Ministère des Armées

Rapport de stage

Master MVA

Compression et accélération de l'inférence sur GPU des réseaux de neurones

29 | Septembre | 2019

Author

Samuel Hurault

1 Introduction

Les principaux traitements appliqués sur un signal de paroles (Speech Activity Detection, Language IDentification, Speaker IDentification...) reposent sur l'usage de classifieurs mis en œuvre par des réseaux de neurones profonds. Le traitement de la parole commence par la transformation du signal audio en sa représentation spectrale 2D. Cette transformation est schématisée figure 1. Après cette transformation, la problématique devient une problématique de traitement d'image. C'est pourquoi nous nous intéressons dans ce travail à l'accélération et à la compression des réseaux de neurones convolutifs.

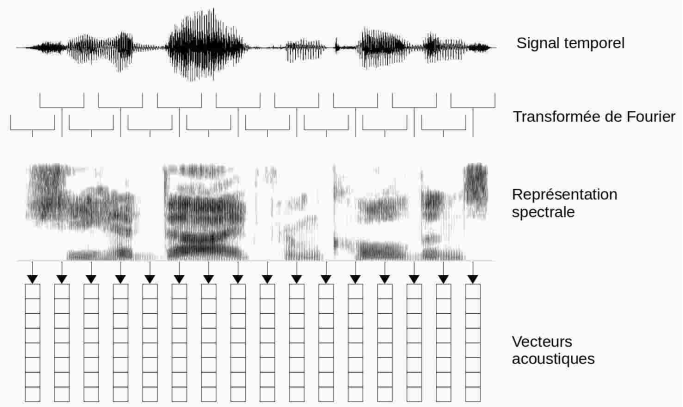


FIGURE 1 – De la parole à l'image

Les compétitions comme ILSVRC [Russakovsky et al., 2015] ont permis la construction de réseaux convolutifs de plus en plus performants en terme de précision. Cependant ces réseaux ont parfois été imaginés sans faire attention à la complexité du modèle ou à son temps d'exécutions. Les performances de réseaux à centaines de couches et millions de paramètres sont possibles au prix de grandes capacités de calculs, temps et mémoire.

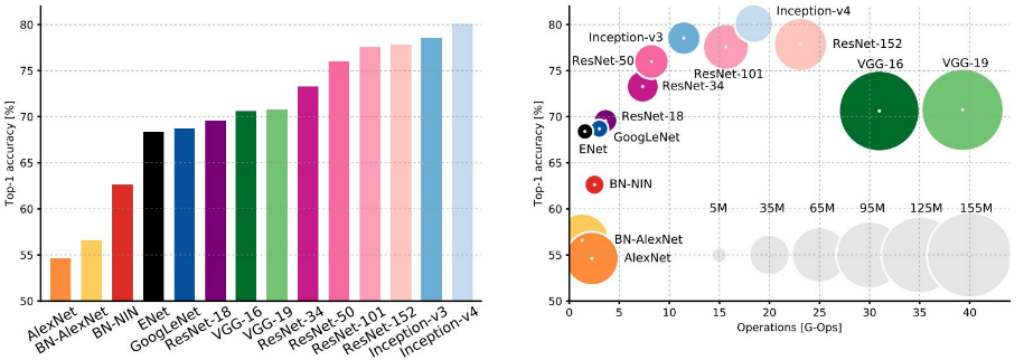


FIGURE 2 – Accuracy et taille de quelques réseaux de neurones convolutifs classiques.

La figure 2, obtenu dans [Canziani et al., 2016], représente, pour quelques réseaux convolutifs classiques pour la base de donnée ImageNet, l'accuracy (pourcentage de bonnes prédictions) en fonction du nombre d'opérations (FLoating Point Operations) nécessaires lors de l'inférence, et du nombre de paramètres du réseau. Les réseaux les plus efficaces à la fois en termes de de précision, nombre d'opérations, et nombre de paramètres comme Inception, GoogleLeNet ou ENet ont été construits avec des architectures très spécifiques, et adaptés à la tâche de classification d'images ImageNet. Les réseaux VGG ou ResNet ont une architecture plus générale et sont plus intéressants à étudier.

Nous avons d'abord réalisé une analyse bibliographique des méthodes de compression et d'accélération de l'inférence des réseaux de neurones profonds [Hurault, 2019]. Dans cette analyse, trois grandes méthodes ont été abordées : le pruning, la quantification et la distillation. Pour ces trois méthodes nous avons présenté un ensemble de démarches et de résultats présents dans la littérature. Nous allons à présent implémenter et tester une partie de ces méthodes sur nos propres réseaux, avec notre propre matériel. Le but sera d'avoir une idée des performances de compression et d'accélération que l'on peut espérer atteindre pour différentes tâches. Nous nous concentrerons sur des réseaux convolutifs appliqués aux tâches du traitement d'images et du traitement de la parole. Après avoir présenté ces réseaux, nous présenterons les expériences réalisées et les performances obtenues pour le pruning, la quantification et enfin la distillation. Pour chaque méthode, nous analyserons séparément les performances métier (précision en accuracy du réseau) et les performances d'accélération (temps de l'inférence du réseau).

2 Modèles et bases de données

Nous présentons ici les différents modèles qui seront utilisés à travers les expériences mises en œuvre. Nous traitons deux tâches différentes : le traitement d'image et le traitement de la parole. Les modèles évalués sont les modèles convolutifs suivants :

- Traitement d'image : ResNet50 entraîné sur base ImageNet, ResNet56 et VGG19 entraînés sur base CIFAR10.
- Traitement de la parole : Un réseau construit pour la tâches de détection de segments parlés sur un signal audio (Speech Activity Detection), que l'on appellera "SAD" et un autre réseau pour la tâche d'identification de la langue (Language IDentification), appelé "LID".

2.1 Bases de données

Les informations des bases de données utilisées sont présentées tableau 1.

Base de donnée	Nb d'images	Nb de catégorie	Taille de l'entrée
ImageNet	$> 10^6$	1000	$224 \times 224 \times 3$
CIFAR10	60 000	10	$32 \times 32 \times 3$
SAD	9000	2	13×41
LID	60000	5	300×80

TABLE 1 – Bases de données utilisées

2.2 Réseaux

2.2.1 Réseau VGG19

Le réseaux VGG19, représenté figure 3 est un réseau convolutif dont l'architecture est classique. Il est constitué de 16 couches convolutives successives puis de 3 couches de type "fully-connected". Il sera utilisé sur la base CIFAR10.

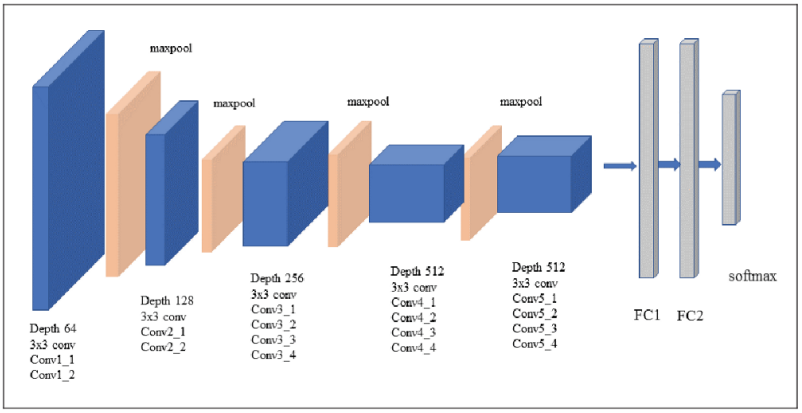


Fig. 3. VGG-19 network architecture

FIGURE 3 – VGG19

2.2.2 Réseaux ResNet

Les modèles ResNet ([He et al., 2015]) sont des réseaux convolutifs classiquement utilisés dans la littérature de compression et d'accélération des réseaux de neurones. Les réseaux résiduels ont permis d'entraîner des réseaux de plus en plus profonds. En effet, l'ajout de couches peut dégrader les performances à cause de l'extinction de gradient. Pour résoudre ce problème, l'idée fut d'ajouter, entre plusieurs couches convolutives, une connexion appelée 'raccourci de l'identité' ("Identity shortcut connection"), représentée figure 4. Avec les notations de la figure, en notant y la sortie, la fonction $\mathcal{F}$ apprend ainsi le résiduel $y-x$. Cet ajout ne dégrade pas les performances du réseau car il est beaucoup plus facile d'apprendre $\mathcal{F} = 0$ plutôt que $\mathcal{F} = x$.

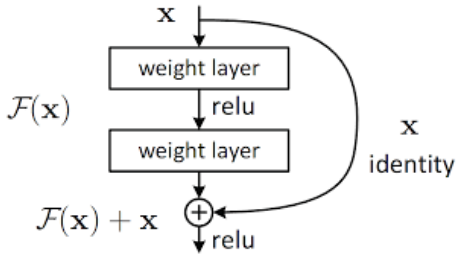


FIGURE 4 – Projection architecture ResNet

Nous utilisons deux réseaux ResNet :

— ResNet56 entraîné sur CIFAR10 (représenté figure 5) est constitué de 56 couches. Toutes les convolutions sont regroupées par deux entre des connections 'shortcut', comme sur figure 4. Afin de faire correspondre les dimensions entre l'entrée et la sortie de la connection 'shortcut', les bords de x sont mis à zéro.

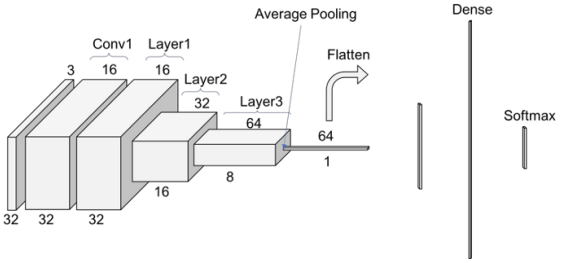


FIGURE 5 – ResNet56

— ResNet50 entraîné sur ImageNet est constitué de 50 couches pour 25.5M de paramètres. Cette fois, comme représenté figure 6, certaines connections "shortcut" ne sont plus de simples identités, mais contiennent une couche convolutionelle avec une fonction d'activation de type ReLU.

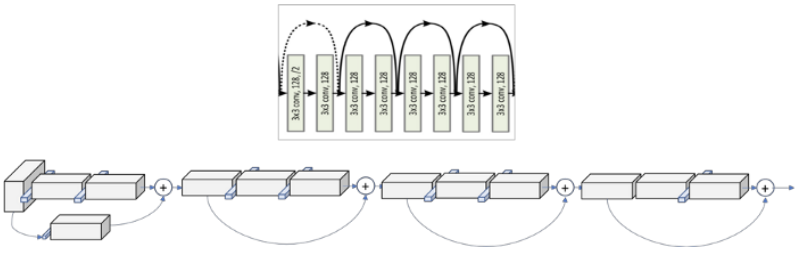


FIGURE 6 – ResNet50

2.3 Traitement de la parole : SAD et LID

Les réseaux de traitement de la parole SAD et LID prennent en entrée des spectrogrammes MFCC pré-calculés pour chaque tâche. Les dimension fréquentielles et temporelles définissent une entrée 2D (image).

- Le réseau SAD est constitué de trois couches convolutives 1D avec successivement 15, 27 et 164 canaux par couches et des noyaux de longueur 10,15 et 16 , suivis d'une couche fully-connected.
- Le réseau LID est constitué de trois couches convolutives 1D avec successivement 1024, 1024 et 128 canaux par couches et des noyaux de longueur 5, suivis de deux couches fully-connected.

Nous présentons tableau 2, le nombre de paramètres de chaque réseau.

Réseau	**Nb paramètres**
ResNet50 ImageNet	$2,56.10^7$
ResNet56 CIFAR10	$8,57.10^5$
VGG19 CIFAR10	$2,01.10^7$
SAD	$8,01.10^4$
LID	$6.34.10^6$

TABLE 2 – Taille des réseaux utilisés en nombre de paramètres

3 Pruning d'un réseau convolutif

Dans cette partie nous implémentons les principaux algorithmes de compression de réseaux convolutifs par "pruning". Ces algorithmes sont présentés dans le détail dans l'analyse bibliographique [Hurault, 2019]. Nous rappelons que le pruning suit le processus suivant :

1. Entraîner un réseau donné.
2. Évaluer et retirer les connections les moins importantes.
3. Ré-entraîner pour affiner les paramètres.
4. Répéter les deux étapes précédentes.

Nous utilisons comme code de base les implémentations Pytorch "Distiller" d'Intel [Zmora et al., 2018]. Tous les réseaux utilisés ont déjà été pré-entraînés.

"Supprimer" une connection signifie mettre à zéro le poids, ou l'ensemble des poids correspondants. On appellera niveau de parcimonie s la fraction (ou le pourcentage) de poids mis à zéro. On rappelle que le pruning sert à la compression et à l'accélération des réseaux. Le niveau de parcimonie obtenu correspondra finalement à un certain niveau de compression et/ou d'accélération. On pourra alors, pour un niveau de parcimonie donné, étudier les performances métier, d'accélération et de compression du modèle.

Lors de cette analyse, nous ne considérons que le pruning des couches convolutives. Malgré que les couches "fully-connected" en fin de réseaux contiennent de nombreux paramètres, le pruning de ces couches n'accélère pas l'inférence.

Pour une couche dont le niveau de parcimonie est passé de 0 à s, le taux de compression obtenue est $\frac{Nb\ paramtres\ rseau\ initial}{Nb\ paramtres\ rseau\ final} = \frac{1}{1-s}$. La courbe représentant ce taux de compression en fonction du niveau de parcimonie est représenté figure 7.

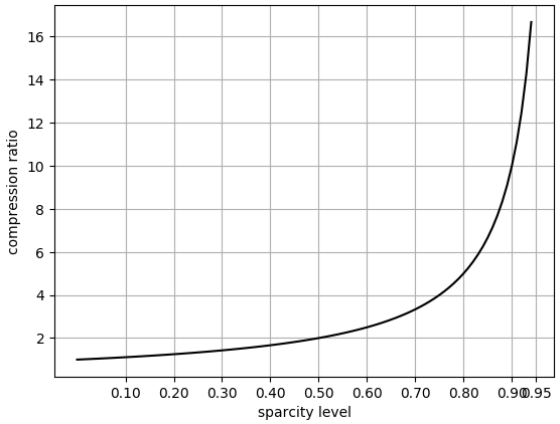


FIGURE 7 – Taux de compression en fonction du niveau de parcimonie obtenu après pruning.

Pour toutes les expériences de pruning réalisées, nous fixons antérieurement, pour chaque couche, le niveau de parcimonie désiré. Ce type de traitement correspond à la section "Predefined architecture pruning" de l'analyse bibliographique [Hurault, 2019].

3.1 Performances métier

Le but de cette section est d'étudier les performances métier de la plupart des méthodes de pruning introduites dans l'analyse bibliographique [Hurault, 2019]. Le pruning sera réalisé à trois échelles différentes :

- pruning 1D ("element pruning") : suppression de poids scalaires individuels.
- pruning 2D ("filter pruning") : suppression de filtres.
- pruning 3D ("channel pruning") : suppression de canaux (ensemble de filtres).

A ces différentes échelles, nous utilisons les métriques suivantes d'évaluation des poids :

- choix aléatoire ("Random"). Elle permet de nous assurer que les autres méthodes sont bien efficaces.
- par magnitudes $L1$ et $L2$.
- par l'analyse du gradient ("Gradient"). Cette méthode était appelée "First order Taylor expansion" dans l'analyse bibliographique [Hurault, 2019].

Pour différentes valeurs de parcimonies s, nous appliquons le traitement suivant :

1. Couche par couche, évaluation et classement des poids selon la méthode choisie.
2. Une fraction s de poids est supprimée. Notons que pour le traitement 3D, supprimer un canal à la couche l affecte la structure à la couche $l+1$.
3. Ré-apprentissage des poids du réseau non affectés par l'opération.

3.1.1 Performances du pruning sur le réseau Resnet56 entraîné sur CIFAR10

Nous testons d'abord les performances de pruning sur le réseau ResNet56 préalablement entraîné sur la base CIFAR10. Les valeurs de parcimonie s testées sont $\{0.2, 0.4, 0.6, 0.8, 0.9\}$. Le ré-apprentissage s'effectue sur 10 époques (ce choix sera discuté en 3.1.2) avec un taux d'apprentissage ("learning rate") 1/10 du learning rate de l'apprentissage de référence. Les autres paramètres d'entraînement sont inchangés (batch size, early stopping, weight decay, momentum..). Le ré-apprentissage est réalisé sur les 50000 images de la base de données d'apprentissage, les performances analysées ci-dessous sont celles obtenues sur la base de donnée de test de 10000 images.

Les performances sont reportées tableau 3 et figure 8. En 8a et 8b sont respectivement présentées les résultats de pruning par filtre et par canal, et en 8c seulement le pruning $L1$.

Tout d'abord, comme attendu, toutes les méthodes sont plus efficaces que le pruning aléatoire. Les meilleures performances sont permises par les pruning $L1$ et $L2$. Le pruning par parcimonie non structurée est le plus efficace : il est possible de mettre à zéro 90% des poids scalaires avec une perte de seulement 3.2% d'accuracy. Lorsque la parcimonie est structurée, les performances se dégradent à partir de 60% des filtres ou canaux retirés. On peut noter que les performances de précision lors de la suppression de filtres ou de canaux sont similaires. En effet, en observant en détail la structure du réseau obtenu lors du pruning par filtre, on voit que lorsqu'un filtre est supprimé, les filtres de son canal sont souvent également retirés.

3.1.2 Ré-apprentissage itératif et limites du pruning

Comme indiqué dans l'analyse bibliographique [Hurault, 2019], il est courant d'utiliser dans la littérature un schéma de pruning et ré-apprentissage itératif. On répète l'opération précédente en augmentant progressivement les parcimonies des couches. La boucle s'arrête lorsque le taux de parcimonie désiré s dans chaque couche est atteint.

Nous testons deux méthodes d'évolution du pourcentage de connections à supprimer à chaque itération.

— Évolution linéaire : Sur n itérations, à chaque étape $\frac{s}{n}\%$ des éléments sont supprimés.
— Évolution non linéaire avec la méthode introduite par [Michael H. Zhu, 2018], et décrite dans [Hurault, 2019]. A l'itération $i \in 1, ..., n$ le taux de parcimonie suit

Parcimonie	**0**	**0.2**	**0.4**	**0.6**	**0.8**	**0.9**
L1 Element	93.39	**93.45**	**93.26**	**93.12**	**92.21**	**90.16**
L1 Filter	93.39	92.44	91.25	89.22	81.88	71.88
L1 Channel	93.39	92.98	92.13	89.89	81.15	66.45
L2 Filter	93.39	92.44	91.25	89.22	81.88	71.88
L2 Channel	93.39	92.84	92.24	89.72	83.68	72.54
Gradient Filter	93.39	92.88	91.09	78.89	26.34	23.51
Gradient Channel	93.39	92.9	91.84	36.09	29.69	15.43
Random Filter	93.39	75.43	37.62	11.73	25.42	24.75
Random Channel	93.39	73.12	36.61	31.64	22.99	23.59

TABLE 3 – Top-1 accuracy après pruning et ré-apprentissage sur 10 époques, pour différentes valeurs de parcimonie et différentes méthodes du réseau ResNet56 entraîné sur CIFAR10. La meilleure méthode est celle du pruning $L1$ appliqué sur chacun des poids. Les courbes associées sont présentées figure 8.

$s_i = s(1-(1-\frac{i}{n})^3)$. L'idée est de retirer beaucoup de connections lors des premières itérations puis de moins en moins ensuite. Cette méthode sera appelée "AGP" (Automatic Gradual Pruner) dans la suite.

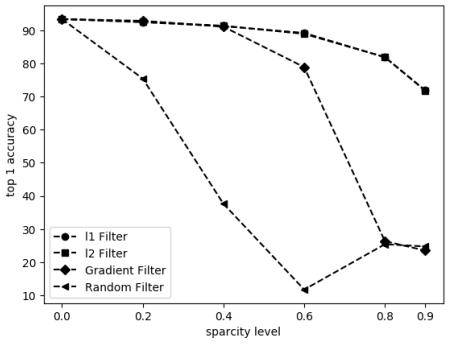
Nous testons une nouvelle fois les performances sur le réseau ResNet56 entraîné sur CIFAR10. La parcimonie finale désirée est $s = 90\%$. Nous proposons ces résultats pour deux méthodes : Pruning par magnitude $L1$ aux échelles du scalaire ("L1 Elements") et du canal ("L1 Channels"). Nous fixons le nombre $n = 10$ d'itérations et nous faisons varier le nombre d'époques de réapprentissage à chaque itérations : 1, 2, 5, 10 et 100. Pour l'évolution non linéaire "AGP", le nombre d'itérations est de $n = 100$ et la parcimonie finale désirée est de 90%. Les résultats obtenus sont présentés figure 9.

En complément, la figure 11 représente les courbes d'apprentissage lors du pruning itératif avec 100 époques/it.

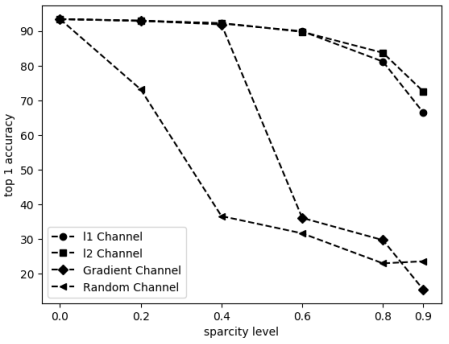
Nous affichons également figure 10 les courbes d'évolution de la parcimonie pour les méthodes linéaires et AGP.

En augmentant le nombre d'époques de ré-apprentissage, les performances continuent d'augmenter de manière notable. Plus le degré de parcimonie est élevé, plus cette augmentation est importante. Le réseau initial a été entraîné sur 200 époques. On observe figure 11 pour des degrés de parcimonies faibles (jusqu'à 70%, c'est-à-dire les 7 premières itérations) ré-apprendre sur 100 époques est suffisant. Cependant, pour des degrés de parcimonie plus forts (80% et 90%, c'est-à-dire les deux dernières itérations), les courbes de loss atteignent à peine l'asymptote de convergence. Cette observation rejoint celles réalisées dans [Michael H. Zhu, 2018] qui propose l'augmentation non linéaire de la parcimonie précédemment testée : sur 100 époques, elle n'améliore cependant pas les performances.

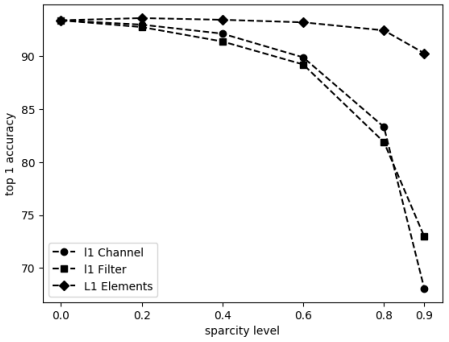
Ainsi il faut un très long ré-apprentissage pour obtenir des performances optimales sur



(a) A l'échelle du filtre



(b) A l'échelle du canal

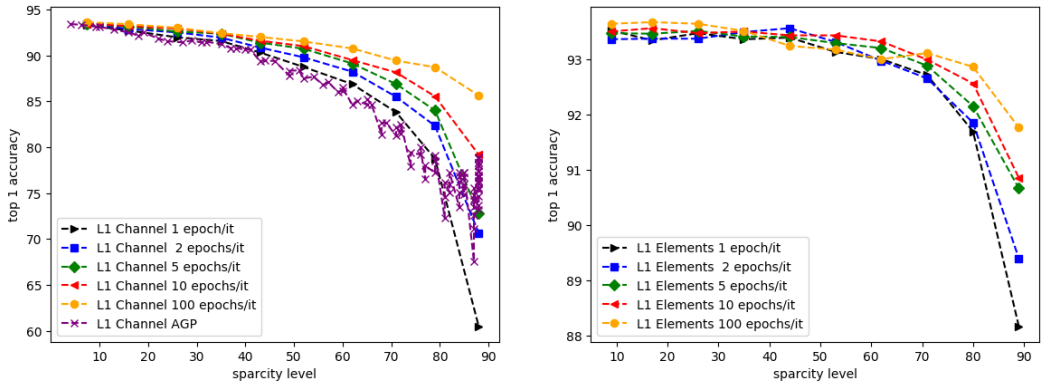


(c) Pruning par magnitude $L1$

FIGURE 8 – Top-1 accuracy après pruning et ré-apprentissage sur 10 époques, pour différentes valeurs de parcimonie et différentes méthodes du réseau ResNet56 entraîné sur CIFAR10. Les résultats chiffrés sont donnés tableau 3.

un réseau qui a subit un pruning dur (plus de 80% de parcimonie). Durant celui-ci, les fonctions des différents filtres sont probablement totalement ré-apprises. Ces résultats nous amènent à suivre les expériences réalisées par [Qin et al., 2018] : nous utilisons le réseau qui a subit le pruning itératif et qui a perdu 90% de ses canaux. Nous conservons la même architecture et ré-apprenons ce réseau avec des poids initialisés aléatoirement, sur 200 époques. Nous comparons les résultats du pruning itératif et sur cet apprentissage figure 12. Nous observons que les performances obtenues sont très proches pour les deux expériences.

Ces observations remettent en perspectives les performances métiers du pruning. Pour des niveaux de parcimonie suffisamment faibles (moins de 70% de parcimonie pour le pruning 3D, 80% pour le pruning 1D), le ré-apprentissage sur un faible nombre d'époques permet de compenser et de retrouver des performances optimales. Dès que le niveau de parcimonie devient trop important, il est alors aussi efficace d'apprendre directement un réseau similaire à l'architecture compressée.



(a) A l'échelle du canal (b) A l'échelle scalaire

FIGURE 9 – Top-1 accuracy après pruning et ré-apprentissage itératif de Resnet56 entraîné sur CIFAR10, en faisant varier le nombre d'époques de ré-apprentissage

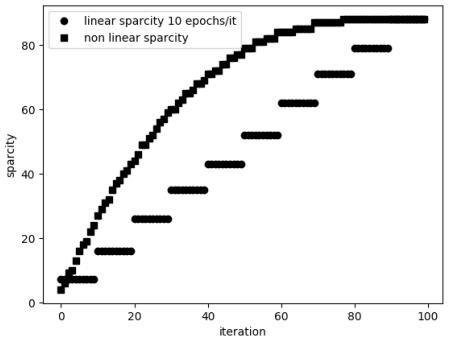
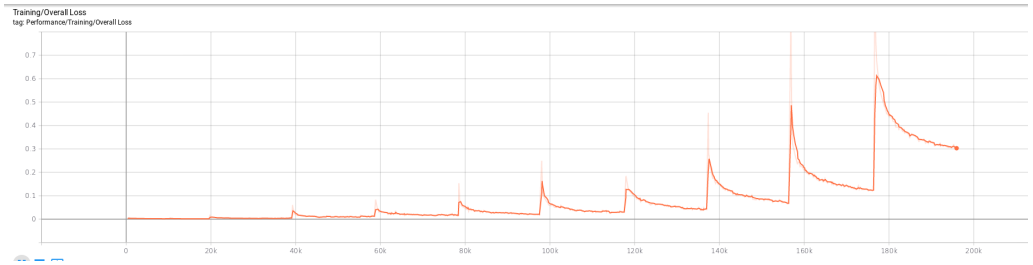


FIGURE 10 – Evolution de la parcimonie pendant le pruning et ré-apprentissage itératif. Nous comparons ici les deux méthodes d'évolution de parcimonie : linéaire et AGP.

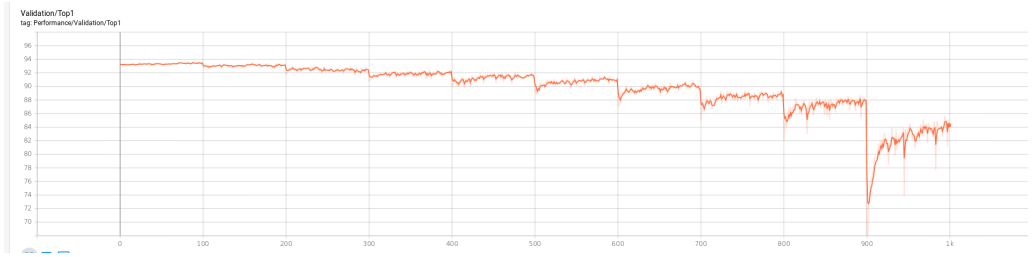
3.2 Performances d'accélération

Afin de profiter du pruning pour accélérer l'inférence et sans matériel spécialisé pour l'accélération du calcul de matrices creuses, il est nécessaire réaliser du pruning structuré à l'échelle du canal. L'implémentation de Distiller [Zmora et al., 2018] choisit de retirer les connections grâce à un masque qui remplace les poids par des zéros. L'architecture du réseau reste inchangé, l'inférence n'est donc pas accélérée. Afin de mesurer les performances d'accélération, à la fin du pruning, nous redéfinissons les architectures de chaque réseau pour supprimer les canaux mis à zéros. Supprimer un canal (et sa carte d'activation correspondante) sur la couche l entraîne aussi la suppression d'une dimension pour chaque canal de la couche $l+1$, comme illustré figure 13.

— **Accélération théorique d'une couche convolutive**



(a) Loss d'apprentissage



(b) Loss de validation

FIGURE 11 – Courbes d'apprentissage lors du pruning et ré-apprentissage itératif canal par canal et pour 100 époques/it.

Les performances d'accélération peuvent être prédites par le nombre de FLOP (Floating Point OPerations) lors de l'inférence sur le réseau traité, indépendamment du matériel utilisé. Nous utilisons les notations introduites en analyse bibliographique [Hurault, 2019] et présentées figure 13. Supposons que le taux de parcimonie de la couche l est P^l. $E(N^l \times P^l)$ filtres seront supprimés. Où $E(x)$ correspond à la partie entière de x. Dans la suite, par soucis de simplicité, nous nous abstenons de la partie entière et considérons $N^l \times P^l$ directement. Les résultats peuvent donc être légèrement différents de ceux présentés tableau 4. Notons les tailles des cartes d'activations d'entrée et de sortie respectivement $H^{l-1} \times W^{l-1}$ et $H^l \times W^l$. Après pruning, la dimension des cartes d'activations passe de $N^l \times H^{l-1} \times W^{l-1}$ à $N^l(1-P^l) \times H^l \times W^l$. A la couche suivante $l+1$, il restera $N^{l+1}(1-P^{l+1})$ filtres de dimension $N^l(1-P^l) \times w^{l+1} \times h^{l+1}$. La convolution à la couche l requiert $N^l H^l W^l C^{l-1} w^l h^l$ FLOP. Après pruning de l et $l+1$, il reste $N^l(1-P^l)H^l W^l C^{l-1} w^l h^l$ FLOP à la couche l et $N^{l+1}(1-P^{l+1})H^{l+1}W^{l+1}N^l(1-P^l)w^{l+1}h^{l+1}$ FLOP à la couche $l+1$. Ainsi une proportion $1-(1-P^{l+1})(1-P^l)$ des calculs (FLOP) sont supprimés à la couche $l+1$, ce qui donne en théorie les résultats du tableau 4. Ces résultats sont vrais si et seulement si deux couches successives sont traitées avec le même niveau de parcimonie.

Parcimonie (en %)	**20**	**40**	**60**	**80**	**90**
Réduction des FLOP (en %)	36	64	84	96	99

TABLE 4 – Pourcentage de FLOP supprimé lors du pruning d'une couche convolutive.

— **Accélération réelle de l'inférence**

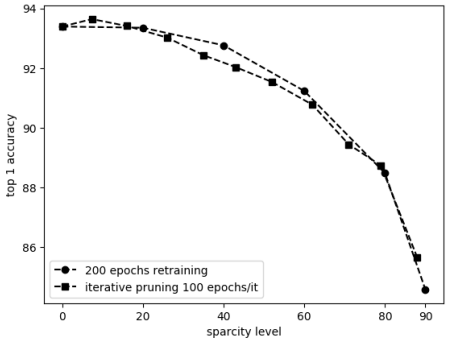


FIGURE 12 – Pruning et ré-apprentissage itératif de Resnet56 ou apprentissage direct de ce réseau avec 90% des canaux supprimés.

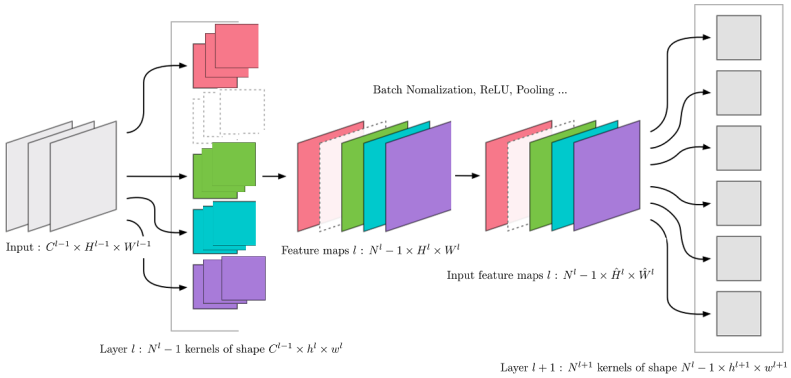


FIGURE 13 – Pruning d'un canal sur une couche de convolution.

Cependant les temps d'accélération théoriques et réels sur GPU peuvent être assez différents. Les stratégies du calcul BLAS influencent grandement les temps d'inférence. D'autres opérations telles que la normalisation des batchs ou le pooling sollicitent aussi du temps de calcul GPU. Pour l'inférence Pytorch, avant l'inférence, le réseau est envoyé sur la mémoire hôte du GPU. Ensuite, pour chaque batch :

1. Les données d'entrée sont transférées depuis le CPU vers le GPU.
2. Le réseau exécute l'inférence.
3. La sortie est récupérée depuis le GPU vers le CPU.

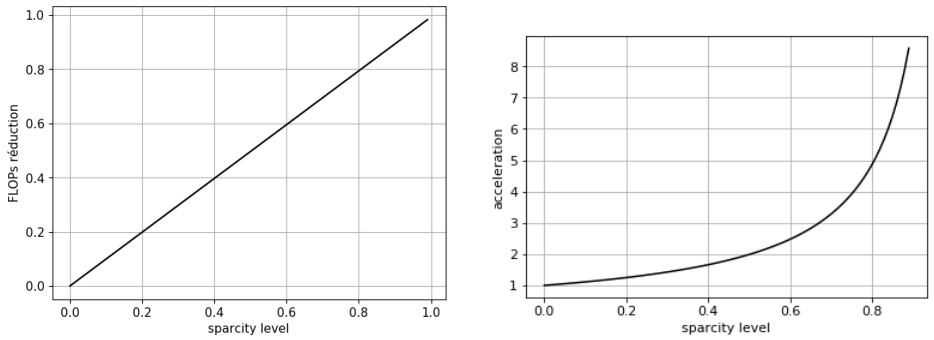
Par défaut, PyTorch exécute ces étapes en mode asynchrone. Dès l'étape 1), plusieurs streams travaillent en parallèle. Alors que toutes les données ne sont pas encore forcément transférées, l'étape 2) commence et le GPU réalise des calculs sur certains streams. Lors de l'étape de retour au CPU du résultat, il faut attendre que tous les streams aient fini leur traitement pour récupérer le résultat. l'exécution asynchrone sera donc plus rapide que l'exécution synchrone. Nous nous attendons à ce que la taille des batchs joue un rôle important pour trouver un débit optimal.

Il est possible de récupérer les temps de latence de chaque étape. On appelle "temps de transfert" la somme des latences des étapes 1) et 2). On appelle "temps d'inférence" le temps de l'étape 2). Pour une exécution asynchrone, on s'attend à ce que le temps d'inférence soit constant, indépendamment de la taille de l'entrée, ou de la charge de calculs. Ces paramètres affecteront le "temps de transfert" de l'étape 3). Pour mieux comparer l'accélération réelle sur GPU permise par le pruning, nous présenterons également les latences obtenues avec une exécution synchrone. Ce sera cette fois bien le temps de latence GPU qui sera réduit. Nous mesurerons également l'accélération de l'inférence sur CPU, indépendant de ces considérations.

Pour toutes les expériences suivantes, la métrique de mesure temporelle est le débit de l'inférence en nombre d'images traitées par seconde. Ce temps global dépend de la taille de batch utilisée. Les expériences seront réalisées pour différentes tailles de batchs ("batch size") et différents degrés de parcimonie.

3.2.1 Pruning de l'architecture ResNet

Concernant la projection de l'identité sur l'architecture ResNet (représentée figure 4), les sorties $F(x)$ et x doivent avoir la même dimension. Il n'est donc pas possible de supprimer des filtres de la convolution précédant la projection sans modifier la dimension de l'entrée x et inversement. L'accélération de cette convolution n'est donc pas possible et les filtres sélectionnés pour le pruning supprimés sont seulement mis à zéro. Ainsi une proportion s des FLOP sont supprimés à chaque couche traitée, où s est le degré de parcimonie du réseau. La figure 14 présente, pour une couche convolutive traitée de l'architecture ResNet56, en fonction du degré de parcimonie souhaité, le pourcentage de FLOP supprimés (14a) ainsi que le facteur d'accélération théorique obtenu (14b).



(a) FLOP supprimés (b) accélération théorique

FIGURE 14 – Résultats théoriques du pruning de l'architecture ResNet56 sur CIFAR10

Les performances réelles d'accélération obtenues sur GPU sont présentées tableau 5 et figure 15. Les tests d'inférence sont mesurés sur 100 batchs pour des tailles de batchs

32, 64, 128, 256, 512. Les valeurs du tableau correspondent aux débit moyenné sur les 100 batchs. On représente également les intervalles de confiances à 95% sur le graphique figure 15.

Parcimonie	**0**	**0.2**	**0.4**	**0.6**	**0.8**	**0.9**
batch size 32	2685	2027	1974	2400	2455	2733
batch size 64	4499	4939	4377	4239	4749	5246
batch size 128	7782	7597	8461	9273	9621	10219
batch size 256	8337	8800	9458	11118	12434	13333
batch size 512	8249	8914	9597	11208	12575	12974

TABLE 5 – Débit GPU (en images/sec) après le pruning pour différentes valeurs de parcimonie et différentes tailles de batch d'entrée. La figure associée

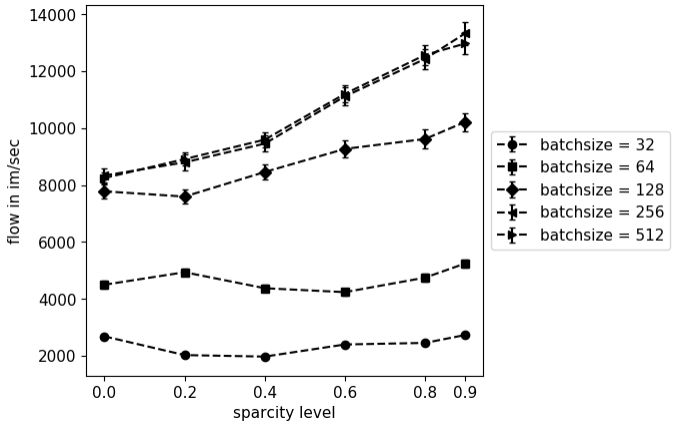


FIGURE 15 – Débit GPU (en images/sec) après le pruning de Resnet56 pour différentes valeurs de parcimonie et différentes tailles de batch d'entrée.

On observe que l'accélération réelle sur GPU est limitée. A partir d'une taille de batch 256, le débit devient constant. Les performances ne sont pas aussi bonnes que celles espérées par analyse théorique, pour les raisons déjà expliquées précédemment. Il est cependant possible, avec une parcimonie à l'échelle du canal de 80%, d'accélérer l'inférence d'un facteur 1.5.

3.2.2 Performances de VGG19 entraîné sur CIFAR10

L'accélération par pruning des réseaux ResNet étant limitée, nous testons également les performances de pruning sur le réseau VGG19.

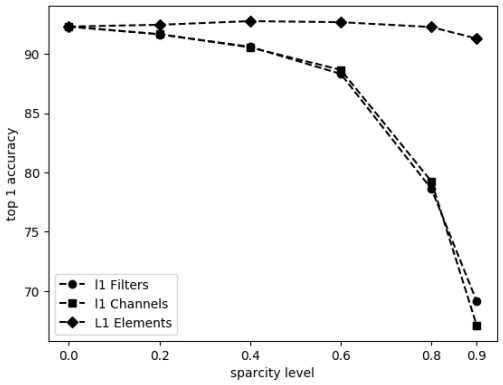
— **Performances de précision**

Comme précédemment, les premières et dernières couches ne sont pas traitées. Nous considérons seulement le pruning $L1$ aux trois échelles possibles, avec un ré-apprentissage sur

10 époques. Tout d'abord, les performances métier sont présentées tableau 6 et figure 16. Il est possible de supprimer 80% des poids scalaires sans perte de performances et 60% des canaux avec une perte de moins de 4% d'accuracy. Les performances de pruning des canaux se dégradent ensuite. Nous ne réalisons pas d'étude détaillée des performances métier comme celle faite plus tôt sur le réseau ResNet56, nous nous concentrons ici sur les performances d'accélération.

Parcimonie	**0**	**0.2**	**0.4**	**0.6**	**0.8**	**0.9**
L1 Element	92.32	92.47	92.78	92.69	92.28	91.3
L1 Filter	92.32	91.66	90.61	88.32	78.6	69.11
L1 Channel	92.32	91.68	90.56	88.66	79.24	67.07

TABLE 6 – Top-1 accuracy après le Pruning pour différentes valeurs de parcimonie et différentes méthodes du réseau VGG19 entraîné sur CIFAR10.



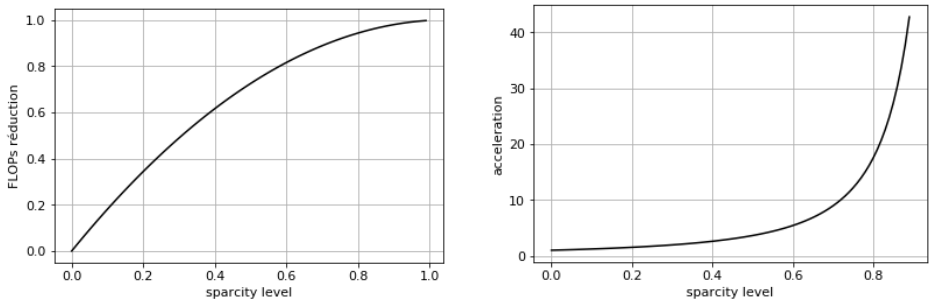
(a) Pruning par magnitude $L1$

FIGURE 16 – Top-1 accuracy après le Pruning pour différentes valeurs de parcimonie et différentes méthodes du réseau VGG19 entraîné sur CIFAR10.

— **Performances d'accélération**

Les performances théoriques de réduction de FLOP et d'accélération maximale sont présentées figure 17. En supprimant 60% des canaux dans chaque couche, on peut retirer 80% des calculs, ce qui permet, idéalement, d'accélérer l'inférence dans chaque couche convolutionelle d'un facteur d'environ 5. Ces performances sont d'abord à comparer avec l'inférence sur CPU du réseau global dont les résultats sont présentés figure 18. Sur CPU, l'accélération réelle des calculs atteint un facteur d'environ 3 en supprimant 60% des canaux.

Les performances réelles d'accélération de l'inférence sur GPU sont présentées tableau 7 et figure 19. On rappelle qu'il est possible de supprimer 80% des poids sans perte de performance. A ce niveau de parcimonie, sur GPU, l'inférence de VGG19 peut-être accélérée d'un facteur 5. On atteint donc cette fois les performances théoriques.



(a) FLOPs supprimés (b) accélération théorique

FIGURE 17 – Résultats théoriques du pruning de l'architecture VGG19 sur CIFAR10

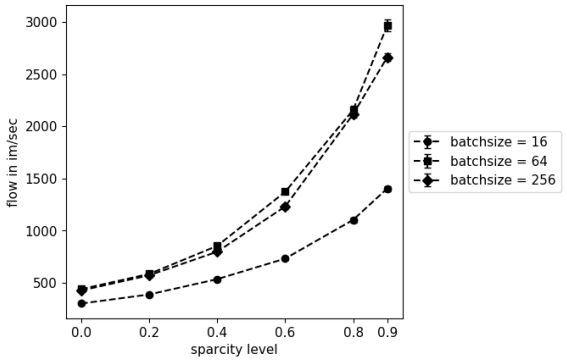


FIGURE 18 – Débit (en images/sec) sur CPU après le pruning de VGG19 pour différentes valeurs de parcimonie et différentes tailles de batch d'entrée.

Pour interpréter ces résultats, nous présentons figure 20 les latences de transferts mémoires GPU-CPU et d'inférence sur GPU. Ces temps sont présentés pour des inférences en mode synchrone et asynchrone. On remarque d'abord, comme attendu, que le temps cumulé est globalement plus court en mode asynchrone qu'en mode synchrone. En mode synchrone, le temps de transfert est bien dominé par le temps d'inférence. Pour une taille de batch 256, ce temps d'inférence est divisé par facteur d'environ 3 pour une parcimonie de 80%. Cela correspond bien aux résultats obtenus sur CPU.

Cependant, les performances d'accélérations de pruning sont meilleures pour l'exécution asynchrone : le temps de transfert est cette fois divisé par un facteur d'environ 5 !

3.3 Pruning des réseaux de traitement de la parole

Nous reproduisons les expériences précédentes en traitement de la parole, avec les réseaux SAD et LID.

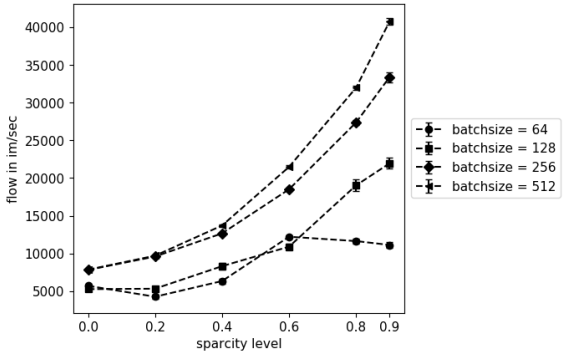


FIGURE 19 – Débit (en images/sec) après le pruning de VGG19 pour différentes valeurs de parcimonie et différentes tailles de batch d'entrée.

Parcimonie	**0**	**0.2**	**0.4**	**0.6**	**0.8**	**0.9**
Batch size 64	5703	4252	6333	12214	11639	11124
Batch size 128	5260	5319	8307	10896	19028	21952
Batch size 256	7812	9585	12659	18511	27370	33394
Batch size 512	7843	9699	13703	21481	31982	40774

TABLE 7 – Débit (en images/sec) après le pruning de VGG19 pour différentes valeurs de parcimonie et différentes tailles de batch d'entrée.

3.3.1 Performances de précision

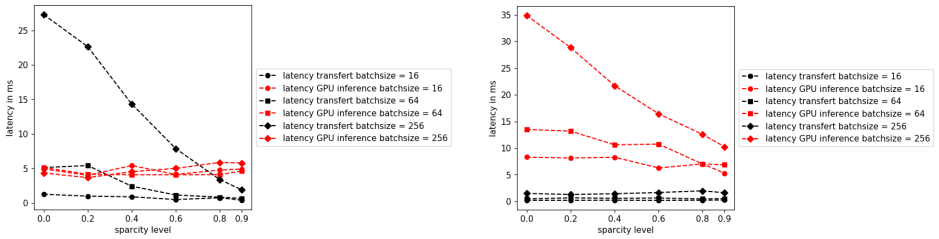
Pour les deux tâches, une seule époque de ré-apprentissage sont nécessaires pour converger. Les résultats sont reportés figure 21 et tableau 8.

Pour les deux réseaux SAD et LID, avec du pruning scalaire, il est possible de retirer au moins 95% des poids sans perte de performance. Avec le réseau LID, on peut même aller jusqu'à 99% de parcimonie scalaire. Avec de la parcimonie structurée par canal, sur le réseau SAD, on peut retirer jusqu'à 80% des canaux en perdant seulement 1.5% d'accuracy. Cependant, le pruning par canal dégrade rapidement les performances du réseau LID.

3.3.2 Performances d'accélération

Les performances d'accélération des inférences sur GPU et CPU des réseaux SAD et LID sont respectivement présentées figures 22 et 24.

— Pour le réseau SAD, nous n'observons pas d'accélération de l'inférence sur GPU. Pour expliquer cela, nous présentons figure 23 les temps de latence pour une exécution synchrone. le temps de latence est très peu impacté par la parcimonie du réseau. En effet, le réseau SAD a très peu de calculs, le parallélisme de l'exécution GPU est déjà permis de réduite le temps d'inférence au maximum. En revanche,



(a) asynchrone (b) synchrone

FIGURE 20 – Latence (en ms) de transfert CPU/GPU et d'inférence GPU pour le réseau VGG19 pour différentes tailles de batch d'entrée et pour des exécutions synchrone et asynchrone.

Parcimonie	**0**	**0.2**	**0.4**	**0.6**	**0.8**	**0.9**
L1 Element	90.5	90.8	90.47	90.87	90.66	90.15
L1 Filter	90.5	90.59	89.96	88.48	84.21	50.0
L1 Channel	90.5	90.65	90.39	89.9	89.01	85.88

(a) SAD

Parcimonie	**0**	**0.25**	**0.75**	**0.85**	**0.95**	**0.99**
L1 Element	86.16	84.19	84.19	85.37	86.03	85.86
Parcimonie	**0**	**0.2**	**0.4**	**0.6**	**0.8**	**0.9**
L1 Channel	86.16	83.16	79.92	80.08	79.08	70.59

(b) LID

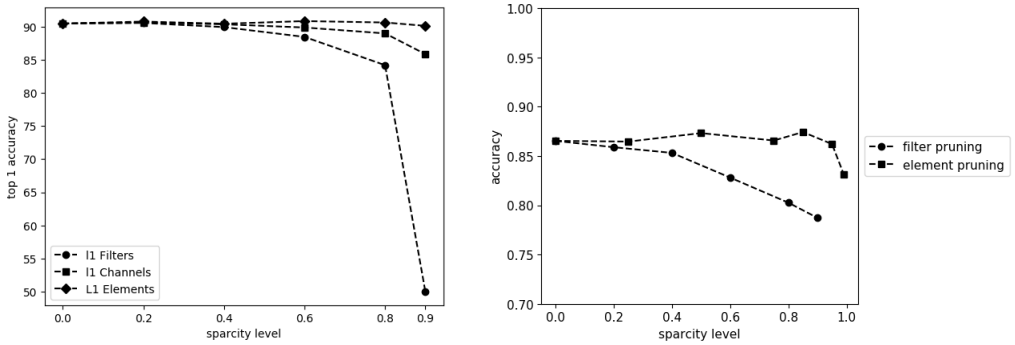
TABLE 8 – Accuracy après le Pruning pour différentes valeurs de parcimonie et différentes méthodes des réseaux SAD et LID. La représentation graphique est donnée figure 21.

l'inférence CPU est bien accélérée : le CPU n'a pas les capacités de parallélisation du GPU.

— Pour le réseau LID, pour des tailles de batchs suffisamment grandes (à partir de 128), le temps d'inférence est cette fois bien impacté par la compression du réseau. Sur GPU, l'inférence peut être accélérée d'un facteur 5 avec 95% de parcimonie. Sur CPU, l'accélération atteint même un facteur 24 avec avec 95% de parcimonie.

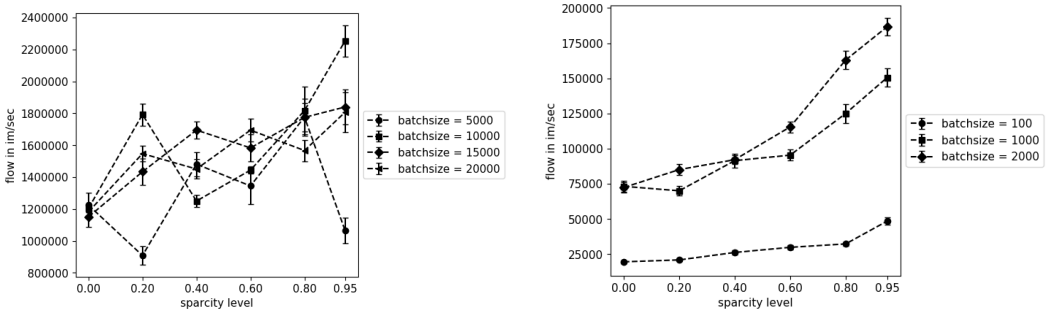
3.4 Conclusion du pruning

Dans cette partie, nous avons étudié les performances de compression et d'accélération de réseaux de neurones convolutifs par pruning. Sur les différents réseaux étudiés, la parcimonie scalaire et non structuré permet de se passer de 80% à 90% des poids des couches convolutives. Ce type de compression a seulement été simulé, il faudrait dans la suite utiliser la méthode de stockage développée par [Han et al., 2015] et expliquée en analyse bibliographique [Hurault, 2019], c'est-à-dire stocker les poids scalaires restants avec leur



(a) SAD (b) LID

FIGURE 21 – Accuracy après le Pruning pour différentes valeurs de parcimonie et différentes méthodes des réseaux SAD et LID. Les performances correspondantes sont données tableau 8.



(a) GPU (b) CPU

FIGURE 22 – Débit (en images/sec) sur GPU et CPU après le pruning de SAD pour différentes valeurs de parcimonie et différentes tailles de batch d'entrée.

index respectifs. Cependant, nous avons observé que les performances d'accélération sur GPU suivent seulement les accélérations dues aux transferts mémoire CPU-GPU. L'accélération GPU dépend donc seulement du taux de parcimonie qu'il soit obtenu avec du pruning scalaire, par filtre, ou par canal. Les performances d'accélération GPU présentées sont donc à mettre en correspondance avec les performances métier de pruning scalaire. Sur la plupart des réseaux, la valeur d'accélération GPU avec 90% de parcimonie est donc atteignable. Il sera donc possible, par exemple pour le réseau LID, d'accélérer l'inférence GPU d'un facteur 5. Cependant, l'inférence sur CPU reste accélérée que si le pruning est structuré à l'échelle du canal.

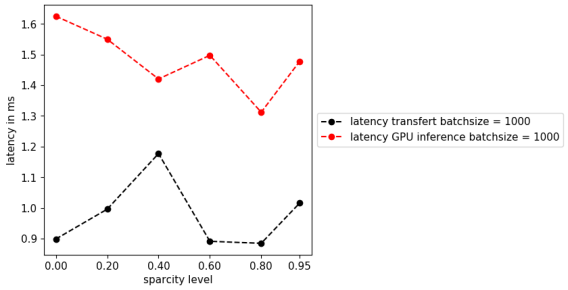
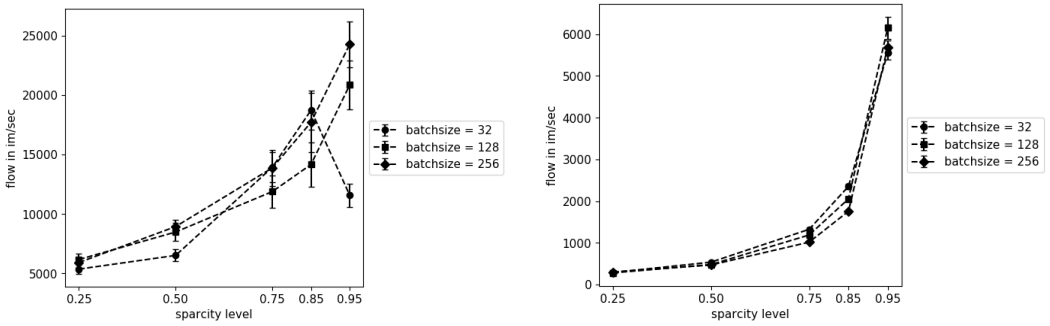


FIGURE 23 – Latence (en ms) de transfert CPU/GPU et d'inférence GPU pour le réseau SAD lors de l'inférence en mode synchrone.



(a) GPU (b) CPU

FIGURE 24 – Débit (en images/sec) sur GPU et CPU après le pruning de LID pour différentes valeurs de parcimonie et différentes tailles de batch d'entrée.

4 Quantification

Nous testons maintenant les performances de quantification de nos réseaux de neurones. Encore une fois, nous séparons l'analyse des performances d'accélération et de compression.

4.1 Performances métier

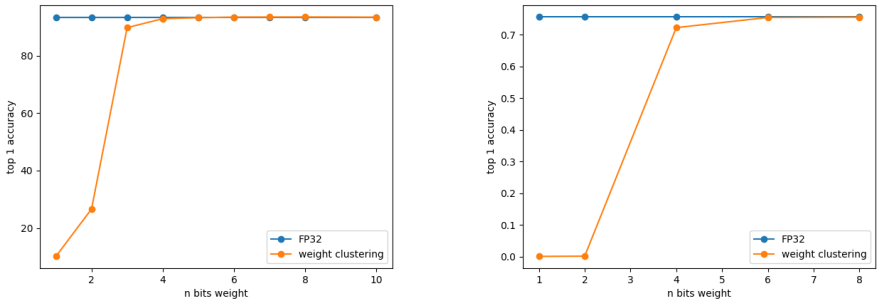
Dans cette section, nous étudions les performances de précision d'un réseau de neurones quantifié. Le but est de savoir jusqu'à quel nombre de bits de quantification des poids et/ou des activations, les performances du réseau ne seront pas dégradées. Nous allons comparer, sur nos réseaux, différentes méthodes de quantification évoquées lors de l'analyse bibliographique [Hurault, 2019]. Ces méthodes se classent en deux groupes : quantification post-apprentissage et quantification avec apprentissage.

4.1.1 Quantification post-apprentissage

Clustering seulement des poids

Nous implémentons d'abord la méthode proposée par [Han et al., 2016] et détaillée dans [Hurault, 2019] : à la couche l, tous les poids scalaires $w \in W^l$ sont partitionnés en k^l clusters $C^l = \{C_1^l, ..., C_k^l\}$ par l'algorithme k-means. Le nombre de clusters est constant dans le réseau et est fixé par le nombre de bits de quantification $k = 2^n$ où n est le nombre de bits choisi. Alors que [Han et al., 2016] ré-entraîne les poids du réseau après quantification, pour l'instant nous quantifions seulement nos poids pré-entraînés sans ré-apprentissage. Cette méthode ne permet pas d'accélérer un réseau grâce au calcul en précision limitée car les cluster restent en précision flottante. On rappelle que le facteur de compression, pour un réseau initialement codé sur b bits (typiquement $b = 32$), est approximativement $\frac{b}{n}$.

Les résultats obtenus sont présentés figure 25. On observe qu'il est possible, sans ré-apprentissage et sans perte de performance, de compresser Resnet56 pour CIFAR10 d'un facteur $32/4 = 8$ ou Resnet50 pour ImageNet d'un facteur $32/5 = 6.4$. On rappelle que le taux de compression obtenu, sans perte de performance, par pruning pour ResNet56 était d'environ de 4. La quantification permet ici une compression deux fois plus efficace.



(a) CIFAR10 (b) ImageNet

FIGURE 25 – Top 1 accuracy après quantification des poids par clustering, sur les bases CIFAR10 et ImageNet

Quantification des poids et des activations

Afin de profiter du calcul en précision limitée détaillé en 4.2.2, nous étudions les performances métiers de réseaux dont les poids et les activations sont quantifiés vers l'entier. Nous utilisons les implémentations [Zmora et al., 2018] et [Zhao et al., 2019] comme bases de code. L'implémentation suit le schéma de "quantification simulée" proposé par [Jacob et al., 2017] et résumé figure 26.

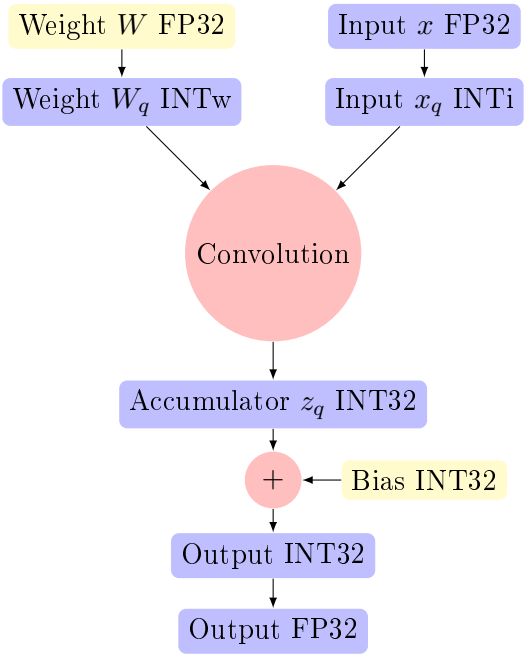


FIGURE 26 – Quantification des poids et activations pour l'inférence.

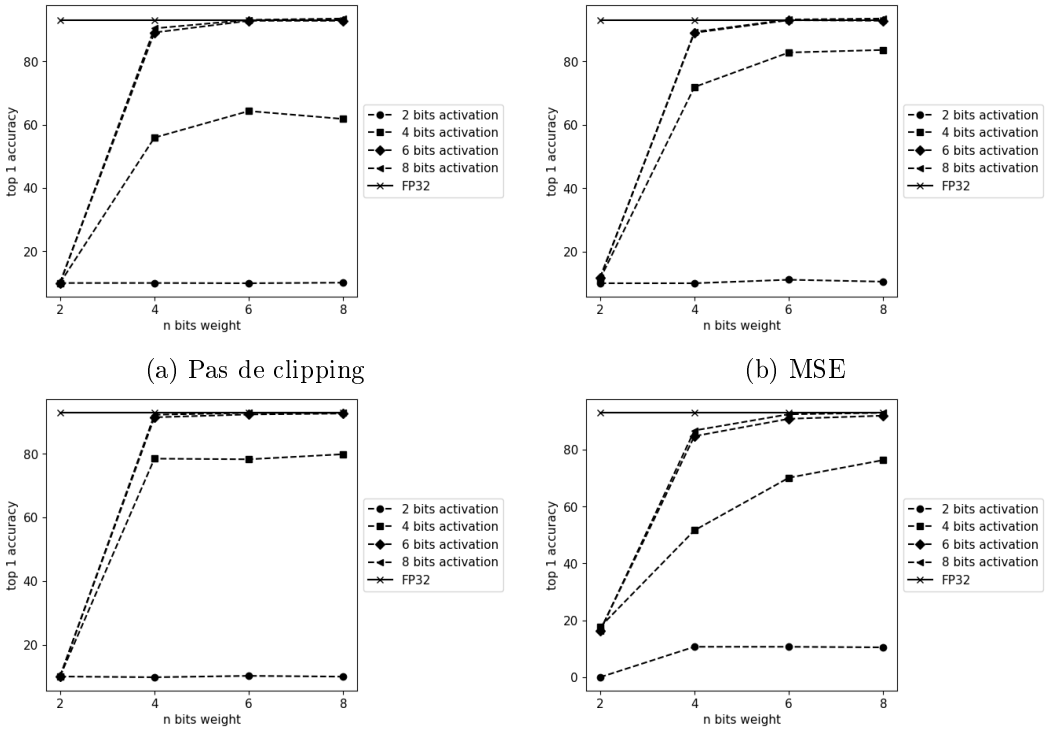
Nous utilisons dans un premier temps la méthode classique d'arrondi uniforme déterministe présenté en analyse bibliographique [Hurault, 2019] et nous comparons différentes méthodes de clipping. Les méthodes comparées sont :

- Pas de clipping (max)
- Moindres carrés (MSE)
- Divergence de Kullback-Leibler (KL)
- ACIQ (Analytical Clipping for Integer Quantization)

Les figures 27 et 28 présentent les performances, pour ResNet56 entraîné sur CIFAR10 et ResNet50 entraîné sur ImageNet, des différentes méthodes de clipping. Nous faisons varier le nombre de bits de quantification des poids et des activations entre 2 et 8 bits.

Les meilleures performances sont obtenues avec les méthodes de clipping par moindres carrés ou par divergence KL. On rappelle que ces méthodes ont le désavantage de nécessiter une calibration antérieure à l'inférence, au contraire d'ACIQ. Cette calibration s'effectue cependant hors ligne, et peut être effectuée une seule fois. Pour ResNet56 sur CIFAR10, le clipping par divergence KL permet une quantification du réseau sans pertes de performances avec des poids 4 bits et des activations 6 bits. Pour ImageNet, il faut utiliser au minimum 6 bits de poids et 6 bits d'activations. On observe aussi que la quantification des activations est plus sensible que celle des poids.

Nous avons également testé la quantification stochastique, aussi présentée en [Hurault, 2019], mais celle-ci n'améliorait pas les performances par rapport à la quantification déterministe



(a) Pas de clipping (b) MSE

(c) KL (d) ACIQ

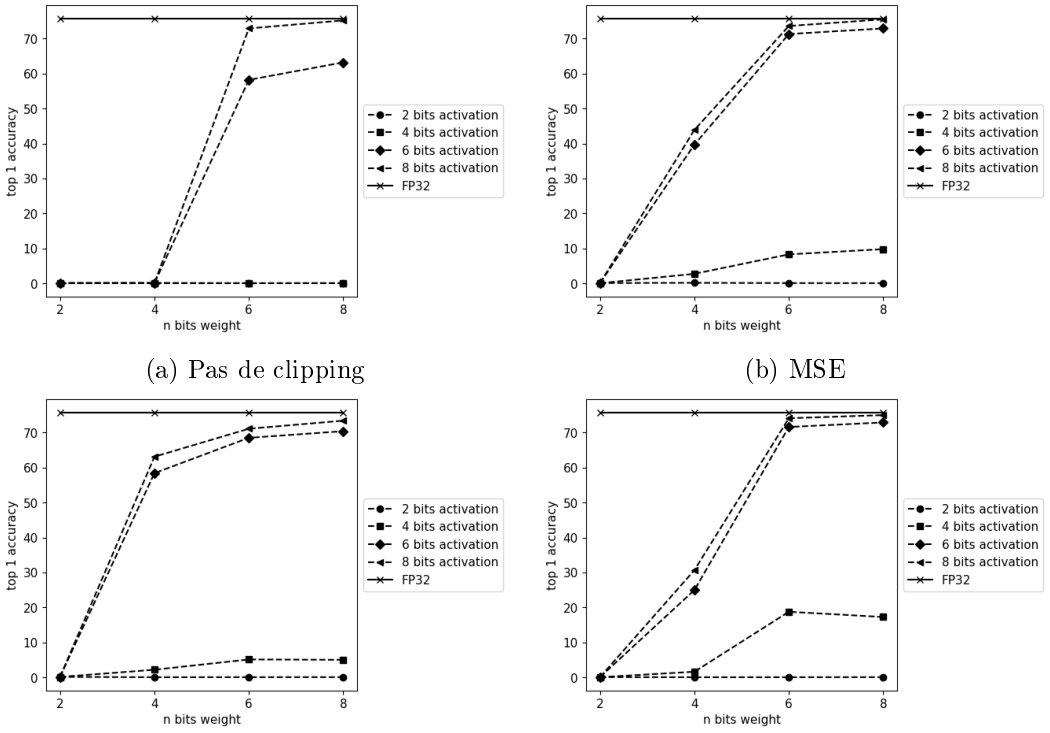
FIGURE 27 – Quantification uniforme post-apprentissage des poids et activations pour différentes méthodes de clipping. Testées sur Resnet56 entrainé sur CIFAR10.

précédente.

4.1.2 Quantification avec apprentissage

Afin d'améliorer les performances précédentes, nous testons maintenant la quantification avec apprentissage. Nous partons d'un réseau pré-entraîné, qui est ré-appris avec inférence quantifiée. Cette méthode d'entraînement, décrite en section "Quantization-aware training" de [Hurault, 2019], est résumée figure 29. La mise à jour des poids se fait en précision flottante et l'inférence est quantifiée.

- Les **poids** sont quantifiés avant chaque passe d'inférence. Par soucis de simplicité, nous utilisons la quantification précédente sans clipping : la mise à l'échelle sur x s'effectue linéairement $[min(x), max(x)] \longrightarrow [-1, 1]$. Le code est adapté pour quantification sur 1 bit : $W_k = \pm\alpha_k$ avec $\alpha_k = \frac{1}{|W_k|}||W_k||_{l1}$ la moyenne des valeurs absolues du vecteur de poids.
- Pour quantifier les **activations**, l'implémentation suit la méthode des "Exponential Moving Average" (EMA) : les moyennes α_k des cartes d'activations mises à jour avec $\alpha_k \longleftarrow \lambda\alpha_k^{new} + (1-\lambda)\alpha_k$ où α_k^{new} correspond à la moyenne calculée après une



(a) Pas de clipping (b) MSE

(c) KL (d) ACIQ

FIGURE 28 – Quantification uniforme post-apprentissage des poids et activations pour différentes méthodes de clipping. Testées sur Resnet50 entrainé sur ImageNet.

passe directe. Pour quantifier les activations sur 1 bit, comme [Zhou et al., 2016] ou [Mishra et al., 2018], les activations sont simplement "clippées" sur $[0, 1]$ sans mise à l'échelle.

Nous apprenons sur 20 époques et le pas d'apprentissage ("learning rate") utilisé est 1/10 de celui de fin d'apprentissage du réseau initial. N'ayant pas accès à la base de donnée d'entraînement d'ImageNet, l'expérience d'apprentissage n'est pas réalisée avec le réseau ResNet50.

Le tableau 9 et la figure 30 présentent les résultats obtenus. On observe qu'avec des poids quantifiés sur 1 seul bit, il est possible d'atteindre 75.5% d'accuracy, soit une augmentation de 17.9% du taux d'erreur. Les performances avec 2 bits d'activations sont moins bonnes que les performances 1 bits car elles sont été obtenues avec quantification des activations grâce à la méthode "EMA" qui n'est pas adaptée pour de la quantification à ce niveau. Ensuite, à partir de poids et activations quantifiés sur 3 et 6 bits, les performances du réseau FP32 sont atteintes. Il fallait au moins 6 bits de poids pour atteindre ces performances sans apprentissage. Notons qu'avec une quantification des poids et activations à 4 bits, les performances ne sont pas dégradées. Cela est intéressant pour les cartes GPU à architecture Turing qui contiennent des cœurs de calcul INT4.

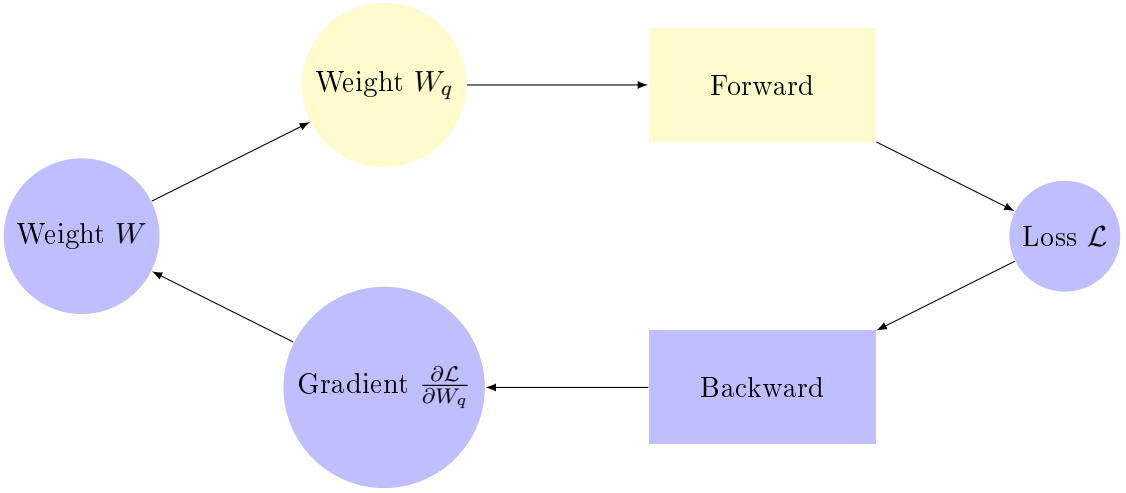


FIGURE 29 – Quantization-aware training. Labels are yellow when we work in limited precision and blue when we work with in full precision.

Nb bits activations / Nb bits poids	**1**	**2**	**3**	**4**	**5**	**6**
1	63.22	64.29	67.36	68.18	68.01	67.8
2	72.91	77.4	78.61	80.35	79.3	79.54
4	75.07	88.87	91.65	92.02	92.16	92.13
6	73.28	90.43	92.75	93.23	93.48	93.35
8	75.47	90.66	92.89	93.23	93.48	93.4

TABLE 9 – Accuracy après ré-apprentissage quantifié sur 20 époques de Resnet56 entraîné sur CIFAR10.

4.2 Performances d'accélération

4.2.1 Inférence en précision limitée

Le but de cette section est de mesurer l'accélération de l'inférence d'un réseau de neurone grâce à la réduction de précision. L'accélération est permise par le calcul en précision entière. Elle suppose la quantification des poids et des activations lors de l'inférence. TensorRT est le seul framework python à utiliser des kernels CUDA spécifiques au calcul en précision limitée INT8. Le framework Pytorch ne permet pas le calcul en précision entière mais seulement en précisions FP64, FP32 et FP16.

4.2.2 Carte GPU RTX5000

Afin de mesurer les performances temporelles de nos algorithmes, nous utilisons la carte graphique NVIDIA QUADRO RTX5000. Nous avons choisi cette carte GPU car elle pos-

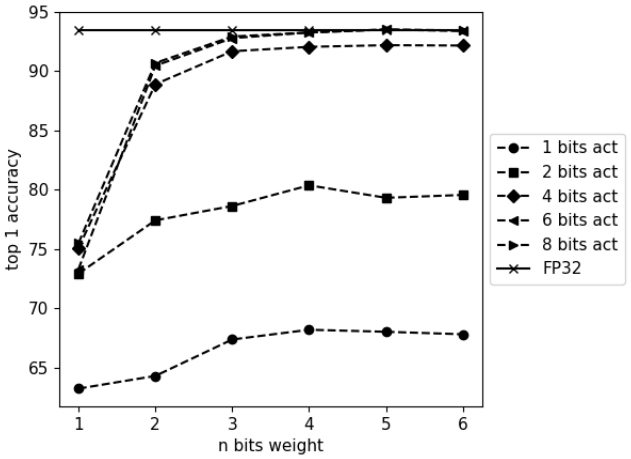


FIGURE 30 – Accuracy après ré-apprentissage quantifié sur 20 époques de Resnet56 entraîné sur CIFAR10.

Mémoire GPU	16GB
Consommation max	265 W
Coeurs de traitement parallèle CUDA	3072
Coeurs NVIDIA Tensor	384
Coeurs NVIDIA RT	48
Pic performances FP32	11.2 TFLOPs
Pic performances FP16	22.3 TFLOPs
Pic performances INT8	178.4 TOPs

TABLE 10 – Performances constructeur carte NVIDIA RTX5000

sède un cœur de traitement identique aux cartes NVIDIA T4. Ces dernières sont des cartes à refroidissement passif, uniquement exploitables sur des serveurs, et dissipant seulement 70W. Basée sur l'architecture Turing, cette carte possède des cœurs de calculs matriciels en INT8, INT4 et INT1. Les performances annoncées par le constructeur sont reportées tableau 10. Les cœurs NVIDIA "Tensor Cores" sont des unités de calculs MMA (Matrix-Multiply-and-Accelerate). Chaque cœur réalise des traitements $D = A * B + C$ où A,B,C et D sont des matrices carrées. Ces cœurs de calcul tensoriel permettent du calcul flottant FP32, FP16 et entier INT8, INT4,INT1.

Pour le calcul flottant FP16, comme illustré Figure 31, le coeur réalise des calculs 4×4 soit 64 opérations flottantes par cycle d'horloge. RTX5000 contient 384 Tensor Cores, 8 par "streaming multiprocessor" (SM). Donc chaque SM réalisera 1024 opérations flottantes par cycle d'horloge. Pour le calcul FP32, la multiplication se fera en demi-précision FP16 et les accumulateurs en précision FP32. NVIDIA nomme ce traitement précision mixte. Comme exposé figure 32, les modes INT8,INT4,INT1 permettent encore de multiplier le

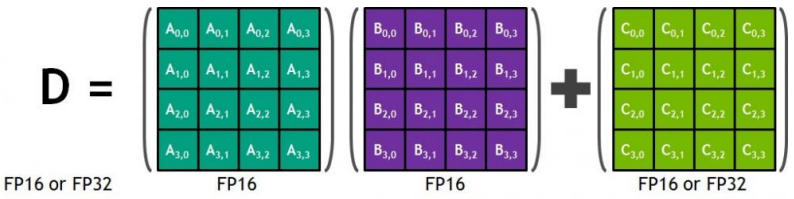


FIGURE 31 – Calcul en précision mixte des coeurs NVIDIA "Tensor Cores"

nombre d'opérations par cycle par 2, 4 et 16.

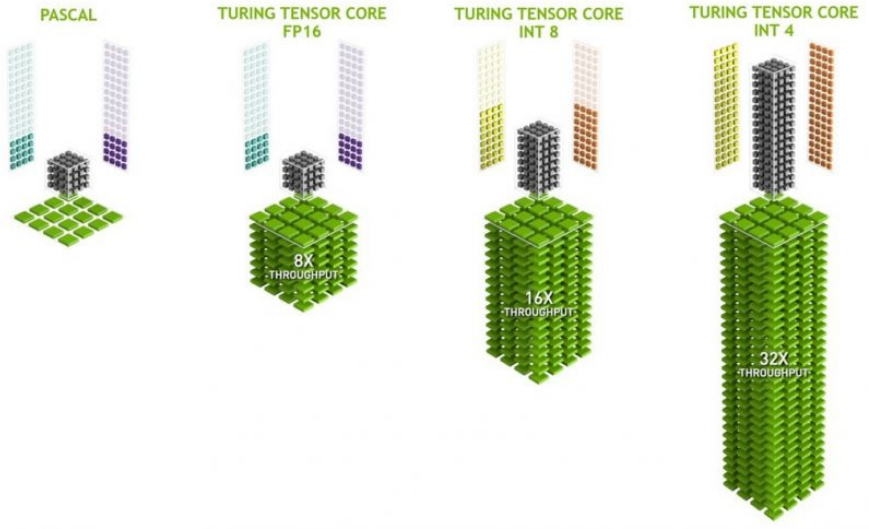


FIGURE 32 – Coeurs NVIDIA Tensor

Afin d'avoir une idée des performances d'accélération possible grâce au calcul en précision limitée, nous utilisons CUTLASS, une librairie C++ qui s'appuie notamment sur l'API WMMA (Warp Matrix Multiply and Accumulate) de CUDA pour réaliser des opérations GEMM (GEneral Matrix to Matrix Multiplication) en précision entière sur les cœurs NIVIDIA Tensor Cores. Une opération GEMM est une opération matricielle générale $C = \alpha A * B + \beta C$. Les méthodes et le code CUTLASS d'optimisation de ces calculs sont décrits précisément en [NVIDIA, 2019]. L'idée principale est de décomposer les calculs en une hiérarchie de blocs illustrés figure 33.

Nous réalisons les opérations $A*B$ sur des matrices carrées A et B de plus en plus grandes. La figure 34 représente le temps de calcul pour chaque format rapporté au temps du calcul FP32 classique. Les moyennes d'accélération sont reportées tableau 11. On observe que l'utilisation des cœurs Tensor Cores en précision mixte accélère le calcul d'un facteur 4. Les calculs FP16 et FP32 en WMMA sont exactement les mêmes car il n'y a pas d'addition d'accumulateurs pour notre expérience. Les performances sont bien conformes aux spécifications attendues. On retrouve aussi les performances d'accélération attendues

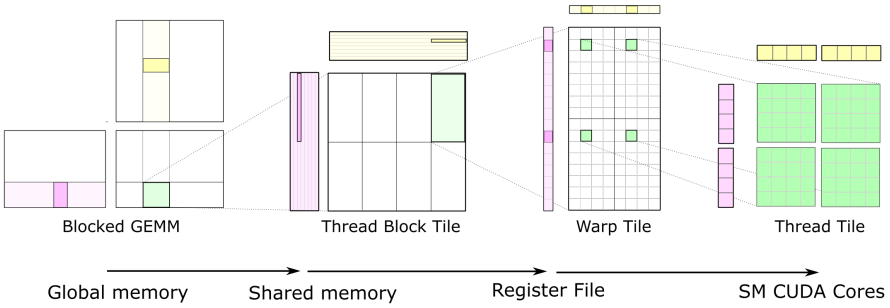


FIGURE 33 – La hiérarchie des transferts de donnée depuis la mémoire lente vers la mémoire rapide lors du calcul GEMM

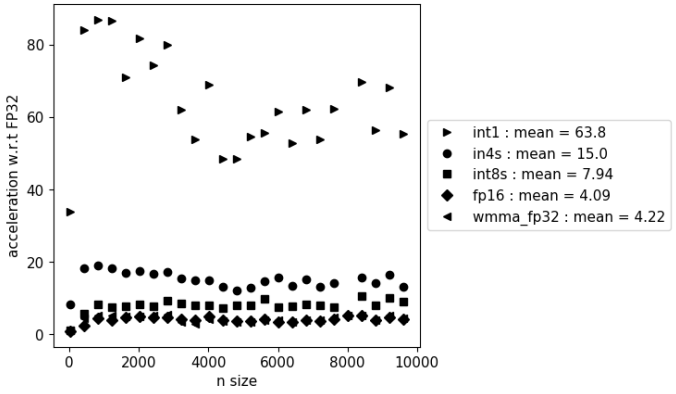


FIGURE 34 – Implémentation CUTLASS de multiplications matricielles sur RTX5000. Pour différentes tailles de matrices $n \times n$, mesure de l'accélération du calcul en précision limitée avec les coeurs Tensor Cores par rapport à l'utilisation des cœurs CUDA en FP32.

en mode entier INT8, INT4 et INT1.

4.2.3 Inférence en PyTorch

Nous mesurons maintenant les performances d'accélération de l'inférence après modification de la précision des poids et des activations d'un réseau de neurones. Comme pour le pruning, nous utilisons d'abord les réseaux ResNet56 entraîné sur CIFAR10 et ResNet50 entraîné sur ImageNet. Pytorch ne permet que l'inférence pour des formats FP32 ou FP16 sur GPU. Pour réaliser ses calculs en FP16 sur GPU, PyTorch peut accéder aux calcul GEMM de précision mixte de Tensor Cores à travers cuDNN. Pour les deux réseaux, l'inférence en FP16 ne modifie pas les performances métier.

Nous observons tableau 12 et figure 35 l'évolution des débits FP16 et FP32 selon la taille de batch en entrée. Les expériences sont répétées sur 100 batchs. Nous affichons la moyenne

Format	Accélération moyenne
FP32 WMMA	4.22
FP16	4.09
INT8	7.94
INT4	15.0
INT1	63.8

TABLE 11 – Implémentations CUTLASS de multiplications matricielles sur RTX5000. Moyenne des accélérations du calcul en précision limitée avec les coeurs Tensor Core par rapport au calcul avec cœurs CUDA en FP32.

et l'intervalle de confiance à 95%.

Pour expliquer ces évolutions, nous observons figure 36, avec ResNet56 entraîné sur CIFAR10, pour des formats FP16 et FP32 :

— Le temps cumulé des transferts CPU-GPU.
— Le temps d'inférence pure sur GPU.

Format / Taille de batch	**1**	**32**	**64**	**128**	**256**	**512**	**1024**
CPU FP32	25	482	611	**638**	610	460	388
FP32	76	2070	3948	7258	7869	**8031**	6597
FP16	49	2630	3141	7669	14144	**15320**	12211

(a) ResNet56 entraîné sur CIFAR10

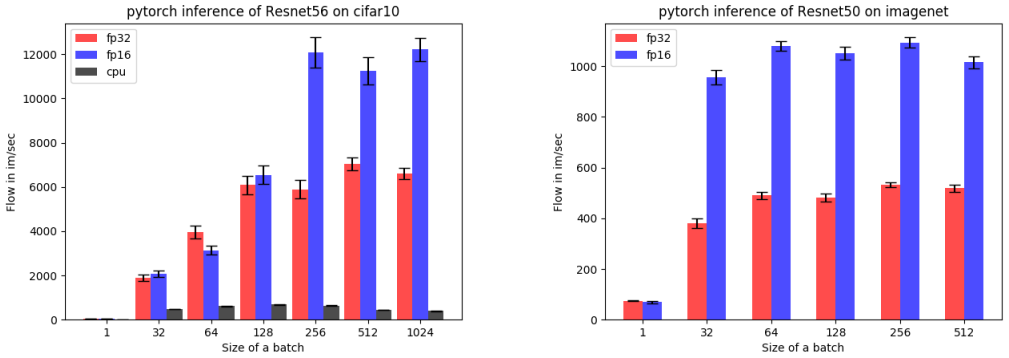
Format / Taille de batch	**1**	**32**	**64**	**128**	**256**	**512**
FP32	75	957	1080	1052	**1094**	1015
FP16	69	380	490	481	**533**	519

(b) ResNet50 entraîné sur ImageNet

TABLE 12 – Débit moyen (nombre d'images traitées / secondes) lors de l'inférence sous PyTorch sur GPU en formats FP16 et FP32, et sur CPU en FP32.

Ces débits ont été obtenus avec une exécution asynchrone : l'accélération de l'inférence est contenu dans le temps de transfert. Le temps de transfert FP32 reste toujours deux fois supérieur au temps de transfert FP16. Ces temps augmentent linéairement avec la taille de l'entrée.

Ainsi pour des petits batchs, les temps de calcul et de transfert sont équivalents, on n'observe pas d'accélération FP16 par rapport à FP32. Pour des plus gros batchs, le temps de transfert devient prépondérant devant le temps de calcul, et le traitement FP16 devient environ deux fois plus rapide que le traitement FP32. De plus, l'augmentation linéaire du temps de transfert en fonction de la taille de batch induit bien un débit $\frac{batch\_size}{temps\ d'infrence}$ constant à partir d'une certaine taille de batch figure 35. Cette taille de batch limite est respectivement, pour les inférences FP32 et FP16, environ 128 et 256.



(a) CIFAR10 (b) ImageNet

FIGURE 35 – Débit (nombre d'images traitées / secondes) de l'inférence sous PyTorch sur GPU en formats FP16 et FP32, et sur CPU en FP32.

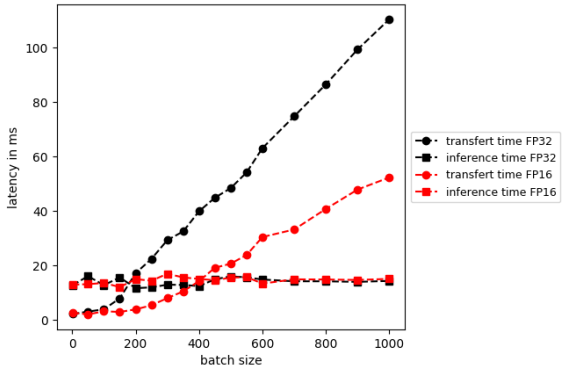


FIGURE 36 – Temps de transfert et de traitement GPU lors de l'inférence de ResNet56 sur CIFAR10, pour des formats FP16 et FP32.

Pour ImageNet, le débit est constant pour des batchs de taille supérieure à 32.

Nous profitons de ces résultats pour comparer l'inférence FP32 sur GPU et CPU, avec ResNet56 sur CIFAR10. LE CPU utilisé est Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz. Le débit optimal FP32 sur GPU est 24× plus rapide que celui sur CPU. Pytorch ne permet cependant pas de réaliser l'inférence en format FP16 sur CPU.

4.2.4 TensorRT

TensorRT est un logiciel d'exécution qui permet l'optimisation de l'inférence des réseaux de neurones sur les GPUs NVIDIA. Parmi les différentes opérations d'accélérations possibles, TensorRT permet notamment de réaliser l'inférence de réseaux de neurones aux formats INT8 et FP16. TensorRT prend en entrée un réseau pré-entraîné et ne traite pas de l'ap-

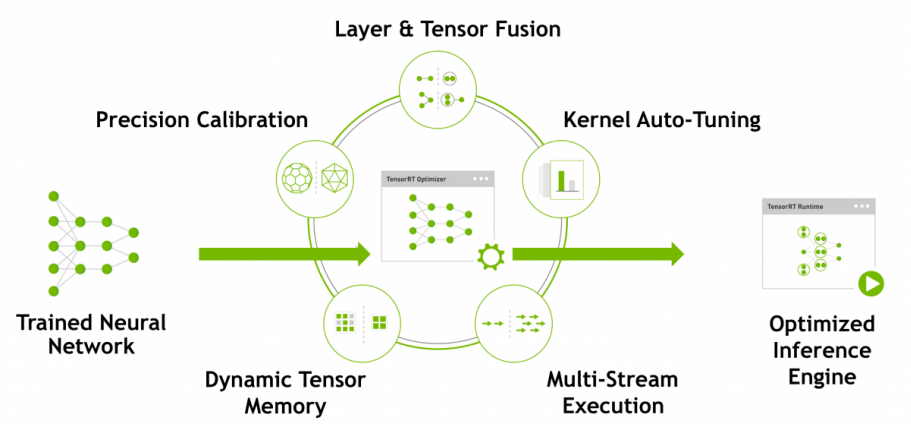


FIGURE 37 – Inférence avec TensorRT

prentissage. TensorRT accède automatiquement aux cœurs Tensor Cores. Nous utilisons l'API Python qui permet de réaliser l'inférence en suivant les étapes suivantes :

1. **Définition d'un réseau TensorRT.** La voie la plus simple pour cela est d'importer un modèle en utilisant les parsers de TensorRT avec les formats Caffe, ONNX ou UFF. Nous utilisons cette méthode avec des modèles entraînés sous Keras ou Tensorflow et convertis en format UFF. Une seconde méthode pour définir un modèle est d'utiliser directement l'API Python de TensorRT pour définir manuellement toutes les couches du réseau. Nous utilisons cette méthode lorsque nos modèles (notamment pour ceux de traitement de la parole) sont uniquement disponibles en format PyTorch.
2. **Utilisation du builder TensorRT pour créer un cadre d'exécution optimisée pour le réseau.** Le builder prend notamment en entrée la précision désirée FP32, FP16 ou INT8. Pour quantifier poids et activations en format INT8, TensorRT doit connaître l'intervalle de valeur de chaque tenseur d'activation. Ceci permet, lors de l'inférence, de déterminer la valeur de seuillage lors de l'opération de quantification. Il est possible de déterminer manuellement l'intervalle de chaque tenseur ou d'utiliser la calibration de TensorRT qui génère ces valeurs grâce à une passe d'inférence sur une base de données de calibration. Dans cette section, nous nous intéressons seulement aux performances d'accélération et non aux performances métier. Nous choisissons donc d'utiliser une calibration sur seulement quelques batchs puis de conserver ces valeurs pour le reste des expériences, même si celle-ci ne sont pas optimales.
3. **Inférence**

Nous répétons les expériences d'inférence des réseaux ResNet50 (CIFAR10) et ResNet56 (ImageNet) avec le framework TensorRT. Nous nous attendons d'abord à ce que les infé-

rences FP32 et FP16 soient accélérées, grâce aux diverses optimisations de TensorRT, d'un facteur constant par rapport à l'inférence sous PyTorch. Au vue des performances obtenues avec cutlass précédemment, l'inférence sous format INT8 doit également permettre une accélération d'un facteur d'environ 2 par rapport au format FP16. Les résultats sont reportés tableau 13 et figure 38.

Format / Taille de batch	**1**	**32**	**64**	**128**	**256**	**512**	**1024**
FP32	858	10573	10738	11591	**11753**	11295	11165
FP16	704	20116	22452	23510	**24168**	24012	23585
INT8	1614	30366	39320	13006	46626	**46902**	46252

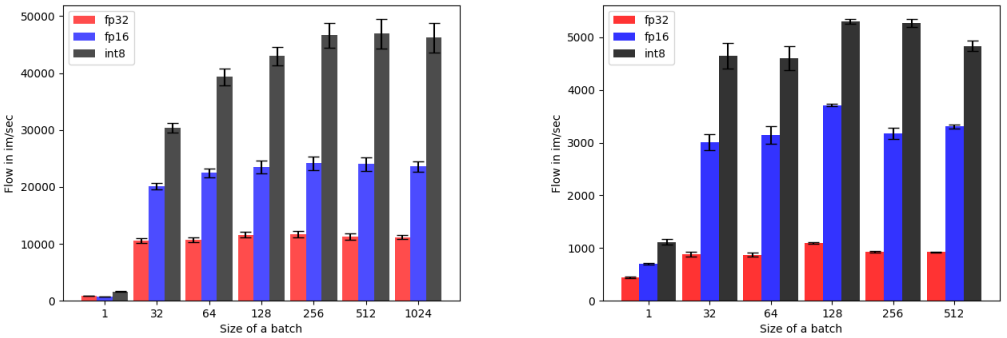
(a) ResNet56 entraîné sur CIFAR10

Format / Taille de batch	**1**	**32**	**64**	**128**	**256**	**512**
FP32	447	881	873	1092	930	926
FP16	699	3008	3145	3709	3171	3307
INT8	1116	4648	4602	5298	5269	4834

(b) ResNet50 entraîné sur ImageNet

TABLE 13 – Débit moyen (nombre d'images traitées / secondes) lors de l'inférence sous TensorRT en formats INT8, FP16 et FP32

Avec le réseau ResNet50 entraîné sur la base ImageNet, la quantification sur 8 bits permet d'atteindre un débit optimal de 5298 images/sec, obtenu une taille de batch de 128. Cela correspond aux performances annoncées par NVIDIA : avec la carte Tesla T4 ils annoncent un débit optimal de 5288 images/sec.



(a) CIFAR10 (b) ImageNet

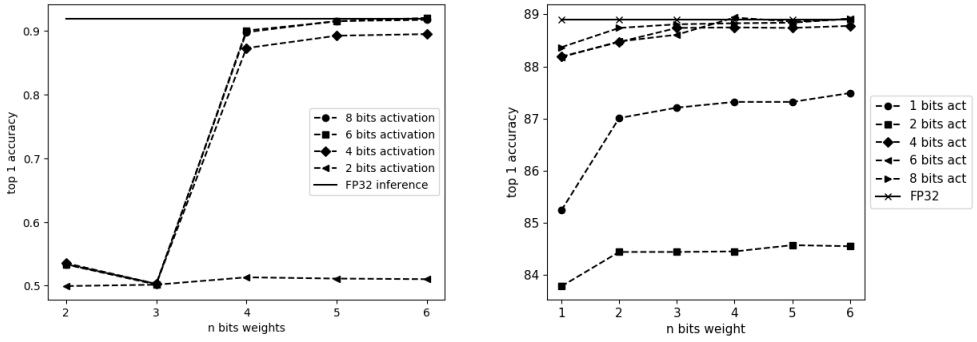
FIGURE 38 – Débit (nombre d'images traitées / secondes) de l'inférence sous TensorRT en formats INT8, FP16 et FP32, pour différentes tailles de batchs d'entrée, sur les bases CIFAR10 et ImageNet

4.3 Performances en traitement de la parole

Nous réalisons les mêmes expériences que précédemment sur les réseaux de traitement de la parole. L'apprentissage est effectué avec les mêmes paramètres que ceux de l'apprentissage initial.

4.3.1 Performances métier

Les performances métier de quantification sur les deux réseaux sont similaires. Elles sont représentées pour le réseau SAD figure 39 et pour le réseau LID figure 40. Sans apprentissage, on peut quantifier jusqu'à 6 bits les activations et 5 bits les poids. Avec ré-apprentissage, il est possible d'atteindre 4 bits d'activations et 3 bits de poids.



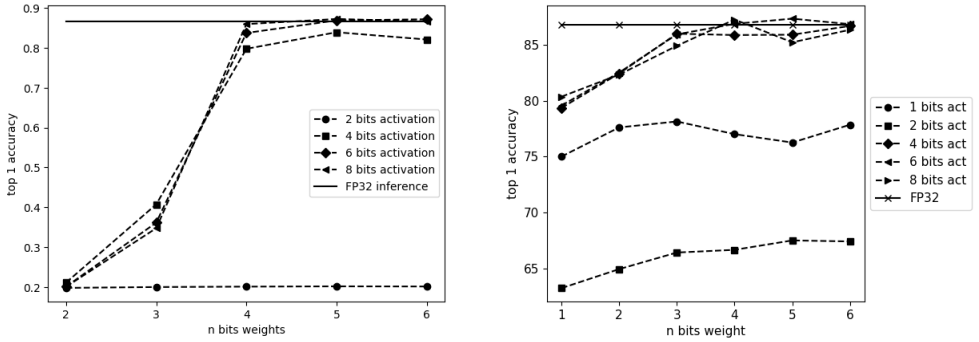
(a) Sans apprentissage (b) Avec apprentissage

FIGURE 39 – Accuracy après quantification uniforme, avec et sans apprentissage, des poids et activations de SAD sur différents bits de poids et activations.

4.3.2 Performances d'accélération

Les débits PyTorch et TensorRT des deux réseaux sont reportés figures 41 et 42.

— Pour la tâche SAD, les résultats obtenus sont difficiles à interpréter. Le réseau étant particulièrement petit, et déjà rapide, nous n'apportons pas de précision supplémentaires sur ces résultats.
— Pour la tâche LID, on retrouve des performances similaires à celles des réseaux ResNet. Les débits sont pratiquement constants car l'accélération est linéaire avec la taille des données d'entrée. En TensorRT, le format INT8 permet d'accélérer l'inférence FP32 d'un facteur environ 6.5. On rappelle qu'en comparaison, l'accélération due au pruning était de facteur 5.



(a) Sans apprentissage (b) Avec apprentissage

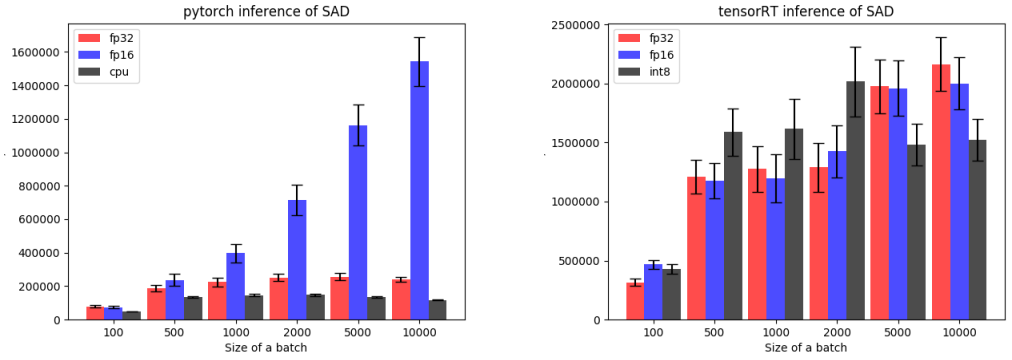
FIGURE 40 – Accuracy après quantification uniforme, avec et sans apprentissage, des poids et activations de LID sur différents bits de poids et activations.

5 Quantification et Pruning

5.1 Quantification vers des poids ternaires

La quantification ternaire quantifie un vecteur x vers les valeurs $(\pm\alpha, 0)$. Elle crée donc des poids binaires ou nuls. Par rapport à la quantification binaire de valeurs $(\pm\alpha)$, on rajoute une valeur possible 0. Intuitivement, si le pruning des poids scalaires est efficace sur un réseau, beaucoup de poids peuvent être mis à zéro sans perte de performances, il serait alors logique d'utiliser la ternairisation par rapport à la binarisation. Cette opération permet le pruning et la quantification simultanée et donc de compresser au maximum un réseau. Pour l'accélération, sur un matériel non spécialisé pour le calcul creux mais spécialisé pour le calcul INT1, le calcul matriciel avec des poids et activations binaire restera plus rapide qu'en ternaire.

On teste les performances de quantification linéaire vers des poids ternaires du réseau ResNet56 entraîné sur CIFAR10. Nous utilisons une quantification avec apprentissage sur 10 époques. Les poids sont quantifiés vers les valeurs $(\pm\alpha, 0)$. Ceci équivaut à une quantification linéaire symétrique sur 2 bits. Nous faisons varier le nombre de bits de quantification des activations sur 2, 4, 6, 8 et 32 bits. Les performances métiers sont reportées figure 43. Après quantification ternaire des poids du réseau, on peut observer que pour chaque couche traitée, on obtient en moyenne plus de 70% de parcimonie. Ainsi, il est possible, avec un très faible dégradation performances (4% d'accuracy), de compresser le réseau en ajoutant 70% de parcimonie et en quantifiant les poids restants sur 1 bit. Cela donné un taux compression finale d'environ 90.



(a) PyTorch (b) TensorRT

FIGURE 41 – Débit (nombre d'images traitées / secondes) de l'inférence de SAD sous PyTorch et TensorRT

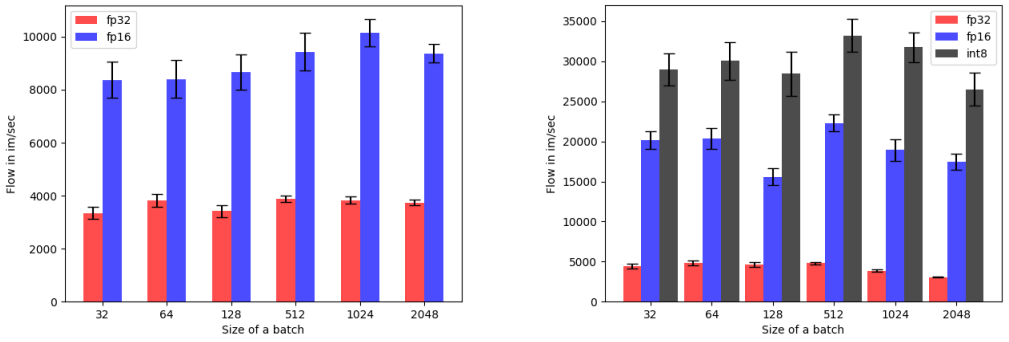
5.2 Quantification d'un réseau compressé

Au cours de l'analyse bibliographique [Hurault, 2019], nous faisons l'analyse que la plupart des réseaux de neurones possèdent une forte redondance de connections. Le pruning de réseaux a pour objectif de sélectionner seulement les connections les plus importantes, en supprimant la redondance et en gardant les poids qui auront un impact fort sur le processus de décision. Au contraire, la quantification exploite cette redondance pour réduire l'expressivité de chaque poids. Cette observation suggère que la quantification d'un réseau qui a déjà subit du pruning ne serait pas efficace.

Le réseau utilisé est ResNet56 entraîné sur CIFAR10. Nous utilisons les réseaux obtenus après pruning (et ré-apprentissage) $L1$, à l'échelle du canal, pour différents niveaux de parcimonie. Nous observons alors figure 44 les performances de quantification uniforme post-apprentissage. Le nombre de bits d'activation est fixé à 8 bits et le nombre de bits de poids varie. Pour un niveau de parcimonie donné, il est possible de quantifier les poids jusqu'à 5 bits sans pertes de performances. Pour un nombre de bits plus faible, plus le degré de parcimonie est élevé, plus les performances se dégradent rapidement. Ainsi, afin d'atteindre les meilleurs performances de compression et d'accélération, il existe un compromis à trouver entre pruning et quantification.

5.3 Conclusion sur la quantification

Dans cette partie nous avons montré que la quantification est une méthode très efficace pour compresser et accélérer l'inférence d'un réseau de neurone. Les performances de compression obtenues sont globalement aussi efficaces que celles du pruning. Nous rappelons que les meilleures performances de compression par Pruning sont obtenues en supprimant environ 90% des poids. A ce niveau de parcimonie, le taux de compression et de 10. Nous



(a) PyTorch (b) TensorRT

FIGURE 42 – Débit (nombre d'images traitées / secondes) de l'inférence de LID sous PyTorch et TensorRT

avons vu que la quantification des poids jusqu'à 3 bits permet souvent de ne pas perdre de performances métier. Cela correspond à un facteur de compression de $32/3 \approx 10$. La quantification permet de profiter de l'accélération des calculs grâce à la précision entière. Les performances d'accélération obtenues étaient limités aux calculs INT8, car aucun framework python permet d'utiliser les fonctions cudnn de calucl INT4 et INT1. Nous avons mis en avant le compromis à réaliser entre pruning et quantification afin de compresser et accélérer au maximum un réseau.

6 Distillation

La dernière partie du stage a été consacrée à la distillation. La méthode de distillation utilisée est celle de [Hinton et al., 2015] présentée en [Hurault, 2019]. Nous avons un réseau "professeur" pré-entraîné, et nous souhaitons apprendre un réseau "élève". La fonction de perte d'apprentissage de ce réseau est :

$$\mathcal{L}_\lambda = (1-\lambda)\mathcal{H}(\sigma(z_s), y_s) + \lambda T^2 \mathcal{H}(\sigma(\frac{z_s}{T}), \sigma(\frac{z_t}{T})) \tag{1}$$

où $\mathcal{H}$ est la fonction de perte de cross-entropy, z_s et z_t les logits des réseaux élèves et professeurs, et T un paramètre appelé température de distillation.

L'utilité de la distillation est expliquée communément par le fait qu'il est plus facile d'entraîner le réseau élève avec les probabilités issues d'un réseau pré-entraîné plutôt qu'avec des étiquettes binaires. La complexité du dataset est d'abord filtrée par le professeur. Par manque de temps pour aborder complètement cette partie, notre travail a seulement consisté en quelques expériences préliminaires sur la tâche SAD. Nous ne cherchons pas à optimiser le choix des paramètres de distillation et T.

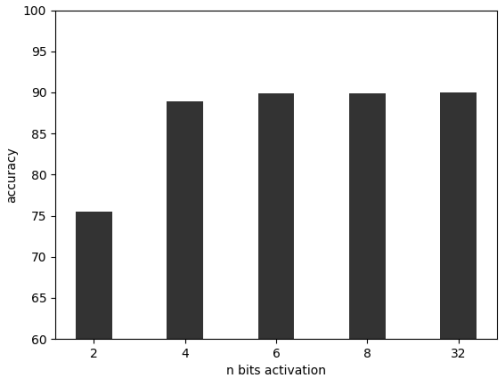


FIGURE 43 – Accuracy après quantification de ResNet56. Quantification des poids sur 2 bits.

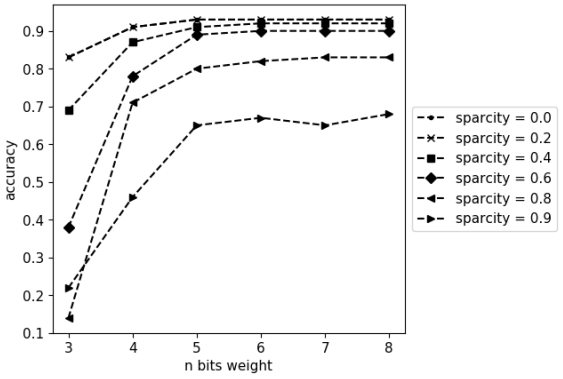


FIGURE 44 – Quantification sur différents bits de poids (activations : 8 bits) après pruning de ResNet56 entraîné sur CIFAR10.

Étant donné un réseau initial professeur pré-appris pour la tâche SAD , et un réseau élève plus petit, le buts de nos expériences a été de comparer :

— L'apprentissage du réseau élève par distillation avec $\lambda = 1$. Le réseau n'apprend qu'à l'aide des logits du professeur. Il n'a pas accès aux étiquettes du dataset.
— L'apprentissage du réseau élève avec $\lambda = 0$. C'est l'apprentissage "normal" avec étiquettes.

Les réseaux élèves testés sont successivement :

— Le réseau SAD compressé en retirant 50% des filtres de chaque couches.
— Seulement la dernière couche "fully-connected" du réseau SAD d'origine ("FC").

Les résultats de la comparaison des apprentissages avec $\lambda = 0$ et $\lambda = 1$ sont présentés tableau 14. Nous observons que l'apprentissage en utilisant les logits du réseau professeur ($\lambda = 1$) est légèrement meilleur que celui avec les labels bruts ($\lambda = 0$). Cependant, les performances restent très proches. La classification du réseau SAD étant binaire, il semble

que l'intérêt d'utiliser les logits du professeurs par rapport aux labels est limité.

Cependant, l'apprentissage du réseau élève a été réalisé de manière non-supervisée. Il serait intéressant pour la suite d'entraîner des plus gros réseaux type LID ou ResNet par distillation sur des données non labellisées, à l'aide d'un gros réseau pré-appris.

Réseau élève	$\lambda = 0$	$\lambda = 1$
SAD compressé à 50%	88.57	88.81
FC	76.14	77.06

TABLE 14 – Expériences de distillation avec le réseau SAD.

7 Conclusion

Dans ce travail nous avons exploré les performances de différentes méthodes de compression et d'accélération de l'inférence GPU de réseaux de neurones convolutifs. Nous avons d'abord mis en place des expériences avec deux réseaux ResNet entraînés sur les base CIFAR10 et ImageNet. Ces méthodes avaient d'abord été expliquées en analyse bibliographique [Hurault, 2019]. L'utilisation de ces deux réseaux classiques nous ont permis de comparer nos résultats avec ceux de la littérature. Ensuite, nous avons pu expérimenter sur des réseaux utilisés en pratique sur des tâches de traitement de la parole.

Nous avons pu constater que les résultats obtenus varient fortement selon la taille et l'architecture du réseau ou de la base de donnée utilisée. Nous avons montré que les performances de pruning étaient limitées lorsque le taux de compression souhaité est trop élevé, notamment lorsque le pruning s'effectue pour des filtres entiers. La quantification permet d'atteindre des taux de compression aussi élevés, tout en favorisant une exécution rapide sur GPU. Cependant, les performances d'accélération après quantification ne sont pas mesurables par TensorRT pour des formats autres que 8 bits.

Enfin, des premières expériences de Distillation nous ont permis de montrer qu'un réseau peut être appris efficacement seulement grâce à un réseau professeur et des données non étiquetés.

Sur le plan personnel, ce stage m'a permis de comprendre plus en profondeur le fonctionnement général des réseaux de neurones. Tout d'abord, l'analyse bibliographique [Hurault, 2019] m'a permis de lire de nombreux articles très récents et de découvrir les théories des méthodes de compression et d'accélération les plus efficaces. Ensuite, l'implémention de ces méthodes et l'analyse de leur performances m'a fait découvrir le fonctionnement des GPU et les problématiques associées dans le contexte de l'inférence d'un réseau de neurones.

Je tiens à remercier mon maître de stage pour son encadrement, sa disponibilité et sa patience pour répondre à mes nombreuses questions. Je remercie également toute l'équipe pour leur accueil chaleureux.

Références

[Canziani et al., 2016] Canziani, A., Paszke, A., and Culurciello, E. (2016). An analysis of deep neural network models for practical applications. *CoRR*, abs/1605.07678.

[Han et al., 2016] Han, S., Mao, H., and Dally, W. J. (2016). Deep compression : Compressing deep neural networks with pruning, trained quantization and huffman coding. *International Conference on Learning Representations (ICLR)*.

[Han et al., 2015] Han, S., Pool, J., Tran, J., and Dally, W. J. (2015). Learning both weights and connections for efficient neural networks. *CoRR*, abs/1506.02626.

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, abs/1512.03385.

[Hinton et al., 2015] Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*.

[Hurault, 2019] Hurault, S. (2019). Compression and acceleration of neural networks : bibliography analysis.

[Jacob et al., 2017] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A. G., Adam, H., and Kalenichenko, D. (2017). Quantization and training of neural networks for efficient integer-arithmetic-only inference. *CoRR*, abs/1712.05877.

[Michael H. Zhu, 2018] Michael H. Zhu, S. G. (2018). To prune, or not to prune : Exploring the efficacy of pruning for model compression.

[Mishra et al., 2018] Mishra, A., Nurvitadhi, E., Cook, J. J., and Marr, D. (2018). WRPN : Wide reduced-precision networks. In *International Conference on Learning Representations*.

[NVIDIA, 2019] NVIDIA (2019). cutlass. `https://github.com/NVIDIA/cutlass`.

[Qin et al., 2018] Qin, Z., Yu, F., Liu, C., and Chen, X. (2018). Demystifying neural network filter pruning. *CoRR*, abs/1811.02639.

[Russakovsky et al., 2015] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3) :211–252.

[Zhao et al., 2019] Zhao, R., Hu, Y., Dotzel, J., Sa, C. D., and Zhang, Z. (2019). Improving neural network quantization without retraining using outlier channel splitting. *CoRR*, abs/1901.09504.

[Zhou et al., 2016] Zhou, S., Ni, Z., Zhou, X., Wen, H., Wu, Y., and Zou, Y. (2016). Dorefa-net : Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, abs/1606.06160.

[Zmora et al., 2018] Zmora, N., Jacob, G., and Novik, G. (2018). Neural network distiller.