ZHANG et Yann LECUN, « No More Pesky Learning Rates », in : *arXiv e-prints*, arXiv :1206.1106 (2012), arXiv :1206.1106, arXiv : 1206.1106 [stat.ML].

[95] Bernhard SCHÖLKOPF, Alexander J SMOLA, Francis BACH et al., *Learning with kernels : support vector machines, regularization, optimization, and beyond*, MIT press, 2002.

[96] Alexander SERGEEV et Mike Del BALSO, « Horovod : fast and easy distributed deep learning in TensorFlow », in : *CoRR* abs/1802.05799 (2018), arXiv : 1802.05799, URL : http://arxiv.org/abs/1802.05799.

[97] Shaohuai SHI et al., « Benchmarking State-of-the-Art Deep Learning Software Tools », in : *CoRR* abs/1608.07249 (2016), URL : http://arxiv.org/abs/1608.07249.

[98] Reza SHOKRI et Vitaly SHMATIKOV, « Privacy-Preserving Deep Learning », in : *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, Denver, Colorado, USA : ACM, 2015, p. 1310-1321, ISBN : 978-1-4503-3832-5, DOI : 10.1145/2810103.2813687, URL : http://doi.acm.org/10.1145/2810103.2813687.

[99] David SILVER et al., « Mastering the game of Go without human knowledge », in : *Nature* 550.*7676* (2017), p. 354.

[100] Nitish SRIVASTAVA et al., « Dropout : A Simple Way to Prevent Neural Networks from Overfitting », in : *Journal of Machine Learning Research* 15 (2014), p. 1929-1958, URL : http://jmlr.org/papers/v15/srivastava14a.html.

[101] Ilya SUTSKEVER et al., « On the importance of initialization and momentum in deep learning », in : *International conference on machine learning*, 2013, p. 1139-1147.

[102] Christian SZEGEDY et al., « Going Deeper With Convolutions », in : *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[103] S. TEERAPITTAYANON, B. MCDANEL et H. T. KUNG, « Distributed Deep Neural Networks Over the Cloud, the Edge and End Devices », in : *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, p. 328-339, DOI : 10.1109/ICDCS.2017.226.

[104] Ilya TOLSTIKHIN et al., « AdaGAN : Boosting Generative Models », in : *arXiv e-prints*, arXiv :1701.02386 (jan. 2017), arXiv :1701.02386, arXiv : 1701.02386 [stat.ML].

[105] Sarah UNDERWOOD, « Blockchain Beyond Bitcoin », in : *Commun. ACM* 59.*11* (oct. 2016), p. 15-17, ISSN : 0001-0782, DOI : 10.1145/2994581, URL : http://doi.acm.org/10.1145/2994581.

[106] Vytautas VALANCIUS et al., « Greening the Internet with Nano Data Centers », in : *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, Rome, Italy : ACM, 2009, p. 37-48, ISBN : 978-1-60558-636-6, DOI : 10.1145/1658939.1658944, URL : http://doi.acm.org/10.1145/1658939.1658944.

[107] Ashish VASWANI et al., « Attention is All you Need », in : *Advances in Neural Information Processing Systems 30*, sous la dir. d'I. GUYON et al., Curran Associates, Inc., 2017, p. 5998-6008, URL : http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf.

[108] P. VIOLA et M. JONES, « Rapid object detection using a boosted cascade of simple features », in : *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*, t. 1, 2001, p. I-I, DOI : 10.1109/CVPR.2001.990517.

[109] C. VONDRICK, H. PIRSIAVASH et A. TORRALBA, « Generating Videos with Scene Dynamics », in : *ArXiv e-prints* (sept. 2016), arXiv : 1609.02612 [cs.CV].

[110] A. WAIBEL et al., « Phoneme recognition using time-delay neural networks », in : *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37.*3* (1989), p. 328-339, ISSN : 0096-3518, DOI : 10.1109/29.21701.

[111] Minjie WANG, Chien-chin HUANG et Jinyang LI, « Supporting Very Large Models using Automatic Dataflow Graph Partitioning », in : *CoRR* abs/1807.08887 (2018), arXiv : 1807.08887, URL : http://arxiv.org/abs/1807.08887.

[112] Yaxing WANG, Lichao ZHANG et Joost VAN DE WEIJER, « Ensembles of Generative Adversarial Networks », in : *arXiv e-prints*, arXiv :1612.00991 (déc. 2016), arXiv :1612.00991, arXiv : 1612.00991 [cs.CV].

[113] M. J. WEINBERGER, G. SEROUSSI et G. SAPIRO, « The LOCO-I lossless image compression algorithm : principles and standardization into JPEG-LS », in : *IEEE Transactions on Image Processing* 9.*8* (2000), p. 1309-1324, ISSN : 1057-7149, DOI : 10.1109/83.855427.

[114] Yifan WU, Fan YANG et Haibin LING, « Privacy-Protective-GAN for Face Deidentification », in : *CoRR* abs/1806.08906 (2018), arXiv : 1806.08906, URL : http://arxiv.org/abs/1806.08906.

[115] Yonghua YIN et Erol GELENBE, « Nonnegative autoencoder with simplified Random Neural Network », in : *CoRR* abs/1609.08151 (2016).

[116] Matthew D. ZEILER, « ADADELTA : An Adaptive Learning Rate Method », in : *CoRR* abs/1212.5701 (2012), arXiv : 1212.5701, URL : http://arxiv.org/abs/1212.5701.

[117] Sixin ZHANG, Anna E CHOROMANSKA et Yann LECUN, « Deep learning with Elastic Averaging SGD », in : *Advances in Neural Information Processing Systems 28*, sous la dir. de C. CORTES et al., Curran Associates, Inc., 2015, p. 685-693, URL : http://papers.nips.cc/paper/5761-deep-learning-with-elastic-averaging-sgd.pdf.

[118] Wei ZHANG et al., « Staleness-aware Async-SGD for Distributed Deep Learning », in : *CoRR* abs/1511.05950 (2015), URL : http://arxiv.org/abs/1511.05950.

[119] Xinyang ZHANG, Shouling JI et Ting WANG, « Differentially Private Releasing via Deep Generative Model », in : *CoRR* abs/1801.01594 (2018), arXiv : 1801.01594, URL : http://arxiv.org/abs/1801.01594.

[120] S. ZHENG et al., « Asynchronous Stochastic Gradient Descent with Delay Compensation », in : *ArXiv e-prints* (sept. 2016), arXiv : 1609.08326.

[121] J. ZHU et al., « Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks », in : *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, p. 2242-2251, DOI : 10.1109/ICCV.2017.244.

[122] Martin ZINKEVICH et al., « Parallelized Stochastic Gradient Descent », in : *Advances in Neural Information Processing Systems 23*, sous la dir. de J. D. LAFFERTY et al., Curran Associates, Inc., 2010, p. 2595-2603, URL : http://papers.nips.cc/paper/4006-parallelized-stochastic-gradient-descent.pdf.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Contribution au développement de l'apprentissage profond dans les systèmes distribués

Mots-clés : Réseaux de neurones profonds, Apprentissage automatique, Calcul distribué

Résumé : L'apprentissage profond permet de développer un nombre de services de plus en plus important. Il nécessite cependant de grandes bases de données d'apprentissage et beaucoup de puissance de calcul. Afin de réduire les coûts de cet apprentissage profond, nous proposons la mise en œuvre d'un apprentissage collaboratif. Les futures utilisateurs des services permis par l'apprentissage profond peuvent ainsi participer à celui-ci en mettant à disposition leurs machines ainsi que leurs données sans déplacer ces dernières sur le *cloud*. Nous proposons différentes méthodes afin d'apprendre des réseaux de neurones profonds dans ce contexte de système distribué.

Title : Contribution to the development of deep learning in distributed systems

Keywords : Deep neural networks, Machine learning, Distributed computing

Abstract : Deep learning enables the development of a growing number of services. However, it requires large training databases and a lot of computing power. In order to reduce the costs of this deep learning, we propose a distributed computing setup to enable collaborative learning. Future users can participate with their devices and their data without moving private data in datacenters. We propose methods to train deep neural network in this distibuted system context.