

Réduction Des Couches Convolutives Dans Les Réseaux Profonds

Vincent Martineau,^{1*}

¹Département d'informatique et de génie logiciel, Université Laval, Quebec, Canada

*E-mail: vincent.martineau.1@ulaval.ca.

Il y a eu des recherches expliquant comment réduire les couches de convolution dans les réseaux de neurones. Cependant dans les articles proposés les modèles utilisés sont généralement AlexNet et VGG. Lorsqu'essayé sur des réseaux plus complexes tels que ResNet et DenseNet les solutions proposées ne fonctionnent pas. Le but est d'explorer les limites de cette approche et de proposer une solution afin de pouvoir généraliser à des réseaux plus complexes et profonds que ceux dans l'article original de Molchanov. Les expérimentations seront faites sur le transfert de modèle entre ImageNet vers Cifar10 et nous verrons les gains pouvant être faits sur des réseaux profonds.

Introduction

Les réseaux CNN sont parmi nous depuis très longtemps (1). Ces réseaux sont très utilisés dans une grande variété de domaines. Que ce soit avec les systèmes de surveillance, les voitures autonomes ou les appareils intelligents, ce type de réseaux fait maintenant partie de notre quotidien (2).

Ces réseaux ont montré qu'ils pouvaient obtenir des résultats excédant les capacités humaines (3). Dans plusieurs cas, ces modèles sont rendus disponibles au grand public et il est possible de les adapter à des problèmes plus simples pour obtenir de très bons résultats (4).

0.1 Description du problèmes

Afin d'utiliser ces réseaux sûrs de petits appareils, certaines techniques (5) ont été évalué dans le passé. Plus récemment, un article propose de faire la réduction de filtres de convolution (6). Cependant l'article se concentre sur les réseaux AlexNet et VGG et ne fonctionne pas sur les réseaux plus complexes présent dans PyTorch (7).

Le but de cet article sera d'évaluer le travail nécessaire pour pouvoir pousser cette logique vers des réseaux plus complexes tels que ResNet, DenseNet et SqueezeNet.

Méthode

La méthode utilisée est une variante de la méthode proposée dans l'article Pruning Convolutional Neural networks for Ressource efficient inference (6). Le but des modifications est d'étendre les capacités de notre modèle à des réseaux beaucoup plus complexes tels que ResNet, DenseNet et SqueezeNet.

Voici une liste des étapes à suivre pour obtenir une réduction intéressante:

1. Appliquer le fine tuning sur un réseau pré-entraîné.
2. Extraire le graphe d'exécution de notre modèle
3. À partir de notre graphe d'exécution, exclure les couches de convolution que nous ne voulons pas affecter. Ces couches sont détaillées plus tard dans cet article.
4. Réduire les filtres en fonction du critère de réduction. Une description du critère est fournie dans une section futur de cet article.
5. Réinitialiser l'optimiseur utilisé pour l'entraînement.

6. Appliquer le fine tuning afin de restabiliser le réseau.
7. Recommencer la réduction tant que le réseau ne possède pas la taille voulue.

Dans cette procédure le critère de sélection est très important pour avoir de bons résultats. Pour cette expérimentation, nous avons utilisé les Taylor Expansions tel que mentionné dans l'article de Molchanov.

Afin de pouvoir exécuter sur des modèles plus complexes, nous avons choisi de ne pas explorer d'autres critères de sélection, mais bien de se concentrer sur l'algorithme nécessaire pour supporter ces modèles. Certains essais ont aussi été effectué avec l'activation moyenne des filtres et les résultats vont dans la même direction que l'article original.

Une des limitations importante dans notre solution est le besoin de réinitialisé l'optimiseur. Il s'agit d'une restriction majeur de notre proposition et des solutions devront être envisagées.

0.1 Les couches à ignorer

Dans la solution proposée, il est important de ne pas appliquer de réduction de filtres juste avant une addition tel que vu dans plusieurs modèles complexes. La figure Figure1 représente cette situation. Le problème dans ce cas est qu'il est nécessaire d'appliquer une réduction équivalente sur toutes les branches entrantes de l'addition. Dans le cas présent, il faudrait donc modifier ce qui vient avant la couche relu et l'entrée de la première convolution du bloc présenté.

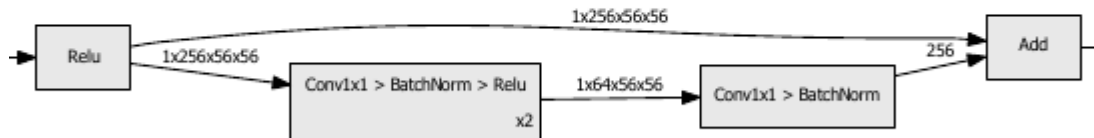


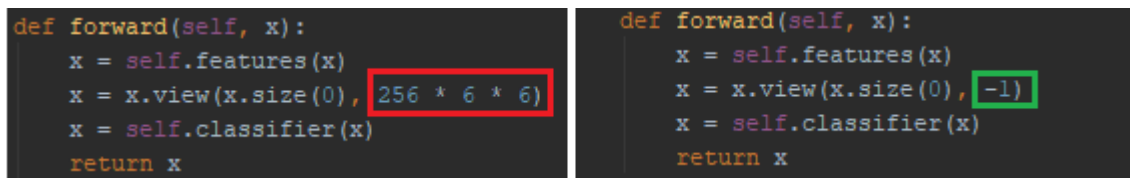
Figure 1: Dans la figure suivante il est impossible de réduire la seconde couche de convolution pour la branche du bas puisqu'il serait aussi nécessaire de modifier la couche précédent ce bloc ResNet pour que la connection résiduelle aie la bonne taille.

Un autre problème potentiel survient lorsqu'une couche est réutilisée. Dans ce cas il faut que l'entrée de notre couche soit compatible avec la sortie de la couche que nous traitons. du coup, nous devons affecter les couches précédentes pour que les dimensions concordent aussi. Ce qui revient au même problème que l'addition.

Il est à noter qu'il est possible qu'il existe un algorithme fiable qui permet de gérer ces cas, mais l'article présent n'explore pas ces options. Finalement, il est crucial de ne pas affecter une couche si elle influence directement le nombre de sorties du réseau. Une telle pratique affecterait directement la performance et l'entraînement du modèle. Il pourrait y avoir un décalage entre les classes lors de la classification et la fonction de perte pourrait devenir invalide.

0.2 Pratiques à éviter

Lors de la construction du modèle, il est très important d'éviter les valeurs fixes lors de l'inférence. Par exemple, dans le modèle AlexNet, il y a le code de la partie gauche de la figure 2 qui nous empêche de réduire la dernière couche de convolution avant la couche pleinement connectée. En effectuant la modification présentée dans la figure 2, il est maintenant possible de réduire cette couche de convolution.



```
def forward(self, x):  
    x = self.features(x)  
    x = x.view(x.size(0), 256 * 6 * 6)  
    x = self.classifier(x)  
    return x
```

```
def forward(self, x):  
    x = self.features(x)  
    x = x.view(x.size(0), -1)  
    x = self.classifier(x)  
    return x
```

Figure 2: On remarque dans la partie de gauche que le code original assume une taille fixe pour la dernière couche de convolution. En plus de limiter le modèle à une certaine taille d'image en entrée on ne peut réduire la dernière couche de convolution. La solution proposée demande la modification dans la section de droite de l'image qui permet une taille variable de la dernière couche de convolution.

0.3 Propagation de l'effet

Lorsque nous retirons un filtre de convolution d'une couche, il faut prendre plusieurs précautions pour assurer la stabilité du système. Par exemple, il faut ajuster les poids, le nombre de biais dans la couche et ajuster les gradients lorsque ces derniers sont contenus dans la couche elle-même. Ce qui est le cas de PyTorch.

Par la suite, il faut ajuster les couches suivantes en fonction de leur type.

Pour les couches de **convolutions suivantes**. Il est nécessaire d'ajuster le nombre de filtres en entrées pour concorder avec la couche de convolution que nous traitons. Il faut aussi ajuster les poids de cette couche.

Pour les couches de **Batchnorm**, il faut ajuster les biais lorsqu'ils sont utilisés et nous effectuons un reset sur les variables utilisées à l'exécution. Dans PyTorch, nous parlons de running means et running vars.

Finalement, nous devons ajuster les couches **pleinement connectées**. Dans le cas de ces couches, nous ajustons les poids et nous appliquons la *weight inference rule* (8) afin de simuler le retrait des poids par *dropout*.

Les autres couches semblent bien supporter la réduction de couches de convolution. Il est à noter que nous avons uniquement essayé les modèles proposés dans PyTorch et que plus de travail pourrait être nécessaire pour des couches plus complexes.

0.4 Description des paramètres

Afin d'obtenir les résultats présentés. Les tests ont été faits sur l'ensemble de données CIFAR10 et les poids pré-entraînés de PyTorch ont été utilisés. La seule transformation appliquée aux images a été d'appliquer une normalisation avec une moyenne de [0.485, 0.456, 0.406] et une déviation standard de [0.229, 0.224, 0.225]. L'optimiseur utilisé était SGD, sans momentum, ni utilisation de Nesterov. Ainsi qu'un learning rate constant de 0.01.

Résultats

Dans un premier temps, nous évaluons l'effet de la réduction des filtres de convolution sur le modèle AlexNet. La figure 3 montre les effets de la réduction.

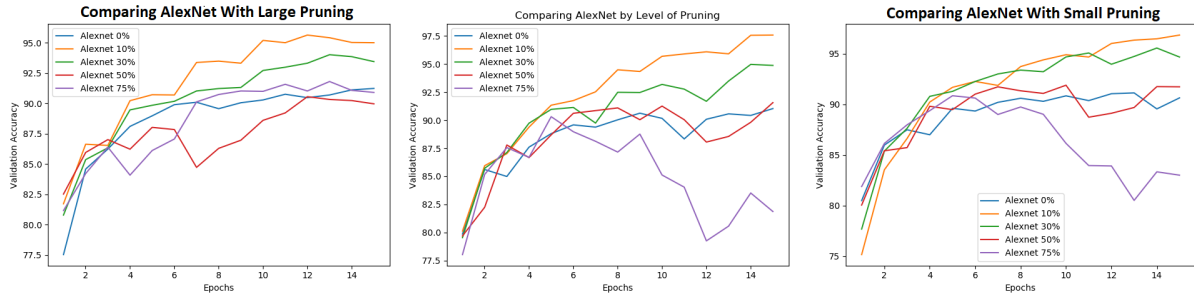


Figure 3: Dans la figure de gauche, on constate une réduction importante des poids à chaque itération(epoch 4 et 7). La figure du centre représente une réduction sur 4 itérations avec 2 étapes de remise à niveau. Sur la figure de droite, on voit une réduction plus progressive faite sur 10 itérations.

Table 1: Comparaison des résultats après la réduction des modèles

	0%	10%	30%	50%	75%
Fast Pruning	91.02	88.94	89.94	88.01	88.79
Average Pruning					
Slow Pruning	90.74	90.14	89.43	87.84	81.02

La figure 3 montre un aspect important. Nous remarquons que la manière de réduire les poids est importante. Une réduction trop rapide des poids va causer un plus grand bruit dans l'entraînement. Cependant le nombre d'epochs d'entraînement effectué après la dernière réduction semble essentiel pour obtenir de bons résultats. À l'opposé, nous remarquons qu'une réduction graduelle appliquée sur une plus longue période apporte plus de stabilité dans le modèle pour un certain temps, mais que le nombre d'itérations de réduction finit par réduire les performances finales du modèle. Une approche entre les deux obtient une bonne stabilité dans le modèle et les résultats finaux sont plus consistants.

Par la suite, nous comparons la différence d'efficacité entre les modèles. Cette différence est visible dans la figure 4 qui compare les modèles peu profonds dans la section de gauche et les modèles profonds de Pytorch dans le graphique de droite. Dans tous les cas, on remarque

que même avec une réduction de 30%. Il est possible de réduire significativement le réseau et maintenir un niveau de performance.

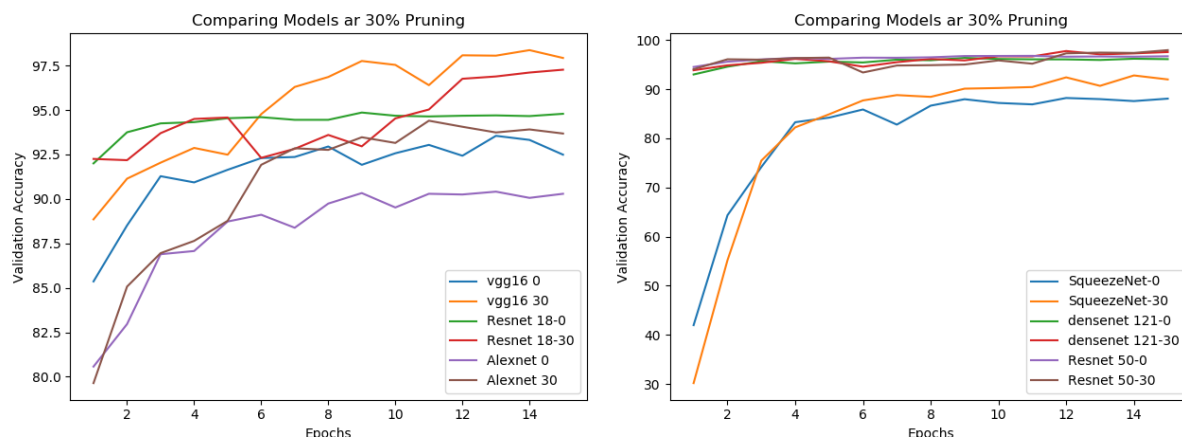


Figure 4: Dans l'image de gauche on peut observer l'effet de la réduction sur les modèles peu profonds. Dans l'image de droite on voit l'effet de la réduction sur les réseaux profonds de PyTorch.

La stratégie utilisée pour la réduction est de faire une réduction progressive. C'est à dire couper 5% filtres pouvant être réduit et répéter pour 6 itérations. Nous observons que cette stratégie donne de meilleurs résultats sur tous les modèles au niveau de la validation. Cependant on remarque que les score finaux sont légèrement inférieurs dans la majorité des cas.

Table 2: Comparaison de la réduction en fonction du modèle

	VGG			AlexNet			ResNet18		
	0%	30%	Diff	0%	30%	Diff	0%	30%	Diff
FLOPs (G)	17.31	10.55	-39.1%	0.815	0.500	-38.7%	1.83	1.40	-23.5%
Params (M)	138.3	80.5	-42.8%	57.0	47.9	-16.0%	11.18	7.53	-32.6%
Score (%)	91.92	92.03	0.11	90.98	88.57	-2.41	94.8	93.11	-1.69

	SqueezeNet			DenseNet121			ResNet50		
	0%	30%	Diff	0%	30%	Diff	0%	30%	Diff
FLOPs (G)	0.516	0.511	-0.97%	2.91	2.14	-26.4%	4.14	2.86	-30.9%
Params (M)	1.24	1.23	-0.81%	7.98	6.12	-23.3%	23.53	11.18	-52.5%
Score (%)	88.22	88.87	0.65	95.99	95.31	-0.68	96.32	94.76	-1.56

En examinant l'effet de la réduction sur les paramètres et le nombre de FLOPs dans la table 1. Nous remarquons qu'une réduction de 30% des filtres disponibles varie grandement d'un modèle à l'autre. Par exemple, dans SqueezeNet, il n'y a que la première couche de convolution qui peut être réduite. Ce qui explique une aussi petite réduction. À l'opposé, les réseaux du type ResNet offrent une excellente réduction, mais ont une dégradation de leur performance.

La perte ne semble pas directement relié au pourcentage de paramètres ou le nombre d'opérations flottantes retirés du réseau. Pour une réduction similaire VGG n'a aucune dégradation alors que ResNet18 montre une perte de 1.69%.

Conclusion

En conclusion nous remarquons: 1- La technique de réduction des filtres de convolution peut être appliqués sur des modèles plus complexes en créant un module analysant l'exécution du modèle. 2 - Les gains en réduction peuvent permettre de réduire les besoins matériels pour beaucoup d'architectures. 3 - La réduction de filtres de convolution offre une forme de régularisation pouvant servir à l'entraînement d'un modèle lorsque appliqué modérément. 4 - La manière d'appliquer la réduction est important afin d'obtenir de bons résultats. 5 - Certains modèles offrent très peu de réduction. 6 - Il reste du travail afin de supporter adéquatement les optimiseurs.

References and Notes

1. Y. LeCun et al. (1989). Backpropagation Applied to Handwritten Zip Code Recognition.
2. 30 amazing applications of deep learning: <http://www.yaronhadad.com/deep-learning-most-amazing-applications/> .
3. K. He, X. Zhang, S. Ren, and J. Sun (2015). Delving deep into rectifiers: Surpassing human-level performance on imagenet classification..
4. Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, Chunfang Liu (2018). A Survey on Deep Transfer Learning.
5. Song Han, William J. Dally (2018). Bandwidth-Efficient Deep Learning.
6. Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, Jan Kautz (2017). Pruning Convolutional Neural Networks for Resource Efficient Inference.
7. <https://pytorch.org/docs/stable/torchvision/models.html> .
8. Srivastava et al. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting..