Comparaison des *frameworks* d'apprentissage profond

Jean-Sébastien Lerat

Haute École en Hainaut, ESTISIM

jean-sebastien.lerat@heh.be

Résumé

L'apprentissage automatique est un ensemble de méthodes issues de l'intelligence artificielle qui visent à apprendre à l'aide de données. Au sein de ces méthodes, l'apprentissage profond se concentre sur des réseaux de neurones qui sont des modèles mathématiques dont l'objectif est de s'inspirer du fonctionnement du cerveau à l'aide de neurones artificiels. Ceux-ci sont très performants lorsqu'il s'agit de reconnaître des entités sur des images et vidéos. Ces méthodes sont possibles grâce aux améliorations technologiques qui permettent d'entraîner plus rapidement des modèles de plus en plus complexes, en particulier sur la carte graphique. Le problème se pose lorsqu'un temps de réponse court est requis ou lorsque le modèle neuronal est trop conséquent pour tenir en mémoire. Dans ce cas il faut distribuer la charge de travail sur plusieurs nœuds de calculs. Afin d'implémenter ce type de solution efficacement et rapidement, différents *frameworks* existent mais ceux-ci n'ont pas encore été comparés en termes d'utilisation de ressources, de stabilité et de performance sur un seul nœud. L'objectif de ce travail est de comparer les *frameworks* d'apprentissage profond sur le CPU et le GPU à l'aide de différents jeux de données et de différents modèles de réseaux de neurones.

Mots-clés : *deep learning*, *framework*, *machine learning*, *benchmark*, *image classification*.

1 Introduction

L'intelligence artificielle est une tendance qui fait régulièrement la une des journaux et des médias. Par exemple, la classification d'images qui permet de reconnaître une entité sur une photo.

Les réseaux de neurones sont des modèles mathématiques dont l'objectif est de s'inspirer du fonctionnement du cerveau à l'aide de neurones artificiels. Ce type de modèle est très performant sur les tâches de classification, en particulier via l'apprentissage profond. Celui-ci consiste à empiler des ensembles de neurones appelés couches.

Des bibliothèques logicielles appelées *frameworks* permettent de développer rapidement une solution d'apprentissage profond, en particulier lorsqu'il existe un interfaçage[1] avec un langage de script tel que Python. La littérature montre qu'il existe plusieurs *frameworks* qui sont activement utilisés et qui offrent des fonctionnalités similaires.

Le choix du *framework* est crucial dans le cas de l'informatique durable (*green* IT), ou de calculs distribués ou bien encore quand les calculs sont déportés en périphérie (*edge computing*). En effet, il est important d'utiliser au mieux les ressources que ça soit pour diminuer la consommation électrique ou bien pour concevoir un système de calculs distribués. De plus, diminuer le temps de calcul est essentiel

[1] En informatique, l'interfaçage est le mécanisme qui permet à deux entités de communiquer. Dans le texte cela correspond à un ensemble de méthodes natives implémentées en C qui peuvent être appelées et exécutées depuis le langage Python.

lorsqu'un modèle doit être entraîné ou mis à jour. Par exemple, l'industrie 4.0 préconise d'utiliser le *cloud computing* afin d'entraîner les modèles. Or le coût dépend directement de la quantité de ressources utilisées et du temps de calcul.

L'évaluation de *frameworks* d'apprentissage profond nécessite d'exécuter une même tâche, composée d'opérations, sur chacun de ces *frameworks*. Dans ce travail, la tâche consiste à entraîner des réseaux de neurones convolutifs (CNN) bien qu'il existe différentes familles de réseaux de neurones dans l'apprentissage profond. Au niveau des *frameworks*, le type de réseaux de neurones implique un nombre différent d'opérations à traiter. Or le protocole expérimental de ce travail considère différents CNN dont la complexité varie. Cette variation permet d'obtenir des résultats par rapport à un nombre différent d'opérations.

Initialement les CNN ont été conçus pour classer des images. Ce type de traitement et de réseaux de neurones est assez représentatif de l'apprentissage profond. Dans ce travail, deux jeux de données de taille différente vont être exploités afin d'évaluer le temps d'exécution des *frameworks*.

Ce travail s'intéresse à la consommation de ressources des différents *frameworks* et leur temps d'exécution en fonction de l'architecture du réseau de neurones, de la taille des données et de leur manière d'exploiter le GPU quand il est disponible. Cela permettrait, dépendant des résultats, de mettre en avant un *framework* particulier plus stable et efficace à la fois sur le CPU et sur le GPU. Faust, Lima-Mendez, Lerat, Sathirapongsasuti, Knight, Huttenhower, Lenaerts & Raes (2015) ont conçu un logiciel et démontré l'importance des métriques afin d'évaluer un réseau d'associations microbiennes.

2 État de l'art

Le temps de réponse et les capacités de calcul sont une préoccupation commune des praticiens de l'apprentissage profond. Une solution classique à ce problème consiste à répartir la tâche d'apprentissage profond sur un *cluster*. En 2015, une interface Spark a été proposée afin de faciliter l'apprentissage profond distribué, ce qui augmente la capacité de calcul et permet de diminuer le temps de réponse.

Moritz, Nishihara, Stoica & Jordan (2015) ont conçu une adaptation de Caffe, plus performante selon les auteurs, sur la technologie de calcul distribué Spark. Par la suite, Shi, Wang & Chu (2018) ont analysé le temps de calculs de quatre *frameworks* d'apprentissage profond depuis un environnement composé d'un seul GPU vers un environnement multi-GPU. La meilleure combinaison d'apprentissage en profondeur distribué était Caffe-MPI, qui surpassait MXNet, Tensorflow et CNTK. Néanmoins, ces comparaisons sont obsolètes en raison de l'évolution des *frameworks*. Par exemple, le *framework* Caffe a évolué et a été intégré au *framework* PyTorch. De plus, l'analyse est uniquement basée sur la distribution sans tenir compte des considérations locales. Or Lerat, Han & Lenaerts (2013) ont souligné que la coopération entre les nœuds est un élément crucial. Le comportement des *frameworks* sur un seul nœud de calcul n'est pas étudié en détail mais uniquement utilisé en tant que base de référence pour mesurer l'accélération de l'apprentissage en profondeur distribué. Néanmoins il n'est pas possible d'extrapoler sur base de ces données afin de déporter le calcul en périphérie puisque les nœuds du système distribué sont caractérisés par de faibles capacités de calcul.

Plus tard, Zhang, Zheng, Xu, Dai, Ho, Liang, Hu, Wei, Xie & Xing (2017) ont développé une surcouche logicielle afin de créer un *framework* d'apprentissage profond basé sur le *framework* Tensorflow. Ceux-ci affirment qu'il est le plus performant. Selon Shanmugamani (2018) le meilleur *framework* d'apprentissage profond est Tensorflow et sa surcouche Keras.

Mayer & Jacobsen ont comparé les *frameworks* en termes d'API, de prise en charge de l'apprentissage profond distribué et parallélisé et de l'importance de la communauté, sans prendre en compte leurs performances. De leur comparaison, Tensorflow est le *framework* préféré de la communauté. Dans notre article, nous nous concentrons sur un seul nœud avec un seul appareil afin de comparer les *frameworks* sur leurs performances locales.

3 Modèles et méthodes

Dans cette section nous décrivons la tâche d'apprentissage profond ainsi que le choix des *frameworks* et des architectures neuronales. Deux tâches interviennent quand il s'agit de réseaux de neurones. Ils sont tout d'abord entraînés, ensuite ils sont exploités. Chacune de ces deux tâches nécessite de prendre une instance de données en entrée et de faire évoluer ces données à travers le réseau de neurones, ce qui produit le résultat. Cependant la tâche d'entraînement nécessite de corriger a posteriori le réseau de neurones, en particulier lorsque le résultat est erroné. C'est pourquoi nous nous sommes concentrés sur la tâche d'apprentissage qui est plus complexe d'un point de vue calculs et mémoire.

3.1 Réseaux de neurones

L'évaluation de *frameworks* nécessite d'entraîner des réseaux de neurones qui diffèrent en complexité soit en nombre de couches neuronales ou sur la complexité des couches. Sans perte de généralité sur le type de réseau de neurones, ce travail se concentre sur les réseaux neuronaux convolutifs (CNN) qui sont particulièrement adaptés au traitement d'images et de vidéos dans le but d'effectuer des tâches de classification. Ce type de réseaux est bien connu de la communauté scientifique, en particulier ceux conçus pour le ImageNet Large Scale Visual Recognition Challenge (INLSVRC).

Dans le but de concevoir une analyse équitable entre les *frameworks*, une sélection d'architectures CNN a été effectuée sur base des challengers du INLSVRC. Les CNN sélectionnés sont AlexNet, MobileNet version 2, ResNet50 et VGG16 qui diffèrent en complexité. La dernière couche de ces réseaux a été modifiée par rapport à leur publication originale afin de les adapter à notre problème de classification à trois étiquettes.

L'architecture AlexNet de Krizhevsky, A., Sutskever & Hinton (2012) a gagné le INLSVRC et a surpassé tous les compétiteurs précédents. Ce fût le modèle le plus complexe de l'époque et réduisant ainsi significativement le top 5 d'erreurs de 26 % à 15,3 %. Cette architecture empile les couches de convolution (11x11, 5x5x3) avec des couches de mise en commun maximum, la fonction d'activation ReLu et avec une couche d'abandon.

En 2014, le Visual Geometry Group a proposé une nouvelle architecture de CNN. Ce groupe lui donne le nom de VGGnet de Simonyan & Zisserman (2014), plus connu simplement comme VGG. La version d'origine est composée de 16 couches (VGG16) de convolution avec 138 millions de paramètres.

En 2015, la nouvelle architecture Residual Neural Network CNN de He, Zhang, Ren & Sun (2016), plus couramment appelée ResNet, est proposée au défi ImageNet. Cette architecture ne propose pas seulement d'adapter ses paramètres comme ses prédécesseurs mais également de choisir le nombre de couches. De plus, elle intègre un nouveau concept qui est l'apprentissage résiduel.

La version 2 de MobileNet de Howard, A., Zhmoginov, A., Chen, L. C., Sandler, M., & Zhu, M. (2018), peut être vue comme une version simplifiée de ResNet qui nécessite donc moins de paramètres. Cela provient du fait d'utiliser un mécanisme de convolution particulier plutôt que les couches complexes de ResNet appelées *bottleneck blocks*. Ce mécanisme de convolution est composé de deux couches : une

convolution appliquée sur chaque canal de l'image et une convolution par point sur les trois canaux. Contrairement aux autres architectures, MobileNet n'utilise pas de couche de mise en commun.

3.2 Protocole expérimental

La tâche d'entraînement pour un problème de classification d'images doit prendre en considération comment fournir les données en entrée du réseau de neurones depuis une image. Ces images peuvent être prétraitées afin de faire correspondre la quantité d'information à celle attendue par le réseau de neurones. Dans ce travail, le prétraitement des images a été conçu à l'aide du processus utilisé dans les publications d'origine des CNN choisis plutôt que de concevoir un prétraitement optimal qui accroît le taux de prédictions correctes étant donné que l'objectif du travail est d'analyser le comportement des *frameworks*. C'est pourquoi les images sont prétraitées afin d'obtenir un tenseur de taille 224x224 sur les 3 canaux RVB. L'objectif est de quantifier l'utilisation des ressources d'une tâche d'apprentissage commune. La séquence de prétraitement des images est la suivante :

1. Étirement et découpe des images à 224x224 pixels
2. Échange aléatoire des lignes
3. Normalisation RVB avec une moyenne de (0,485 ; 0,456 ; 0,406) et un écart type de (0,229 ; 0,224 ; 0,225)
4. Conversion en structure de tenseur du *framework*

La méthode d'optimisation est le gradient stochastique avec un taux d'apprentissage de 0,001 et un moment de 0,9. La fonction de perte est l'entropie croisée. Les deux jeux de données[2] ont été fournis par le service ILIA du département des sciences de la faculté d'ingénieur de l'université de Mons. Le petit jeu de données (*small*) se compose de 791 photos et le grand jeu de données (*big*) se compose de 6003 photos. Chaque jeu de données possède trois classes : feu, fumée, pas de feu.

L'objectif est bien de mesurer l'utilisation des ressources informatiques utilisées par les *frameworks* et non pas d'optimiser les paramètres d'apprentissage afin d'obtenir un taux de prédiction acceptable. C'est pourquoi l'ensemble de chaque jeu de données est utilisé afin d'entraîner les CNN à l'aide des différents *frameworks* sans les diviser en un jeu de test et un jeu de validation. Rechercher d'autres valeurs d'hyperparamètres tels que la manière de prétraiter les données ou bien encore la méthode d'optimisation ne vont que permettre d'accélérer le traitement ou bien améliorer la précision, ce qui n'est pas l'objectif de ce travail.

3.3 Frameworks

Bien que plusieurs *frameworks* existent dans la littérature scientifique, seuls quelques *frameworks* sont pertinents dans le cadre de ce travail. En effet, dans un souci de compatibilité et de performance, les *frameworks* ont été sélectionnés sur base de quatre critères :

1. Une implémentation native afin d'obtenir des temps de réponses courts, sans surcouche logicielle telle qu'un interpréteur.
2. Une prise en compte des bibliothèques CUDA et OpenCL afin de tirer parti des GPUs. Une interface Python afin de permettre le prototypage rapide et qui est le langage le plus exploité

[2] Des jeux de données de tailles différentes ont été utilisés mais étant donné la linéarité des résultats, nous ne reprenons que le petit et le grand jeux de données.

par la communauté d'apprentissage profond. En effet, Python est un langage de script concis qui permet d'implémenter rapidement une solution logicielle.
3. Une communauté active afin d'assurer la longévité et l'adaptabilité du *framework* au cours du temps face aux nouvelles technologies.

Les cinq *frameworks* qui respectent ces conditions ont été initiés ou sont supportés par une entreprise ou une fondation : MXNet d'Apache Foundation, Paddle de Baidu Incorporated Compagnies, pyTorch de Facebook Incorporated Compagnies, Singa de Apache Foundation et Tensorflow de Google. Ooi, Tan, Wang, Wang, Cai, Chen, Gao, Luo, Thung, Wang, Xie, Zhang & Zheng (2015) continuent à développer Singa dans le but de le spécialiser dans l'apprentissage profond distribué.

3.4 Configuration matérielle

La machine d'expérimentation est équipée d'un processeur AMD 2950x RYZEN THREADRIPPER, d'une carte graphique EVGA GeForce RTX 2080 Ti et de 128Go de mémoire vive en DDR4 à 2933Mhz. Le jeu de données est stocké sur un disque SDD 860 EVO 4To de Samsung tandis que la partie logicielle est installée sur un SSD 970 Pro M.2 PCIe NVMe 1To de Samsung.

4 Résultats

Les temps d'exécution moyens sur 100 époques[3] exprimés en secondes sont repris aux Tableaux 1 et 2 qui correspondent respectivement sur le CPU et le GPU. Sur CPU, MXNet et Tensorflow traitent plus rapidement MobileNet contrairement à Paddle, pyTorch et Singa qui traitent plus rapidement AlexNet. Les deux architectures les plus coûteuses en temps sont ResNet et VGG, par ordre croissant. Comme mentionné précédemment, Singa est incapable de gérer automatiquement le jeu de données plus conséquent et ne fournit donc que des résultats sur le petit jeu de données. En effet, Singa lance un fil d'exécution (*thread*) à chaque image du jeu de données afin de la charger et de la préparer en mémoire. Le système arrive à saturation dans le cas du grand jeu de données.

Framework	**AlexNet**		**MobileNet v2**		**ResNet 50**		**VGG 16**	
	Small	*Big*	*Small*	*Big*	*Small*	*Big*	*Small*	*Big*
MxNet	401	2993	181	1391	547	4181	2178	16480
Paddle	430	1107	431	1045	2273	5790	7273	16493
pyTorch	139	879	221	1459	480	3122	630	9010
Singa	273	/	430	/	977	/	1227	/
Tensorflow	528	3957	420	3223	835	6289	1979	16397

Tableau 1 : Moyenne des temps d'exécution sur 100 époques exploitant le CPU, exprimée en seconde.

Sur GPU, bien que Singa ne soit capable que de gérer le petit jeu de données, il se montre plus rapide avec un temps de 6 secondes sur toutes les architectures. Celui-ci charge toutes les données directement en mémoire et les transfère au GPU, c'est pourquoi il se montre plus rapide. Il n'y a pas de temporisation et de limitation dans l'utilisation des ressources. Les autres *frameworks* ne permettent pas de traiter aussi

[3] Une époque est un passage complet sur les données d'entraînement.

rapidement les données mais ils sont capables de gérer le grand jeu de données. La différence par rapport au CPU, c'est que les opérations matricielles s'effectuent en parallèle sur le GPU et permettent donc de fournir des résultats plus rapidement, en particulier quand l'architecture neuronale nécessite plus d'opérations matricielles. Dans cette configuration, AlexNet est l'architecture la plus rapide à traiter sauf pour Tensorflow qui prend 2 secondes de plus par rapport à MobileNet. La complexité de l'architecture VGG devient équivalente à celle de ResNet. Clairement Paddle et Tensorflow sont les *frameworks* qui nécessitent le plus de temps par rapport à pyTorch et MXNet. En effet, l'écart est de l'ordre de 300 secondes sur le petit jeu de données et de 2000 secondes sur le grand jeu de données. D'après ces résultats, pyTorch est le plus rapide sur CPU, excepté dans la configuration MXNet sur MobileNet et MXNet est le plus rapide sur GPU suivi de près par pyTorch.

Framework	**AlexNet**		**MobileNet v2**		**ResNet 50**		**VGG 16**	
	Small	*Big*	*Small*	*Big*	*Small*	*Big*	*Small*	*Big*
MxNet	19	114	28	182	33	232	28	185
Paddle	328	2380	338	2533	332	2500	337	2462
pyTorch	23	122	40	254	43	271	34	190
Singa	6	/	6	/	6	/	6	/
Tensorflow	343	2542	341	2564	351	2605	350	2648

Tableau 2 : Moyenne des temps d'exécution sur 100 époques exploitant le GPU, exprimée en seconde.

Lorsque la configuration change et que l'on passe d'un gradient stochastique à une donnée vers le traitement par petits lots (*mini-batch*) à 100 données[4], MXNet et Paddle ne sont pas capables d'adapter leur utilisation des ressources et font crasher l'exécution. Notons qu'aucune erreur n'est apparue avec une taille de 50. Etant donné le critère de stabilité, ces *frameworks* ont été retirés de notre analyse. Les *frameworks* capables d'adapter leur utilisation des ressources sont pyTorch et Tensorflow. Les Figures 1 et 2 montrent respectivement l'utilisation du CPU, de la RAM et le nombre de fils d'exécution utilisés par pyTorch et Tensorflow. Afin de mieux comprendre l'influence de la quantité de données et de la complexité de l'architecture neuronale, deux cas d'utilisation ont été étudiés : *SimpleBig* qui correspond à AlexNet avec le grand jeu de données et *ComplexSmall* qui correspond à VGG avec le petit jeu de données.

Avec un gradient stochastique, pyTorch a un nombre de fils d'exécution qui varie autour de 13 (11 à 17) lorsqu'il y a beaucoup de données et un modèle simple à traiter. De ce cas-là, c'est la manière de charger les données qui va permettre de diminuer le temps d'exécution. Ce *framework* semble donc charger des données jusqu'à atteindre un seuil de 18 % (≈ 23 Go) d'exploitation de la mémoire vive. Dans les deux cas de figure, pyTorch arrive à exploiter le CPU à 100 %, il n'y a donc pas de perte de ressource et de temps CPU. Cependant sa quantité mémoire semble dépendre des données à charger étant donné qu'il ne nécessite plus que 2 % (≈ 2,56 Go) de mémoire vive lorsqu'il traite le cas du *ComplexSmall*. Le nombre de fils d'exécution a une valeur constante de 11. Lorsqu'il est exécuté en mini-batch, pyTorch change son comportement et fixe son nombre de fils d'exécution à 11 dans tous

[4] Des tests ont été réalisé avec différentes tailles de lots, par pas de 10, afin de vérifier que les *frameworks* sont capables de s'adapter aux besoins des utilisateurs. Les *frameworks* qui ne sont pas capables de s'adapter échouent à une taille de 100 ou à une taille inférieure. C'est pourquoi la taille de 100 a été choisie.

les cas bien qu'il garde une activité CPU à 100 %. Ce qui est intéressant c'est qu'en chargeant les données de la sorte, il réduit sa quantité de mémoire vive nécessaire en-dessous des 2 % dans les deux cas pour un temps d'exécution amélioré par rapport au simple gradient stochastique.

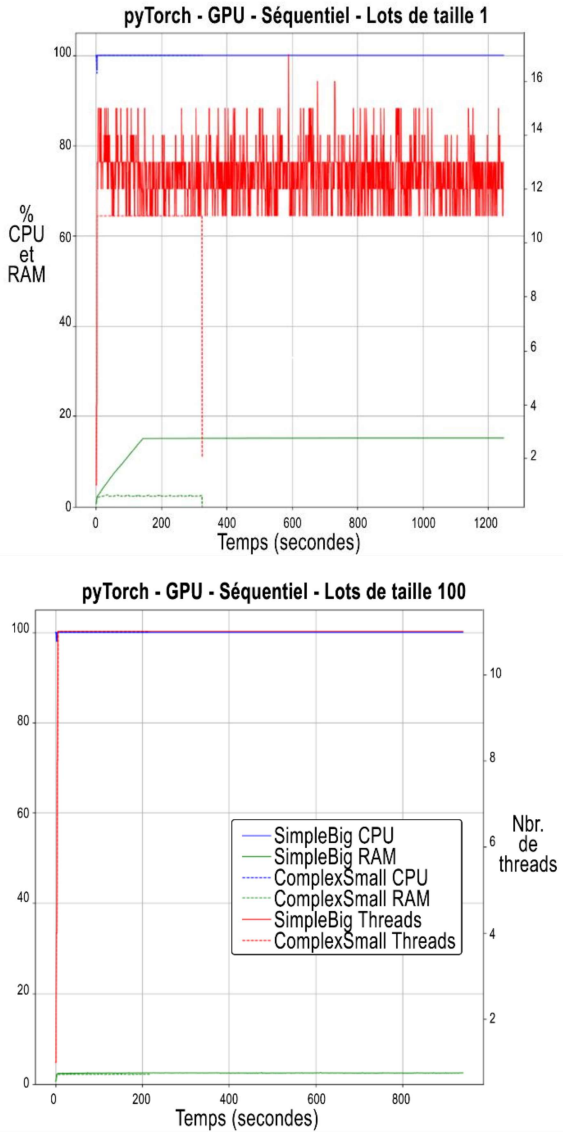

Figure 1 : Utilisation de la RAM, du CPU et nombre de fils d'exécution avec le framework pyTorch.

Dans les Tableaux 1 et 2, pyTorch est bien plus efficace que Tensorflow et nous avons constaté qu'il utilise les ressources de manière à se réguler sur des seuils d'utilisation, en particulier le CPU qui est pleinement utilisé, ceci afin de fournir le GPU en données d'entrée. Tensorflow quant à lui a un profil d'utilisation assez différent et varie plus sur les trois critères de mesures. Lorsqu'il est utilisé avec le gradient stochastique, son nombre de fils d'exécution est plus important lorsqu'il est utilisé avec un modèle compliqué et peu de données que lorsqu'il est utilisé sur un modèle simple avec beaucoup de données contrairement à pyTorch. En effet, le nombre de fils d'exécution oscille de 18 à 20 dans ce cas et de 9 à 19 dans le cas du *SimpleBig* après une période de croissance linéaire. De manière analogue, l'utilisation du CPU augmente rapidement jusque 90 % et augmente linéairement vers les 100 %. Après la croissance linéaire, le taux d'utilisation du CPU n'est pas stable et oscille entre 95 % et 100 % dans le cas du *ComplexSmall* et du *SimpleBig*. Bien que Tensorflow nécessite plus de temps que pyTorch, il

nécessite moins de mémoire et augmente par pas jusqu'à 2 % d'utilisation contre 18 % pour pyTorch. Il semble donc moins gourmand en mémoire. Lorsqu'il est exécuté en mini-batch, son comportement change et à tendance à éviter les phases de croissance linéaire. En effet, son taux d'utilisation du CPU oscille rapidement entre 95 % et 100 % et l'utilisation de la mémoire vive varie entre 2 % et 30 %. Son temps d'exécution s'en trouve nettement amélioré par rapport à pyTorch et passe de ≈85000 secondes à ≈3500 secondes contre ≈1250 secondes à ≈950 secondes pour pyTorch. Notons que les temps d'exécution moyens par époque sont meilleurs lorsqu'il n'y a pas de processus qui mesurent l'utilisation des ressources. En mini-batch, Tensorflow a encore un nombre de fils d'exécution qui varie de 18 à 22 mais oscille moins fréquemment qu'avec le gradient stochastique.

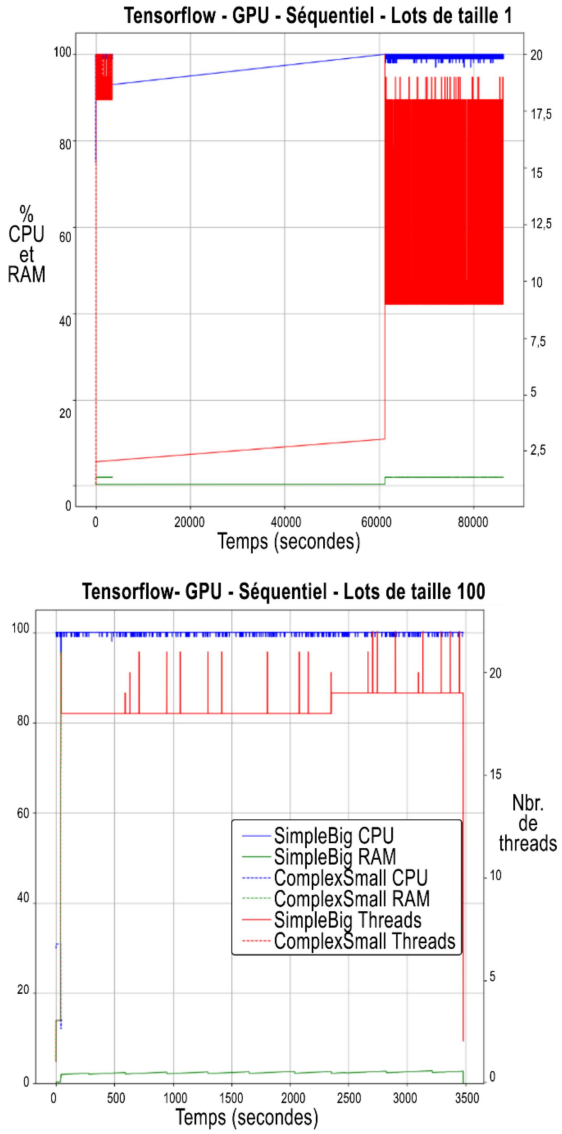

Figure 2 : Utilisation de la RAM, du CPU et nombre de fils d'exécution avec le framework TensorFlow.

La Figure 3 reprend la proportion de temps passé dans les fonctions CUDA dans le cas d'utilisation SimpleBig en mini-batch de pyTorch et de Tensorflow afin de mieux comprendre pourquoi ces *frameworks* diffèrent non seulement sur CPU mais aussi sur GPU. Ces graphiques ne reprennent que les fonctions les plus significatives, c'est-à-dire celles qui sont utilisées à au moins 1 % du temps. L'implémentation GPU via CUDA est principalement constituée de 3 fonctions pour Tensorflow :

cudaStreamCreateWithFlags, *cudaLaunchKernel* et *cudaFree*. Bien que l'information des paramètres ne soit pas disponible, Tensorflow crée des flux d'exécution asynchrones via *cudaStreamCreateWithFlags* étant donné qu'une méthode avec une signature plus simple existe si les flux doivent être synchrones. Par contre, cette méthode doit être de temps en temps appelée en synchrone et de temps en temps en asynchrone puisqu'il n'y a pas d'appel significatif explicite à une quelconque synchronisation. Les autres méthodes permettent d'exécuter des primitives ainsi que libérer la mémoire sur le GPU. En revanche l'approche de pyTorch est différente et s'occupe plus précisément de l'allocation mémoire sur le GPU. En effet les appels à *cudaMalloc* permettent de réserver de la ressource et de copier les données de manière asynchrone, c'est-à-dire en continuant à traiter les données présentes sur GPU. D'ailleurs pyTorch crée très peu de flux synchrones, uniquement ceux nécessaires à la synchronisation du modèle lors de sa mise-à-jour durant la phase d'entraînement. Une meilleure utilisation des ressources comme le CPU et une stratégie plus axée sur les transferts de données sont des clefs qui permettent à pyTorch de devancer Tensorflow.

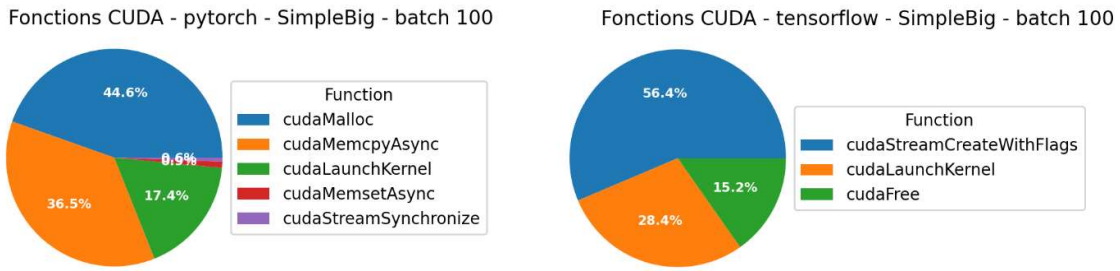

Figure 3 : Proportion de temps passé dans les fonctions CUDA.

5 Conclusion

Au cours de ce travail, nous avons comparé cinq *frameworks* d'apprentissage profond en termes d'utilisation des ressources et de temps de réponse sur quatre architectures CNN formées sur un petit et un grand ensemble de données. Nous avons conçu différentes configurations pour comprendre comment les *frameworks* fonctionnent et se comportent à la fois sur le CPU et le GPU. De notre analyse, nous avons montré que Singa était incapable de traiter un grand ensemble de données sans code supplémentaire car il nécessite un fil d'exécution par entrée. Paddle et MXNet se comportent mal lorsque le mini-batch est utilisé en raison d'erreurs d'allocation. Tensorflow et pyTorch sont les *frameworks* les plus stables de nos configurations.

Dans presque tous les cas, pyTorch surpasse les autres *frameworks* sur le CPU et surpasse également Paddle et Tensorflow sur le GPU. L'inconvénient de pyTorch est que ses besoins en mémoire sont beaucoup plus élevés que d'autres comme Tensorflow. Néanmoins, en ce qui concerne le mode mini-batch, pyTorch réduit drastiquement ses besoins en mémoire et surpasse Tensorflow pour le même coût de ressources.

Comme indiqué, pyTorch semble être le meilleur compromis entre stabilité et performances sur un seul nœud. Néanmoins, on sait que le calcul parallèle peut accélérer les algorithmes. Les tâches d'apprentissage en profondeur peuvent également être mises en parallèle en entraînant le modèle sur des mini-batches distincts tout en synchronisant le gradient. D'autre part, le coût de la synchronisation peut avoir un impact sur l'accélération de telle sorte que le calcul parallèle devient plus lent, surtout sur le GPU lorsque des données doivent être chargées dans sa mémoire. De la même manière, le calcul distribué est une piste lorsqu'un temps de réponse rapide est nécessaire. Les avantages et les inconvénients sont équivalents à ceux mentionnés dans ce travail mais peuvent être plus élevés que le calcul parallèle. En effet, chaque nœud de calcul peut se concentrer sur un sous-ensemble de données

d'entrée, ce qui signifie qu'il n'est pas nécessaire de charger et de décharger les données d'entrée dans la mémoire, tandis que la synchronisation par gradient nécessite des messages réseau plus lents que la synchronisation de la RAM pour le cas d'utilisation d'un seul nœud.

Les résultats de l'analyse ne dépendent pas de la version du langage Python[5] étant donné qu'il n'est que le langage utilisé pour écrire le code source mais qui sera exécuté de manière native. Les résultats ne peuvent changer que si l'un des *frameworks* change le cœur de son fonctionnement impliquant une refonte complète du code natif. Cela n'est pas prévu dans les feuilles de route de ces logiciels.

Dans le cadre de travaux futurs, nous souhaitons explorer le comportement de pyTorch dans ces configurations, mais aussi comprendre quand utiliser le calcul parallèle, le calcul distribué ou l'absence de parallélisme. Il est assez important de comprendre ces perspectives afin de concevoir une architecture informatique performante. Le Edge computing est un cas d'utilisation avec de nombreux appareils à faible capacité de calcul qui doivent se synchroniser de manière distribuée. Enfin, nous souhaitons concevoir un *framework* qui automatisera le déploiement de machines virtuelles ou de conteneurs Docker afin d'obtenir rapidement le résultat d'une tâche d'apprentissage profond. Une meilleure utilisation et une accélération des ressources sont nécessaires pour obtenir un temps de réponse rapide mais aussi pour minimiser le nombre de machines virtuelles, ce qui réduit le coût.

Remerciements

Pour mener à bien cette étude la Haute École en Hainaut a bénéficié d'un subside de la Fédération Wallonie-Bruxelles (JCM/TP/BS/mo/c999). L'auteur remercie le service ILIA de l'université de Mons qui a fourni des jeux de données ainsi que des commentaires qui ont permis d'améliorer la qualité du travail.

Références bibliographiques

Faust, K., Lima-Mendez, G., Lerat, J. S., Sathirapongsasuti, J. F., Knight, R., Huttenhower, C., Lenaerts, T. & Raes, J. (2015). Cross-biome comparison of microbial association networks. *Frontiers in microbiology* 6, 1200.

He, K., Zhang, X., Ren, S. & Sun, J. (2016). *Deep residual learning for image recognition*. *The IEEE conference on computer vision and pattern recognition*, 770-778. arXiv:1512.03385.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L. C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. *Proceedings of the IEEE conference on computer vision and pattern recognition*, 4510-4520.

Mayer, R. & Jacobsen, H. A. (2020). Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools. *ACM Computing Surveys (CSUR)* 53(1), 1-37.

Krizhevsky, A., Sutskever, I. & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25, 1097-1105.

Lerat, J. S., Han, T. A. & Lenaerts, T. (2013, August). Evolution of common-pool resources and social welfare in structured populations. In *Proceedings of the Twenty-Third international joint conference on Artificial Intelligence*, 2848-2854. Issu de : https://www.ijcai.org/Proceedings/13/Papers/419.pdf (consulté le 16/01/22).

[5] Réalisé dans la version 3.8.

Moritz, P., Nishihara, R., Stoica, I. & Jordan, M. I. (2015). *Sparknet: Training deep networks in spark*. arXiv preprint arXiv:1511.06051.

Ooi, B. C., Tan, K. L., Wang, S., Wang, W., Cai, Q., Chen, G., Gao, J., Luo, Z., Thung, A., Wang, Y., Xie, Z., Zhang, M. & Zheng, K. (2015). SINGA: A distributed deep learning platform. In *Proceedings of the 23rd ACM international conference on Multimedia*, 685-688, doi.org/10.1145/2733373.2807410.

Shanmugamani, R. (2018). *Deep Learning for Computer Vision: Expert techniques to train advanced neural networks using TensorFlow and Keras*. S. l.: Packt Publishing Ltd.

Shi, S., Wang, Q. & Chu, X. (2018, August). Performance modeling and evaluation of distributed deep learning frameworks on gpus. *In 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, 949-957. a rXiv:1711.05979.

Simonyan, K. & Zisserman, A. (2014). *Very deep convolutional networks for large-scale image recognition*. arXiv preprint arXiv:1409.1556.

Zhang, H., Zheng, Z., Xu, S., Dai, W., Ho, Q., Liang, X., Hu, Z., Wei, J., Xie, P. & Xing, E. P. (2017). Poseidon: An efficient communication architecture for distributed deep learning on clusters. *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17). July 12–14, 2017, Santa Clara, CA, USA*, 181-193. Issu de : https://www.usenix.org/system/files/conference/atc17/atc17-zhang.pdf (consulté le 16/01/22).