Reconnaissance des formes pour l'analyse et l'interprétation d'images

Réseaux convolutionnels pour l'image.

Loüet Joseph & Karmim Yannis
Spécialité **DAC**

Table des matières

1 Introduction aux réseaux convolutionnels

Questions

1) On a comme nouvelles tailles :
— $n'_x = \left\lfloor \frac{x-k+2p}{s} + 1 \right\rfloor$
— $n'_y = \left\lfloor \frac{y-k+2p}{s} + 1 \right\rfloor$
— $n'_z = C$
Le nombre de poids à apprendre pour une convolution à un seul filtre est : $k^2 \times z + 1$
Le nombre de poids à apprendre en Fully Connected à un seul filtre : $x \times y \times z \times n'_x \times n'_y$

2) Les avantages de la convolution par rapport au Fully Connected sont que le nombre de paramètres à apprendre pour une convolution est inférieur que pour le Fully Connected et que l'on garde une forme de structure. En revanche, sa limite principale est la perte d'information.

3) L'intérêt du pooling spatial est de diminuer les tailles n_x et n_y et donc réduire le nombre de paramètres tout en résumant l'informations des différentes zones.

4) Si l'on prend un image plus grande, cela ne change pas le nombre de poids à apprendre pour les filtres de convolutions puisqu'ils ne dépendent pas des dimensions x et y. En revanche, pour les couches Fully Connected, comme le nombre de poids à apprendre dépendent des dimensions de l'entrée, ces couches ne pourront correspondre correctement puisque les sorties des filtres de convolution seront plus grandes.

5) Les couches Fully Connected sont des convolutions telles que la taille du kernel correspond à la taille des entrées et que que le nombre de filtre de convolutions correspond aux dimensions $n'_x \times n'_y$ de la sortie.

6) Si l'on remplace les Fully Connected par leur équivalent en convolutions, on peut alors calculer la sortie mais cette dernière sera de plus grande dimension et cela correspondra à un Fully Connected sur des sous-parties des entrées. Cela correspondrait au même réseau que précédemment appliqué sur chacune des parties de la taille de l'image initiale, comme une convolution dont le stride serait équivalent au stride de l'équivalent Fully Connected.

7) Les tailles des *receiptive field* de la première couche de convolution sont $k_1 \times k_1$ où k_1 est la taille du kernel de la première convolution. Ces tailles peuvent varier en fonction du padding et du neurone considéré (sur une bordure ou dans un coin). Si le neurone considéré prend en compte des valeurs du padding, alors la taille du *receiptive field* est inférieur. On obtient donc les équations suivantes :

$$
\begin{aligned}
n_{out} &= \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1 \\
j_{out} &= j_{in} * s \\
r_{out} &= r_{in} + (k-1) * j_{in}
\end{aligned}
$$

— La première équation calcule le nombre de d'éléments en sortie en fonction du nombre d'éléments en entrée, le padding, le stride et la taille du kernel.
— La deuxième équation calcule le *jump* qui est le nombre d'éléments que l'on saute en fonction du stride et des strides précédent (et donc du *jump* des couches précédentes)
— La troisième équation calcule la taille des *receiptive field* en fonction de la taille des *receiptive field* de la couche précédentes.

De ce fait, on remarque que si $k > 1$ et $j_{in} > 0$ alors la taille des *receiptive field* est croissante. Donc, plus on avance sur des couches plus profondes, plus le nombre de pixels considéré est grand.

2 Apprentissage *from scratch* du modèle

2.1 Architecture du réseau

Questions

8) On rappelle les équations des tailles de sortie :

$$
n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1
$$

Si l'on veut $n_{in} = n_{out} (= n)$, alors :

$$
\begin{aligned}
s &= p \times \frac{2}{n-1} + \frac{n-k}{n-1} \\
p &= s \times \frac{n+1}{2} - \frac{n-k}{2}
\end{aligned}
$$

9) De même que précédemment, on veut $n_{out} = \frac{n_{in}}{2}$, alors :

$$
\begin{aligned}
s &= p \times \frac{2}{\frac{n_{in}}{2} - 1} + \frac{n_{in} - k}{\frac{n_{in}}{2} - 1} \\
p &= s \times \frac{\frac{n_{in}}{2} - 1}{2} - \frac{n_{in} - k}{2}
\end{aligned}
$$

Généralement, pour réduire la dimension par deux, nous faisons un max pooling avec un kernel de taille 2, un stride de 2 et aucun padding.

10)

— Pour la première couche, nous avons 32 convolutions avec une taille de kernel égale à 5 et une image de taille $32 \times 32 \times 3$. Ainsi le nombre de poids à apprendre pour cette couche est : $(5^2 \times 3 + 1) \times 32 = 2432$ et une taille de sortie $16 \times 16 \times 32$ avec le max pooling réduisant la hauteur et la largeur de notre sortie par 2.
— Pour la deuxième couche, nous avons 64 convolutions avec une taille de kernel égale à 5 et une sortie de la couche précédente de taille $16 \times 16 \times 32$. Ainsi le nombre de poids à apprendre pour cette couche est : $(5^2 \times 32 + 1) \times 64 = 51264$ et une taille de sortie qui est $8 \times 8 \times 64$ avec le max pooling réduisant la hauteur et la largeur de notre sortie par 2.
— Pour la troisième couche, nous avons 64 convolutions avec une taille de kernel égale à 5 et une sortie de la couche précédente de taille $8 \times 8 \times 64$. Ainsi, le nombre de poids à apprendre pour cette couche est : $(5^2 \times 64 + 1) \times 64 = 102464$ et une taille de sortie qui est $4 \times 4 \times 64$ avec le max pooling réduisant la hauteur et la largeur de notre sortie par 2.
— Pour la quatrième couche, nous avons un fully-connected avec 1000 neurones en sortie suivi d'un ReLU. Ainsi, le nombre de poids à apprendre est $4 \times 4 \times 64 \times 1000 = 1024000$. La sortie est un vecteur de taille 1000 puisque le ReLU ne change pas la taille de sortie.
— Pour la cinquième couche, nous avons un fully-connected avec 10 neurones en sortie suivi d'un Softmax. Ainsi, le nombre de poids à apprendre est $1000 \times 10 = 10000$. La sortie est un vecteur de taille 10 puisque le Softmax ne change pas la taille de sortie.

11) Le nombre total de poids à apprendre est donc $2432 + 51264 + 102464 + 1024000 + 10000 = 1\ 190\ 160$. Nous pouvons ce nombre au nombre d'exemples en apprentissage qui comporte 50 000 de taille $32 \times 32 \times 3$ ce qui fait 51 200 000 pixels codés en RGB.

12) Si l'on prend 1000 descripteurs et 10 classes, le nombre de poids à apprendre est $(1000 + 1) \times 10 = 10\ 010$ (Vecteur de taille 1000 et le biais pour les 10 SVMs).

2.2 Apprentissage du réseau

Questions

14) La différence entre le calcul de la loss et de l'accuracy en train et en évaluation est qu'on ne mets pas à jour les paramètres de notre réseau avec une backpropagation dans le cas d'une évaluation.

16) Les effets du pas d'apprentissage et de la taille du mini-batch est la vitesse de convergence de notre modèle. Ce sont des hyper-paramètres de notre modèle qu'il faut ajuster, surtout pour le pas d'apprentissage, la taille du mini-batch n'est généralement pas très importante. Il existe des optimiseurs comme ADAM par exemple qui adapte le pas d'apprentissage pour une convergence plus efficace.

17) L'erreur de la première époque correspond à l'erreur du réseau qui est parti de poids initialisé aléatoirement. Notre modèle fait donc au début des prédictions très aléatoires qu'il corrige au fur et à mesure.

18) À partir d'une certaine époque, le réseau sur-apprend, on a des très bons résultats en apprentissage mais cela s'effondre en test.

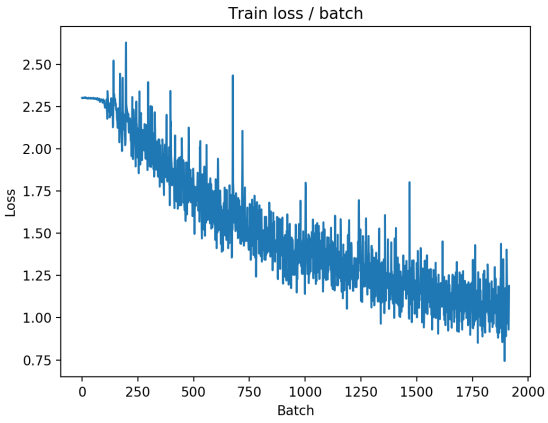

FIGURE 1

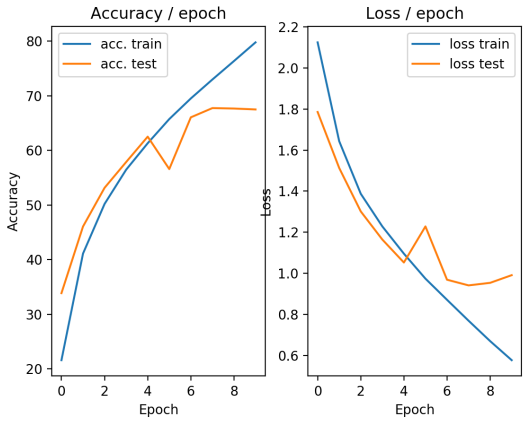

FIGURE 2 – Courbe de loss et d'accuracy pour les données CIFAR.

3 Amélioration des résultats

3.1 Normalisation des exemples

Questions

19) Résultats :
On a un meilleur comportement lors des 5 premières époques, avec de meilleur

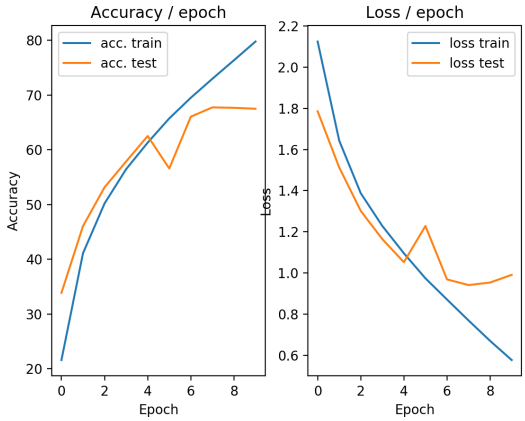

FIGURE 3 – Courbe de loss et d'accuracy pour les données CIFAR avec normalisation des données et 6 EPOCHS.

score en précision top1 et top5. Le modèle surapprend ensuite à partir de la 6ème époque.

20) On calcul la normalisation uniquement sur l'ensemble d'apprentissage et on utilise ces mêmes valeurs pour la validation puisque normalement l'ensemble de validation doit permettre de tester notre modèle, et nous ne devons pas exploiter ses données.

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

Questions

22) On a augmenté les données en appliquant un crop aléatoire sur nos données et une symétrie horizontale avec une probabilité de 0.5.
Résultats :

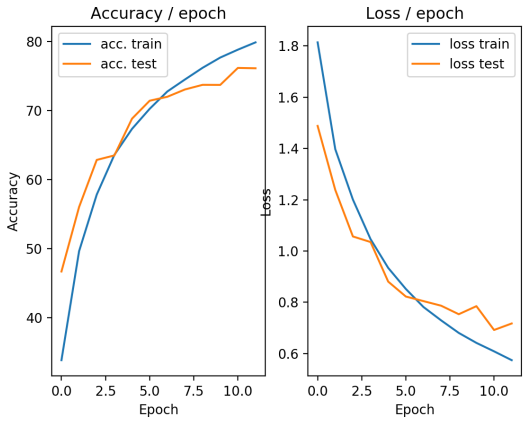

FIGURE 4 – 12 epochs d'apprentissage avec *data augmentation* sur les données CIFAR10.

La data augmentation est efficace pour notre modèle, puisque même avec plus d'epochs on sur-apprend beaucoup moins que dans nos résultats précédents. On peut donc atteindre une accuracy en évaluation plus élevée.

23) La symétrie horizontale ne peut pas être appliqué à toute les images, par exemple pour la reconnaissance de lettre ou de nombres cela va fausser nos données. En revanche pour la reconnaissance d'objet ou d'animaux cela peut être une bonne méthode d'augmentation des données.

24) Le risque de biaiser et fausser les images est important avec de la *data augmentation*. Il faudra également appliquer un prétraitement à nos données en test pour chaque image.

25) Il existe d'autres techniques de *data augmentation* comme la rotation, la translation ou encore l'ajout de bruit dans l'image.

3.3 Variantes sur l'algorithme d'optimisation

Questions

26) Nous avons modifié le code afin d'ajouter un momentum de 0.9 à notre optimiseur SGD et d'utiliser un learning rate scheldurer pour notre descente de gradient.

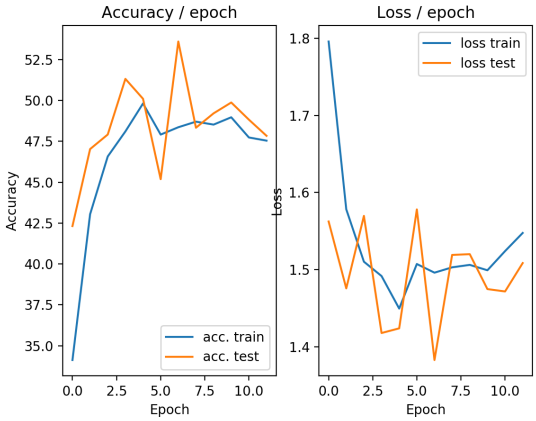

FIGURE 5 – 12 epochs d'apprentissage avec learning rate scheldurer et momentum de 0.9 sur les données CIFAR10.

Avec un momentum de 0.9 notre modèle a du mal à converger, on en prend un plus petit afin d'éviter ce type d'oscillations.

27) Pour la descente de gradient stochastique il est fréquent que l'on oscille autour du minimal local, et que l'on prenne donc plus de temps pour converger. Le **momentum** accélère la convergence en conservant la mise à jour effectué au pas précédent et en la pondérant avec un terme $\gamma < 1$. Quand la nouvelle mise à jour est dans la même direction que la précédente alors on accélère la descente, sinon lorsque on est dans un sens différent ou opposé on va diminuer cette descente.

Le learning rate scheldurer est un pas d'apprentissage adaptatif, il est parfois utile d'avoir un grand pas d'apprentissage au début de l'entraînement afin de s'extraire d'un minimum local. En revanche, au fur et à mesure de notre apprentissage on voudrait que notre pas d'apprentissage se réduise afin de pouvoir converger. On a donc plus de chance avec cette méthode de trouver le minimal global de notre fonction de coût.

28) L'optimiseur le plus courant et le plus utilisé est l'optimiseur ADAM qui est efficace pour la plupart des tâches en Deep Learning. Il existe également l'optimiseur avec pas adaptatif Adagrad, adapté pour des données sparses.

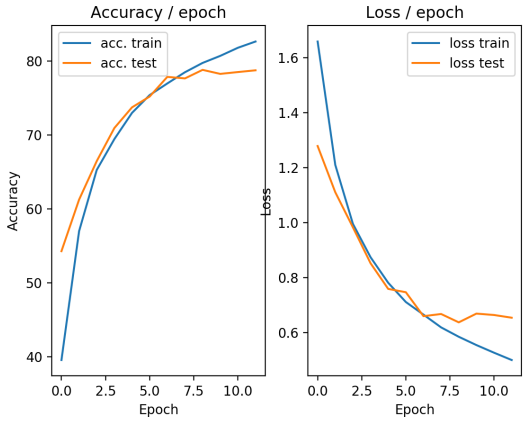

FIGURE 6 – 12 epochs d'apprentissage avec learning rate scheldurer et momentum de 0.5 sur les données CIFAR10.

3.4 Régularisation du réseau par *dropout*

Questions

29) Résultats :
On remarque que contrairement à notre résultat précédent notre modèle sur apprend beaucoup moins, le dropout est donc efficace pour la régularisation de notre réseau.

30) La régularisation est un processus consistant à ajouter de l'information à un problème pour éviter le surapprentissage. Cette information prend généralement la forme d'une pénalité envers la complexité du modèle. Cela peut également consister à pénaliser les valeurs extrêmes, ce qui correspond souvent à un surapprentissage.

31) Pour rappel, la couche de dropout permet de désactiver aléatoirement à chaque passe une partie des neurones du réseau pour diminuer artificiellement le nombre de paramètres. Ainsi, cette couche empêche le surapprentissage car elle empêche que certains noeuds du réseau soient plus influent que d'autres dans la prédiction. En effet, si les prédictions s'effectuent principalement sur quelques noeuds du réseau, alors nous nous exposons à du surapprentissage car notre réseau aura peu de flexibilité. Si l'on désactive ces noeuds dominants durant l'apprentissage, cela va obliger aux autres noeuds de prendre en importance et donc éviter le surapprentissage. De plus, avec cette couche, les poids du réseau seront supérieurs que sans. Dans des cas ou nous n'avons pas beaucoup de données, le dropout permet de rendre le réseau plus robuste à des transformations des images (translations,rotations...).

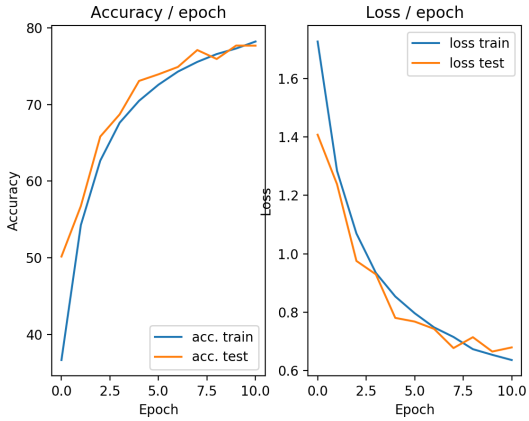

FIGURE 7 – 12 epochs d'apprentissage avec régularisation *dropout* sur les données CIFAR10.

32) L'hyper-paramètre de cette couche permet d'ajuster le pourcentage de noeud que l'on va désactiver. Si cet hyper-paramètre est trop petit alors on augmente le risque de sur-apprentissage car les noeuds dominants seront moins souvent désactivés. En revanche, si l'hyper-paramètre est trop grand, les poids du réseau seront largement supérieurs que normalement et donc le réseau final seront plus instable. Selon la dernière référence du TP, la valeur optimal s'approche de 0,5 et pour le cas où toutes les couches ont le même nombre de neurones, les résultats sont correctes si $0.4 < p < 0.8$ où p est l'hyper-paramètre.

33) Pendant l'apprentissage, la couche de dropout désactive aléatoirement à chaque passe une partie des neurones du réseau. En revanche, lors de la validation, la couche de dropout n'aura aucun effet sur le réseau, elle ne désactivera pas des neurones du réseau. C'est pour cela que nous utilisons `model.train()` et `model.eval()` qui permet de spécifier à Pytorch que nous sommes soit en entraînement, soit en validation.

3.5 Utilisation du batch *normalization*

Questions

34) On expérimente cette fois la batch normalization. Le principe est de ne plus normaliser sur l'ensemble des données, mais uniquement de centrée réduire sur les exemples que l'on a dans notre batch. Cela permet à chaque couche d'apprendre plus indépendament des autres couches du réseau, et évite ainsi le sur-apprentissage.
Résultats :

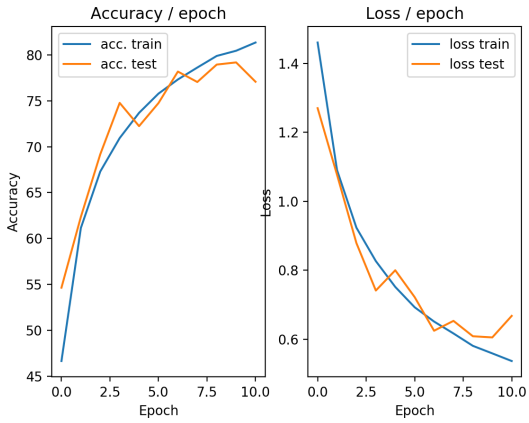

FIGURE 8 – 12 epochs d'apprentissage avec régularisation *batch normalization* sur les données CIFAR10.