Polytech - Deep Learning - TP 5-6
Réseaux convolutionnels pour l'image

Nicolas Thome Rémy Sun Corentin Dancette Matthieu Cord

Un compte-rendu des TPs 3, 4, 5 et 6 est à rendre pour le 3 Novembre 2022 (inclus) au plus tard. Instructions sur la page du site : `https://deep-learning-polytech.github.io`.

Objectifs

L'objectif de ce TME est de se familiariser avec les réseaux de neurones convolutionnels, très adaptés aux images. Pour se faire, nous étudierons les couches classiques de ce type de réseaux et nous mettrons en place un premier réseau appris sur le jeu de données standard CIFAR-10.

Une fois le premier réseau testé, nous aborderons différentes techniques permettant d'améliorer le processus d'apprentissage du réseau : normalisation, data augmentation, variantes de SGD, dropout et batch normalization.

Partie 1 – Introduction aux réseaux convolutionnels (1 heure)

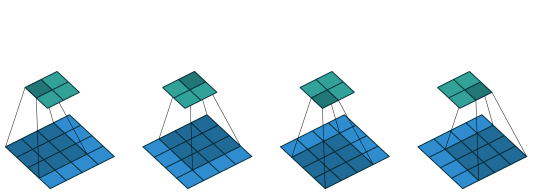
Les réseaux de neurones convolutionnels (CNN) sont devenus les architectures état-de-l'art dans la quasi-totalité des tâches de *machine learning* appliquées aux images. Ces réseaux diffèrent des réseaux plus classiques par les couches de base utilisées dans leur architecture. On y retrouve souvent deux types de couches différents : la convolution et le pooling.

1.1 Les couches de convolution

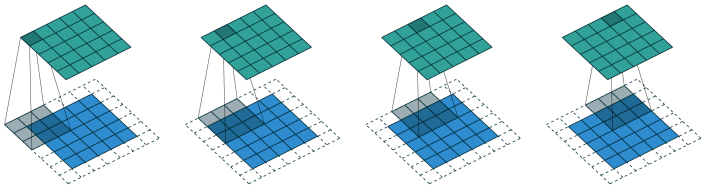
Entrée Ces couches prennent en entrée un ensemble de D feature maps (c'est-à-dire un tenseur soit l'équivalent en 3 dimensions d'une matrice), chaque feature map étant une matrice de taille $n_x \times n_y$. On a donc une entrée de taille $n_x \times n_y \times D$.

Sortie Sur cette entrée, on applique C convolutions, chacune avec un filtre de convolution de taille $w \times h \times D$, on a généralement $w = h = k$ qu'on appelle *taille du kernel*. Ce sont ces C filtres qui constituent les paramètres que l'on va apprendre. Chaque convolution avec un filtre produit une feature map de sortie, on a donc en sortie un ensemble de feature maps de taille $n'_x \times n'_y \times C$, où n'_x et n'_y dépendent de la façon dont on réalise la convolution.

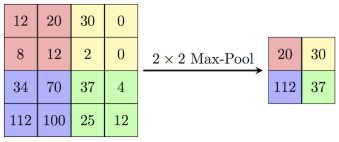
Hyperparamètres En plus du nombre de filtres et de la taille du kernel, il y a deux hyperparmètres courants pour la convolution, le **padding** et le **stride**. Le padding indique combien de rangées de 0 on ajoute autour de l'entrée. Le stride indique de combien de pas on se déplace entre deux calculs de la convolution. La convolution "classique" a une sortie plus petite que l'entrée à cause de la taille du filtre que l'on doit placer à l'intérieur de la feature map d'entrée, voir figure 1a. Ajouter du padding permet par exemple de retrouver la taille initiale, voir figure 1b. Ajouter un stride permet de sauter des valeurs, cela correspond à faire du sous-échantillonnage de la sortie, voir figure 1d.



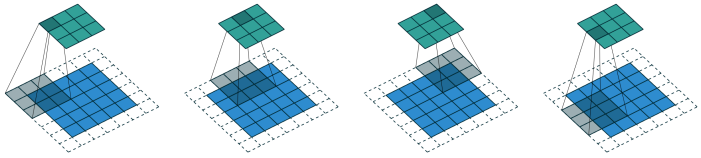
(a) Convolution simple, pas de padding, stride de 1, kernel de taille 3.



(b) Convolution avec padding de 1, stride de 1, kernel de taille 3.



(c) Exemple de max pooling, pas de padding, stride de 2, kernel de taille 2.



(d) Convolution avec padding de 1, stride de 2, kernel de taille 3.

FIGURE 1 – Illustrations de convolutions et pooling. Pour les convolutions, feature maps d'entrée en bas (1 seule ici, normalement plusieurs), feature map de sortie en haut.

Références Une démonstration de la convolution est visible à l'adresse `http://cs231n.github.io/assets/conv-demo/index.html`, et l'article de Dumoulin & Visin (2016) dont est tirée la figure 1 est très intéressant pour comprendre les convolutions et certaines variantes non présentées ici.

1.2 Couches de pooling

Principe Le pooling est une fonction de sous-échantillonnage spatial. Elle prend toujours en entrée des feature maps de taille $n_x \times n_y \times D$, et réduit les deux premières dimensions spatiales. Le calcul s'effectue selon le même principe qu'une convolution, mais s'applique sur chaque feature map indépendamment. On parcourt donc chaque feature map avec une fenêtre glissante, mais au lieu de réaliser un produit de convolution entre la fenêtre et un filtre, on réalise une opération de pooling sur la fenêtre d'entrée pour produire une valeur. On obtient donc une sortie de taille $n'_x \times n'_y \times D$ avec n'_x et n'_y significativement plus petits que n_x et n_y (souvent d'un facteur 2).

Hyperparamètres Une couche de pooling a une **taille de kernel** (définissant la taille de la fenêtre), un **stride** et du **padding** qui se comportent comme pour la convolution, et une **opération de pooling**, les plus courantes étant le *max pooling* (on garde la valeur maximale dans la fenêtre) et l'*average pooling* (on prend la valeur moyenne de la fenêtre). Un pooling très courant est un max pooling avec kernel de taille 2, stride de taille 2 et sans padding, comme présenté figure 1c.

1.3 Architectures convolutionnelles courantes

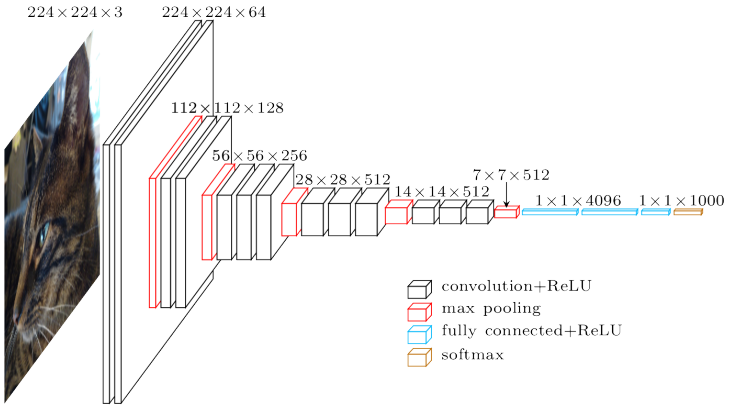


FIGURE 2 – Réseau VGG16.

Les réseaux de neurones convolutionnels classiques sont généralement composés d'une succession de couches de convolutions (avec ReLU) avec de plus en plus de filtres, et dont la dimension spatiale est progressivement réduite par des couches de max pooling possiblement jusqu'à aggregation totale des dimensions spatiales, il ne reste donc plus que la "profondeur" correspondant au nombre de filtres appliqués par la dernière convolution ($1 \times 1 \times C$). On y ajoute enfin généralement une ou quelques couches linéaires (appelées *fully-connected*). Un exemple de ce type d'architecture est le réseau VGG16 Simonyan & Zisserman (2014), *c.f.* Figure 2.

Depuis, des architectures plus complexes se sont développées, notamment les architectures Inception ou ResNet, et leurs dérivés et combinaisons.

1.4 Questions

1. Considérant un seul filtre de convolution de padding p, de stride s et de taille de kernel k, pour une entrée de taille $x \times y \times z$ quelle sera la taille de sortie ?
   Combien y a-t-il de poids à apprendre ?
   Combien de poids aurait-il fallu apprendre si une couche fully-connected devait produire une sortie de la même taille ?
2. Quels avantages apporte la convolution par rapport à des couches fully-connected ? Quelle est sa limite principale ?
3. Quel interêt voyez-vous à l'usage du pooling spatial ?
4. ★ Supposons qu'on essaye de calculer la sortie d'un réseau convolutionnel classique (par exemple celui en Figure 2) pour une image d'entrée plus grande que la taille initialement prévue (224×224 dans l'exemple). Peut-on (sans modifier l'image) calculer tout ou une partie des couches du réseau ?
5. Montrer que l'on peut voir les couches fully-connected comme des convolutions particulières.
6. Supposons que l'on remplace donc les fully-connected par leur équivalent en convolutions, répondre à nouveau à la question 4. Si on peut calculer la sortie, quelle est sa forme et son intérêt ?
7. On appelle champ récepteur (*receptive field*) d'un neurone l'ensemble des pixels de l'image dont la sortie de ce neurone dépend. Quelles sont les tailles des receptive fields des neurones de la première et de la deuxième couche de convolution ? Pouvez-vous imaginer ce qu'il se passe pour les couches plus profondes ? Comment l'interpréter ?

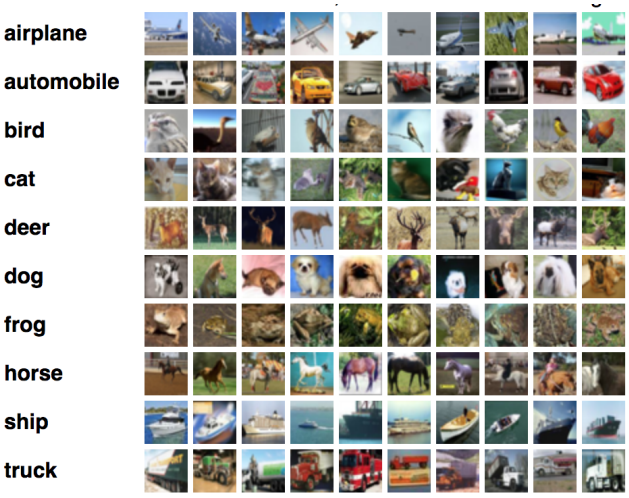


FIGURE 3 – Base de données CIFAR-10.

Partie 2 – Apprentissage *from scratch* du modèle (1 heure)

Nous allons désormais implémenter notre premier réseau convolutionnel que nous allons appliquer à la base de données CIFAR-10 (Krizhevsky, 2009, *c.f.* Figure 3). Cette base d'images RGB de 32×32 pixels comporte 10 classes, 50k images en *train* et 10k images en *test*.

2.1 Architecture du réseau

Le réseau que nous allons implémenter a un style proche de l'architecture AlexNet de Krizhevsky et al. (2012) adaptée à la base CIFAR-10 dont les images sont plus petites. Il sera composé des couches suivantes :

- `conv1` : 32 convolutions 5×5, suivie de ReLU
- `pool1` : max-pooling 2×2
- `conv2` : 64 convolutions 5×5, suivie de ReLU
- `pool2` : max-pooling 2×2
- `conv3` : 64 convolutions 5×5, suivie de ReLU
- `pool3` : max-pooling 2×2
- `fc4` : fully-connected, 1000 neurones en sortie, suivie de ReLU
- `fc5` : fully-connected, 10 neurones en sortie, suivie de softmax

Questions

8. Pour les convolutions, on veut conserver en sortie les mêmes dimensions spatiales qu'en entrée. Quelles valeurs de padding et de stride va-t-on choisir ?
9. Pour les max poolings, on veut réduire les dimensions spatiales d'un facteur 2. Quelles valeurs de padding et de stride va-t-on choisir ?
10. Pour chaque couche, indiquer la taille de sortie et le nombre de poids à apprendre. Commentez cette répartition.
11. Quel est donc le nombre total de poids à apprendre ? Comparer cela au nombre d'exemples.
12. Comparer le nombre de paramètres à apprendre avec celui de l'approche BoW et SVM.

2.2 Apprentissage du réseau

Pour l'apprentissage du réseau, partir du fichier `base.py` fourni. Ce fichier apprend un petit réseau convolutionnel historique, appelé LeNet5, sur la base de données MNIST. Commencez par lire et faire des tests avec ce code avant de le modifier pour faire ce que l'on veut.

Questions

13. Lisez et testez le code fourni. Vous pouvez lancer l'entrainement par cette commande : `main(batch_size, lr, epochs, cuda=True)`
14. Dans le code fourni, quelle différence importante y a-t-il entre la façon de calculer la loss et l'accuracy en train et en test (autre que les données sont différentes) ?
15. ★ Modifiez le code pour utiliser la base CIFAR-10 et implémenter l'architecture demandée ci-dessus. (la classe est `datasets.CIFAR10`). Attention à bien faire suffisament d'epoques pour que le modèle ait fini de converger. Qu'obtenez-vous à la fin de l'entrainement ?
16. ★ Quels sont les effets du pas d'apprentissage (*learning rate*) et de la taille de *mini-batch* ?
17. ★ A quoi correspond l'erreur au début de la première époque ? Comment s'interprète-t-elle ?
18. ★ Interpréter les résultats. Qu'est-ce qui ne va pas ? Quel est ce phénomène ?

Partie 3 – Améliorations des résultats (2 heures)

★ Pour cette partie, dans le rapport de fin de séance, indiquer les méthodes que vous avez réussi à implémenter et pour chacune expliquer en une phrase son principe et pourquoi cela permet d'améliorer l'apprentissage.

Nous allons maintenant voir plusieurs techniques classiques pour améliorer les performances de notre modèle.

3.1 Normalisation des exemples

Une technique courante en *machine learning* est de normaliser les exemples afin de mieux conditionner l'apprentissage. En apprentissage de CNN, la technique la plus courante est de calculer sur l'ensemble de train la valeur moyenne et l'écart type de chaque channel RGB. On obtient donc 6 valeurs. On normalise ensuite chaque image en soustrayant à chaque pixel la valeur moyenne correspondant à son channel et en le divant par l'écart type correspondant.

Pour CIFAR-10, les valeurs sont $\mu = [0.491, 0.482, 0.447]$ et $\sigma = [0.202, 0.199, 0.201]$.

Implémentation Ajouter la normalisation dans le pré-processing des données.
Cela s'effectue lors de l'appel à `datasets.CIFAR10` :

```
train_dataset = datasets.CIFAR10(path, train=True, download=True,
transform=transforms.Compose([
        # Ensemble de transformations à appliquer, on a en entrée une image PIL
        ↪ (Python Image Library). Se référer à la documentation PyTorch dans le
        ↪ package torchvision.transforms
        transforms.ToTensor() # Transforme l'image PIL en torch.Tensor
        # Ajouter la normalisation avec les valeurs trouvées ci-dessus
]))
```

Questions

19. ★★ Décrire vos résultats expérimentaux.
20. Pourquoi ne calculer l'image moyenne que sur les exemples d'apprentissage et normaliser les exemples de validation avec la même image ?
21. **Bonus :** Il existe d'autres schémas de normalisation qui peuvent être plus efficaces comme la normalisation ZCA. Essayer d'autres méthodes, expliquer les différences et comparer les à celle demandée.

3.2 Augmentation du nombre d'exemples d'apprentissage par *data augmentation*

Les réseaux de convolutions contiennent souvent plusieurs millions de paramètres et sont appris sur de très grosses bases d'images. CIFAR-10 est un ensemble d'images relativement petit par rapport au nombre de paramètres du modèle. Pour contrer ce problème, on utilise des méthodes de *data augmentation* qui cherchent à artificiellement augmenter le nombre d'exemples disponibles, ce qui peut être crucial quand l'ensemble d'apprentissage est petit.

Le principe consiste à générer à chaque *epoch* une "variante" de chaque image, en lui appliquant des transformations aléatoires. Ici, on propose de tester deux transformations les plus courantes : un crop aléatoire de taille 28×28 pixels (parmi les 32×32) et une symétrie horizontale aléatoire de l'image (1 chance sur 2 de l'appliquer).

Notons que ces transformations sont appliquées à l'ensemble de train, mais il faut parfois ajuster la phase d'évaluation pour en tenir compte. Par exemple ici, on a réduit la taille de nos images d'apprentissage. On pourrait penser à plusieurs stratégies pour contrer cela, en particulier : modifier le modèle pour prendre des images plus petites et faire un crop centré sur les images de test ou ajouter du padding autour des images de train pour revenir à une taille 32×32.

Pour conserver le même nombre de paramètres dans le modèle (afin que nos résultats soient plus facilement comparables), on propose de modifier la couche `pool3` en ajoutant l'option `ceil_mode=True` pour que la taille spatiale de sortie soit toujours 4×4.

Implémentation

— Modifier l'appel à `datasets.CIFAR10` pour ajouter en *train* un crop aléatoire taille 28 et une symétrie horizontale aléatoire ; et en *test* un crop centré taille 28.
— Modifier la couche `pool3` pour ajouter l'option `ceil_mode=True`.

Questions

22. ★★ Décrire vos résultats expérimentaux et les comparer aux résultats précédents.
23. Est-ce que cette approche par symétrie horizontale vous semble utilisable sur tout type d'images ? Dans quels cas peut-elle l'être ou ne pas l'être ?
24. Quelles limites voyez-vous à ce type de data augmentation par transformation du dataset ?
25. **Bonus :** D'autres méthodes de data augmentation sont possibles. Chercher lesquelles et en tester certaines.

3.3 Variantes sur l'algorithme d'optimisation

Jusqu'à maintenant, on a utilisé un algorithme d'optimisation simple, la SGD. Il existe de nombreuses variantes de cet algorithme permettant une convergence plus rapide ou meilleure.

Une stratégie consiste à modifier le *learning rate* au cours de l'apprentissage (généralement pour le diminuer), par exemple, on peut appliquer une décroissance exponentielle à η selon la formule $\eta_t = \eta \times m^t$ où t est l'*epoch* et m le momentum.

```python
# Importer le package
import torch.optim.lr_scheduler
# [...]
# Créer le scheduler
optimizer = ...
lr_sched = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.95)
# [...]
# Modifier le learning rate après chaque epoch
epoch(...)
lr_sched.step()
```

Implémentation

— Ajouter un *learning rate scheduler* et utiliser une décroissance expoentielle avec un coefficient de 0.95.

Questions

26. ★★ Décrire vos résultats expérimentaux et les comparer aux résultats précédents, notamment la stabilité de l'apprentissage.
27. Pourquoi cette méthode améliore-t-elle l'apprentissage ?
28. **Bonus :** De nombreuses autres variantes de la SGD existent et de nombreuses stratégies de planification de *learning rate* existent. Lesquelles ? En tester certaines.

3.4 Régularisation du réseau par *dropout*

On a vu à la question 10 que la couche `fc4` du réseau contient un très grand nombre de poids, le risque de sur-apprentissage dans cette couche est donc important. Pour remédier à ce problème, nous allons ajouter une couche permettant une régularisation très efficace pour les réseaux de neurones : la couche de *dropout*. Le principe général de cette couche est de désactiver aléatoirement à chaque passe une partie des neurones du réseau pour diminuer artificiellement le nombre de paramètres. Toutes les informations nécessaires sur cette couche se trouvent dans l'article correspondant de Srivastava et al. (2014).

Implémentation Ajouter une couche de dropout entre `fc4` et `fc5`.

Questions

29. ★★ Décrire vos résultats expérimentaux et les comparer aux résultats précédents.
30. Qu'est-ce que la régularisation de manière générale ?
31. Cherchez et "discutez" des possibles inteprétations de l'effet du dropout sur le comportement d'un réseau l'utilisant ?
32. Quelle est l'influence de l'hyper-paramètre de cette couche ?
33. Quelle est la différence de comportement de la couche de *dropout* entre l'apprentissage et l'évaluation ?

3.5 Utilisation de *batch normalization*

Une dernière statégie d'apprentissage que nous allons tester est la *batch normalization* de Ioffe & Szegedy (2015). Il s'agit d'une couche qui apprend à renormaliser les sorties de la couche précédente, permettant de stabiliser l'apprentissage.

Implémentation Ajouter une couche de *batch normalization* 2D immédiatement après chaque couche de convolution.

Question

34. ★★ Décrire vos résultats expérimentaux et les comparer aux résultats précédents.

Références

Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv*, 2016.

Sergey Ioffe and Christian Szegedy. Batch normalization : Accelerating deep network training by reducing internal covariate shift. In *Proceedings of the 32nd International Conference on Machine Learning*, 2015.

Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*. 2012.

K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv*, 2014.

Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout : a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research (JMLR)*, 2014.