

Les réseaux de neurones simples ne sont pas adaptés à la reconnaissance d'images. Si chaque pixel devait être associé à un poids à optimiser, les paramètres seraient trop nombreux pour converger lors de l'apprentissage. Il faut donc plutôt passer par des réseaux convolutionnels et cette approche Deep Learning, même difficile à paramétrier, a des avantages sur la reconnaissance BoW du TP 1-2.

Considérations théoriques

Une image s'étudie en 3D, avec des dimensions planes qui lui sont propres et un nombre de canaux multiple (souvent 3 pour RGB). Cette troisième dimension ne gêne pas l'apprentissage ; le seul secret des réseaux convolutionnels est leur façon de considérer la *surface* des images.

Dans une couche convolutionnelle, chaque neurone est responsable d'une *région* de l'entrée. Cette connectivité locale – là où un réseau classique est fully-connected (FC) – diminue le poids calculatoire des couches et permet d'approfondir le réseau en maintenant un nombre décent de paramètres.

En complément, des couches de *pooling* fournissent leur aide en résumant l'information contenue dans les entrées et contribuent donc à faciliter les calculs en réduisant directement leur taille.

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 FC devait produire une sortie de même taille ?

La taille (x_c, y_c) de la sortie est $\left\lfloor \frac{x-k+2p}{s} \right\rfloor + 1, \left\lfloor \frac{y-k+2p}{s} \right\rfloor + 1$. Sa dimension z_c est égale au nombre de filtres, ici 1.

Dans une couche de convolution, pour chaque filtre de kernel k, $x_c \times y_c$ neurones observent séparément une région de taille k^2 de l'entrée (avec z canaux) et traitent ces $k^2 \times z$ pixels dont ils sont responsables. Mais comme ils partagent leurs paramètres, il y a seulement $k^2 \times z$ poids et un biais à apprendre par filtre.

Dans une couche FC, chaque neurone reçoit tous les pixels de l'image en entrée, ce qui fait $x \times y \times z$ poids à donner. Il faut par ailleurs $x_c \times y_c$ neurones pour que la sortie fasse la même taille que ci-dessus et chacun doit gérer séparément son apprentissage ; cela fait $x \times y \times z \times x_c \times y_c$ poids et $x_c \times y_c$ biais à apprendre.

C'est la dépendance des poids des couches FC à la taille de l'image (présence de x et y dans l'équation) qui démultiplie les paramètres à apprendre et compromet leur application.

2 Quels avantages apporte la convolution par rapport aux couches FC ? Quelle est sa limite principale ?

La convolution réduit considérablement le nombre de poids à apprendre. Cela diminue le temps de calcul et augmente les chances de convergence. Elle permet aussi de traiter la totalité de l'image d'une façon sensée (région par région), et conserve donc la cohérence spatiale de l'image ; elle peut détecter des patterns locaux alors qu'une couche FC ne voit que des pixels indépendants.

Cependant, la localité peut aussi être un désavantage. On ne s'intéresse qu'à des régions de l'image, et on ne peut donc pas avoir une vision générale ; cela peut être compensé par un choix de kernel plus grand.

3 Quel intérêt voyez-vous à l'usage du pooling spatial ?

Le pooling spatial sert de résumé informatif sur les régions de l'image. Il permet de réduire la taille $x_p \times y_p$ de l'output par rapport à l'original en condensant l'information (en faisant un maximum ou une moyenne par région). Les convolutions ultérieures auront donc moins de données à traiter en quantité absolue, ce qui réduit les calculs à faire ; de plus, comme c'est une concentration et non une suppression, chaque élément de sortie correspondra à une vision globale et informera sur une zone étendue dans l'image de départ.

4 Supposons qu'on essaye de calculer la sortie d'un réseau convolutionnel pour une image d'entrée plus grande que la taille prévue. Peut-on calculer tout ou une partie des couches du réseau ?

→ Les paramètres appris pour une couche de convolution dépendent seulement de la taille du kernel. Si ce kernel ne change pas, on peut donc se servir de la même couche pour un input plus grand ; la sortie sera aussi plus grande, mais cela ne dérange pas les couches suivantes tant qu'elles sont du même type.

→ De même pour les couches de pooling qui ne dépendent d'aucun paramètre.

→ Par contre, les couches FC attendent une taille d'input précise. Comme elles apprennent des poids par pixel, elles ne peuvent pas gérer une entrée de taille différente ; impossible de mapper les poids dessus si elle est plus grande que prévu lorsqu'elle sort des convolutions précédentes.

Le calcul sur une image plus grande devra donc s'arrêter au niveau des couches FC. On pourra tout de même arriver jusqu'à elles, car elles sont généralement placées ensemble à la fin du graphe de calcul.

5 Montrer que l'on peut voir les couches fully-connected comme des convolutions particulières.

Les couches FC correspondent à des convolutions pour une taille de kernel égale à la taille de leur entrée ($x \times y \times z$), sans padding, et un nombre de filtres égal à la taille désirée de la sortie (n).

Cela donne bien le même nombre de paramètres, et ils correspondent aux mêmes éléments de l'input :

→ La couche FC associe ($x \times y \times z$) poids à chacun de ses neurones (un par pixel). Elle contient n neurones en tout (où n est un nombre arbitrairement choisi), ce qui fait en tout $(x \times y \times z) \times n$ paramètres.

→ Une couche convolutionnelle associe $k_1 \times k_2 \times z$ poids aux neurones de chaque filtre (cf. Q1), où $k_1 \times k_2$ correspond à la taille du kernel du filtre. Si l'on décide que $k_1 = x$ et $k_2 = y$, et qu'il y a n filtres, elle aura donc au total $(x \times y \times z) \times n$ paramètres comme prévu. Le kernel étant étalé sur tout l'input, il ne sera appliqué qu'une fois, et le calcul de chaque neurone correspondra à un pixel initial ; la connexion locale s'étend sur une taille de région égale à celle de l'image. Il y a donc bien aussi une égalité sémantique.

6 Supposons que l'on remplace les FC par leur équivalent en convolutions, répondre de nouveau à la question 4. Si on peut calculer la sortie, quelle est sa forme et son intérêt ?

Les couches FC étaient le facteur bloquant pour appliquer un réseau sur une image plus grande. On peut donc désormais finir le calcul. La sortie obtenue sera plus grande.

Si la tâche est une classification, donc avec un output initial scalaire, on obtient maintenant plusieurs chiffres (distribués en 2D) qui correspondent à plusieurs régions de l'image et qui peuvent être interprétés comme des prédictions associées à chacune.

7 On appelle champ récepteur d'un neurone l'ensemble des pixels de l'image dont sa sortie dépend. Quelles sont les tailles des receptive fields des neurones de la première et de la deuxième couche de convolution ? Que se passe-t-il pour les couches plus profondes ? comment l'interpréter ?

La taille du champ récepteur pour la première couche de convolution est k_1^2 , soit la taille du kernel des premiers filtres. Dans la deuxième couche, un autre kernel sert à filtrer la sortie de la première, la taille du champ récepteur des neurones est donc k_2^2 ; mais rapporté à l'image originale, c'est un carré de côté

$$k_1 + (k_2 - 1) * s_1 \quad \text{où le stride } s_1 \text{ de la première doit être pris en compte.}$$

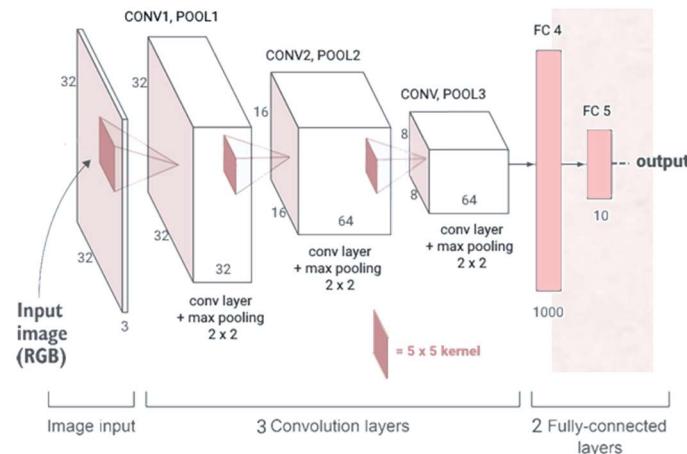
La formule est une somme d'une couche à l'autre, donc la taille du champ récepteur ne peut qu'augmenter : il y a une tendance de « dézoom » qui s'accentue avec la profondeur. Si les premières couches voient des features très locales, comme des orientations de lignes (les poids appris peuvent très bien être des filtres de Sobel...) sur quelques pixels, les dernières couches dans le réseau détecteront forcément des features plus grandes – donc potentiellement des objets avec une sémantique haut niveau, comme des visages.

Architecture du réseau

On essaie maintenant d'assembler des couches convolutionnelles et de pooling de façon cohérente pour un problème concret : on souhaite traiter la base CIFAR-10 pour la reconnaissance d'objets.

L'architecture du réseau est donnée, ce qui nous épargne les recherches nécessaires à son design.
Etant donnée sa similarité au réseau AlexNet, on a adapté ci-contre une illustration tirée de www.kaggle.com/l0new0lf/alexnet qui nous aide à le visualiser.

Figure 1 – Réseau utilisé pour l'implémentation →



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 ?

Pour qu'une convolution conserve la taille des images, on reprend les équations données en Q1 : il faut que $\left\lfloor \frac{x-k+2p}{s} \right\rfloor + 1 = x$ et que $\left\lfloor \frac{y-k+2p}{s} \right\rfloor + 1 = y$. La taille du kernel k étant fixée (et impaire par convention), on se simplifie la vie en fixant un stride s de 1 ; un padding p de taille $\frac{k+1}{2}$ remplit alors les deux conditions.

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 ?

La sortie d'un max pooling sur l'image est aussi de taille $\left\lfloor \frac{x-k+2p}{s} \right\rfloor + 1, \left\lfloor \frac{y-k+2p}{s} \right\rfloor + 1$. Pour trouver p et s de sorte que $\left\lfloor \frac{x-k+2p}{s} \right\rfloor + 1 = \frac{x}{2}$ et $\left\lfloor \frac{y-k+2p}{s} \right\rfloor + 1 = \frac{y}{2}$, on décide arbitrairement d'utiliser un padding nul ; on doit alors résoudre $\left\lfloor \frac{x-k}{s} \right\rfloor = \frac{x}{2}$, ce qui fonctionne si $k = s = 2$. Même raisonnement pour y. On doit donc utiliser, avec un padding nul, un kernel de 2 et un stride de 2.

10 Pour chaque couche, indiquez la taille de sortie et le nombre de poids. Commentez cette répartition.

		Fonctionnement	Nombre de poids liés	Taille de la sortie
au départ	entrée			
passée à	conv1	32 filtres 5 x 5	$5 \times 5 \times 3 \times 32 = 2400$	de taille 32 x 32 x 3
passée à	pool1	max 2 x 2 (red. de 2)	aucun poids à apprendre	sortie 32 x 32 x 32
passée à	conv2	64 filtres 5 x 5	$5 \times 5 \times 32 \times 64 = 51200$	sortie 16 x 16 x 32
passée à	pool2	max 2 x 2 (red. de 2)	aucun poids à apprendre	sortie 8 x 8 x 64
passée à	conv3	64 filtres 5 x 5	$5 \times 5 \times 64 \times 64 = 102400$	sortie 8 x 8 x 64
passée à	pool3	max 2 x 2 (red. de 2)	aucun poids à apprendre	sortie 4 x 4 x 64
passée à	fc4	1000 neurones	$4 \times 4 \times 64 \times 1000 = 1024000$	sortie 1000 x 1
passée à	fc5	10 neurones	$1000 \times 10 = 10000$	sortie 10 x 1

et le calcul se termine avec un softmax sur les 10 éléments finaux.

→ La taille de la sortie diminue au fil des max poolings. C'est voulu pour intégrer peu à peu l'information contenue dans les différentes régions. Sa profondeur augmente avec le nombre de filtres, qu'on multiplie arbitrairement dès qu'on le peut pour disposer de plusieurs lectures possibles des mêmes entrées.

→ Les paramètres à apprendre se multiplient donc couche après couche à cause de ce nombre de filtres, même si on maintient les mêmes tailles de kernel.

En arrivant sur les couches FC, on constate un bond conséquent du nombre de paramètres à apprendre : la couche FC4 à l'interface des convolutions est la raison majeure.

Mais grâce aux couches FC, l'image initiale devient un vecteur monocanal qui résume tout ce qui précède ; en appliquant un softmax, on obtient simplement une distribution de probabilité sur les classes. Au prix de ce coût calculatoire, on parvient donc à condenser parfaitement l'information pour faire la classification.

11

Quel est donc le nombre total de poids à apprendre ? Comparer cela au nombre d'exemples.

En sommant les poids à apprendre pour toutes les couches, on obtient un total de 1 190 000. Même si la convolution aide déjà beaucoup par rapport à un réseau fully-connected (cf. Q1), ce nombre reste très supérieur au nombre d'exemples disponibles pour l'entraînement dans la base CIFAR (50 000). La relation est certes plus complexe, mais il y a un risque intuitif de surapprentissage (cf. résultats Q18).

12

Comparer le nombre de paramètres à apprendre avec celui de l'approche BoW et SVM.

Le Bag of Words du TP1 contenait mille mots-image (SIFTs). Une SVM devait catégoriser chaque image en fonction de la fréquence des mots dedans. Elle devait donc donner un poids à chacun – ce qui fait mille paramètres, et c'est environ mille fois moins que pour ce réseau.

L'approche BoW+SVM permettait d'obtenir de bons résultats avec peu de paramètres, mais cela ne la rend pas préférable pour autant. De fait, elle demandait d'articuler plusieurs unités de calcul qu'il fallait *fine-tuner* séparément. Le réseau de neurones, en revanche, fonctionne end-to-end : même si la modulation des hyperparamètres est difficile, elle est faite une fois pour toutes et toutes les opérations sont automatisées à l'intérieur.

De plus, un réseau de neurones fonctionnel peut sélectionner automatiquement des caractéristiques adéquates dans les images d'une base et les utiliser pour sa prédiction. C'est un avantage majeur par rapport à un BoW figé qui aurait été construit à l'avance.

Apprentissage du réseau

On essaie maintenant de faire apprendre le réseau convolutionnel construit à la section précédente. On se rendra compte que l'utiliser tel quel n'est pas suffisant : le risque de surapprentissage relevé à la Q11 est une menace réelle pour la performance.

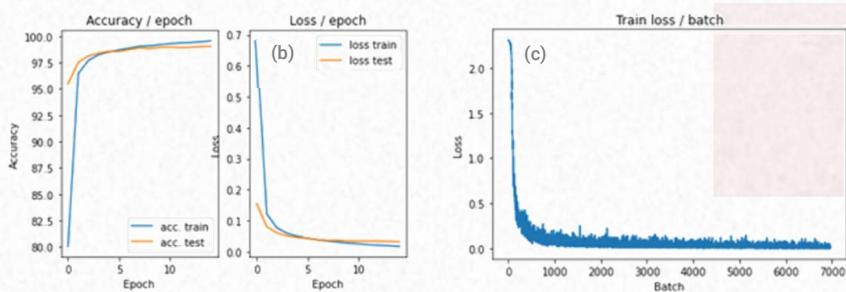
13

Lancez l'entraînement avec le code fourni.

Un premier essai a été réalisé avec un code *dummy* pour la base MNIST (plus petite, et en noir et blanc). L'apprentissage se fait sans problème - au bout de 15 epochs, la loss a convergé vers 0 en train et en test.

(a) moyenne	en train	en test
de la précision	99.56%	99.03%
de la loss	0.0168	0.0318

Figure 2 – Résultats sur MNIST :
(a) statistiques finales, (b) courbes d'apprentissage sur 15 epochs,
(c) surveillance de la loss par batch.



La précision est parfaite en train et en test (tableau (a)), meilleure qu'au TME précédent (cf. Q2). La loss par batch ne fait que diminuer de façon constante pour se rapprocher de 0 (courbe (c)). C'est ce qu'on attend. On constate juste des indices légers de surapprentissage en (b) : après 7 epochs, la loss en test se met à diminuer moins vite que la loss en train. Il aurait fallu arrêter l'entraînement plus tôt pour l'éviter.

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) ?

La loss en train est calculée pendant chaque epoch, au fur et à mesure qu'on rencontre les exemples. En test, elle n'est relevée qu'en fin d'epoch. (La raison de ce choix est que la loss en train est utile, comme elle conditionne la mise à jour des paramètres, là où la loss en test n'est qu'un indicateur d'apprentissage.)

Il explique en fait la grande différence dans l'accuracy train et test à la première epoch, où la loss en test est artificiellement plus élevée ; le modèle progresse si vite qu'il est nettement meilleur en fin d'epoch.

16

Modifiez le code pour utiliser la base CIFAR-10 et implémenter l'architecture demandée. Quels sont les effets du pas d'apprentissage (learning rate η) et de la taille de mini batch ?

Pour le paramétrage, on a déjà longuement détaillé l'impact théorique de η et des diverses variantes de gestion des données (toutes ensemble, ou par minibatch...) au TME précédent. Les courbes ci-dessous montrent un effet en pratique sur la loss, l'accuracy et le temps de traitement par epoch, sur 15 epochs :

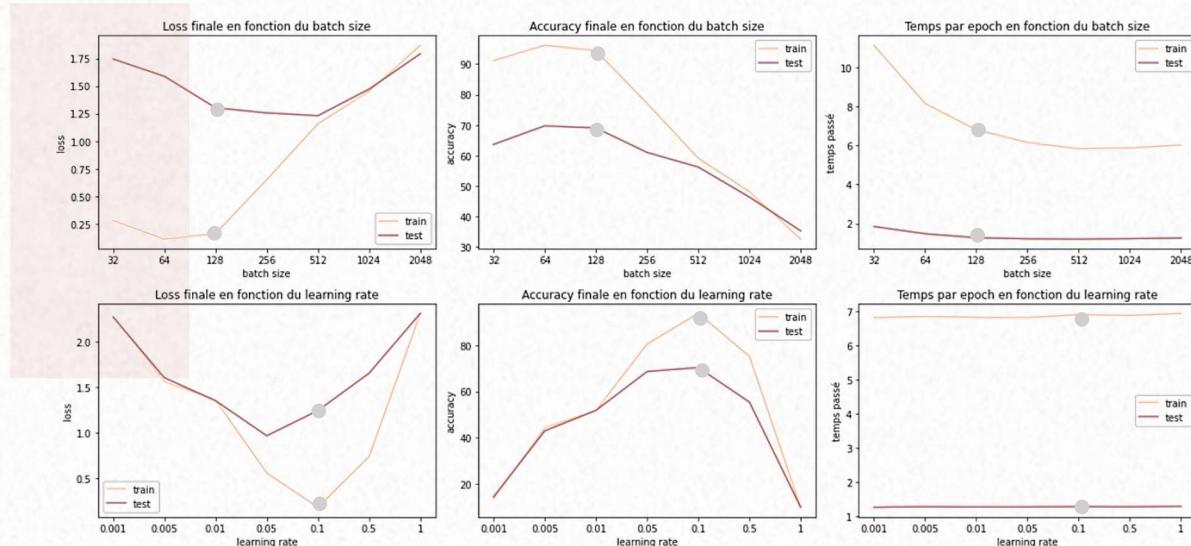


Figure 3 – Résultats MNIST : impact des hyperparamètres sur l'apprentissage. Paramètres du meilleur compromis signifiés par ●

Côté loss, accuracy : learning rate et batch size impactent la bonne marche de l'apprentissage.

→ Avec le même nombre d'epochs, un petit pas d'apprentissage rend le processus trop lent pour converger. S'il est trop grand, il entraînera une divergence. Les courbes montrent en effet qu'il faut un juste milieu.

→ Quant à la taille des batchs, le modèle doit pouvoir s'informer rapidement sur les données pour faire assez de mises à jour par itération ; or, diviser les données en grand batchs entraînera peu de mises à jour, donc un apprentissage là aussi trop lent. Les courbes montrent bien qu'il faut favoriser les petits batchs.

Côté temps par epoch : il dépend du nombre de batchs visités, donc du batch size seul. Si les batchs sont petits, de 32 éléments par exemple, il faudra en traiter plus pour passer sur la totalité des données. On favorise donc les batchs un peu plus grands pour éviter cela.

Pour batch size, on arrive donc à des conclusions contradictoires selon que l'on optimise l'accuracy ou le temps passé ; on cherche un équilibre. A chercher la plus petite loss, la plus grande accuracy possible, et un temps de traitement minimal si ce n'est pas incompatible, les courbes montrent qu'un learning rate de 0.1 et un batch size de 128 donnent un bon compromis. C'est ce qu'on utilise dans la suite.

17

A quoi correspond l'erreur au début de la première époque ? Comment s'interprète-t-elle ?

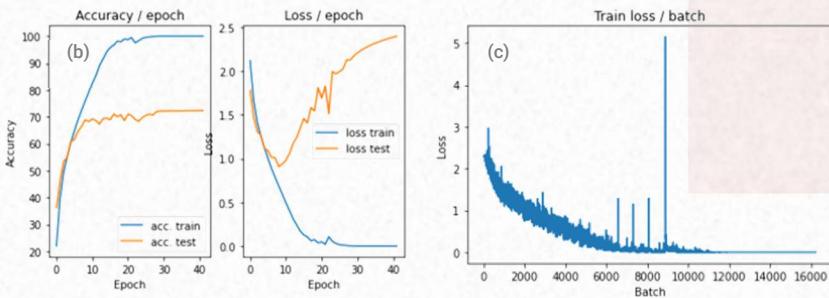
Le modèle découvre les données lors de la première époque, et l'erreur obtenue est systématiquement très grande. De fait, elle ne dépend que de l'initialisation des poids (aléatoire, Xavier, etc.) et n'est pas apprise. La première erreur sert donc de score de référence, et il faut l'améliorer au cours de l'apprentissage en optimisant les paramètres ; si on n'y arrive pas, c'est que le modèle n'est pas adapté au problème.

18

Interpréter les résultats. Qu'est-ce qui ne va pas ? Quel est ce phénomène ?

Comme en Q13, l'apprentissage se fait [fig. 4 (abc) page suivante]. Les courbes d'apprentissage en train (b) montrent bien une loss décroissante accompagnée d'une accuracy croissante. Les deux convergent, on finit sur une accuracy optimale (100%) et une loss nulle, toutes deux maintenues à partir de la 20^{ème} epoch.

	en train	en test
(a) moy. finale de la précision	100.0%	72.32%
de la loss	0.0001	2.4014



Cependant, à partir de la 10^{ème} epoch, le modèle semble être tombé dans un surapprentissage prononcé qui s'accentue avec le temps : la loss en test diverge là où la loss en train continue à diminuer (courbes (b)). Vu l'écart, c'est un problème grave qui donne une accuracy décevante en test : 72% (tableau (a)). Sur le graphique (c), on remarque aussi que certains batchs semblent poser problème au modèle en train. La loss présente des sauts ponctuels à des valeurs très élevées, comme s'il n'était pas adapté aux données précises qu'il vient de rencontrer. (On arrive malgré cela à la loss nulle qui explique la précision de 100%).

Tous ces indices trahissent en fait un manque dans la capacité de généralisation du modèle.

Améliorations des résultats : traitement sur les données

Il y a diverses solutions pour pallier ce manque et réduire le surapprentissage. Le premier ensemble de méthodes s'applique directement aux données passées en entrée.

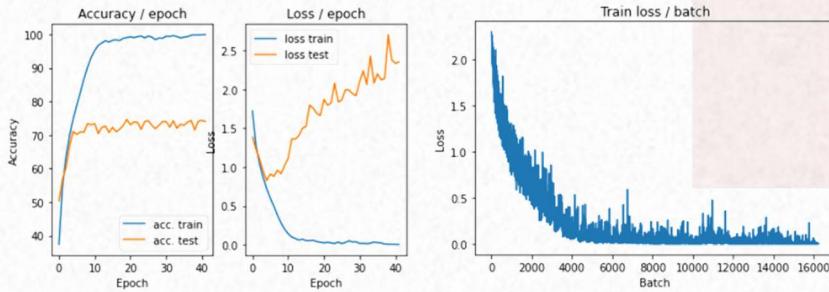
On propose d'abord de normaliser leur distribution (retrancher la moyenne, diviser par l'écart-type).

19

Décrire vos résultats après normalisation des exemples.

moyenne finale	en train	en test
de la précision	99.90%	74.02%
de la loss	0.0034	2.3529

Figure 5 – Résultats CIFAR-10 post-normalisation standard. Même étude qu'avant.



REMARQUES COTE TRAIN

- Par rapport à la Q18, l'accuracy se développe plus vite (sur le graphique (b), elle atteint un plateau au bout de 12 epochs là où il en fallait 20 avant).
- Pourtant, la loss par batch (c) ne s'annule plus comme elle le faisait, ce qui explique la perte de 0.1% d'accuracy finale (a).
- Malgré cette incertitude, la loss par batch (c) ne présente plus de pics majeurs : cela peut vouloir dire que le modèle reconnaît les batchs comme similaires, donc qu'il a compris la forme générale des données.

REMARQUES COTE TEST

- Sur le graphique (b), l'écart du surapprentissage demeure et il se déclare plus tôt (epoch 5 vs. 10 en Q18) du fait de la vitesse d'apprentissage accrue.
- L'accuracy finale (a) a un peu augmenté : + 2%.

CONCLUSION – Le modèle arrive plus loin plus vite, mais se trompe toujours en test. Normaliser l'aide à généraliser sur les données ; mais cette aide étant la même pour le train et le test, on ne réduit pas l'écart et on ne traite pas le surapprentissage.

20

Pourquoi ne calculer l'image moyenne que sur les exemples d'apprentissage, et normaliser les exemples de validation avec la même image ?

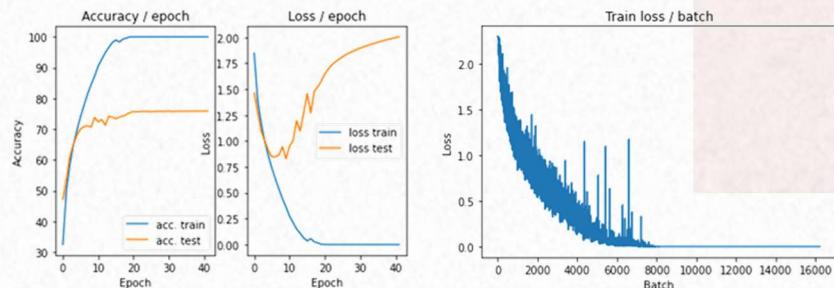
Normaliser les exemples de validation séparément à l'évaluation revient à s'informer sur leur distribution. Cela améliorerait les performances de façon malhonnête. L'évaluation doit se faire dans un contexte

réaliste, le modèle doit découvrir le test à l'aveugle ; il faut donc s'assurer que la normalisation ne soit fondée que sur les exemples d'apprentissage, les seuls qui soient officiellement connus.

2 D'autres schémas de normalisation peuvent être plus efficaces, comme la normalisation ZCA. Les essayer, expliquer les différences et les comparer à celle demandée.

moyenne finale	en train	en test
de la précision	100.0%	75.88%
de la loss	0.0001	2.0050

Figure 6 – Résultats CIFAR-10 post-normalisation [-1,1]. Même étude.



La normalisation [-1,1] suit le même principe qu'avant, les intervalles de valeurs sont juste différents. Les résultats sont comparables avec ceux de la normalisation standard (juste un changement de contraste). D'un autre côté, notre implémentation de ZCA est victime d'un bug récurrent, nous n'avons donc pas de résultats. Sur le principe, elle diffère de la normalisation standard en cela que les données sont centrées, mais pas réduites avec l'écart-type global ; en fait, on fait dépendre le paramètre de réduction d'une PCA (Principal Components Analysis) et le calcul final dépend de la covariance interpixels dans le dataset.

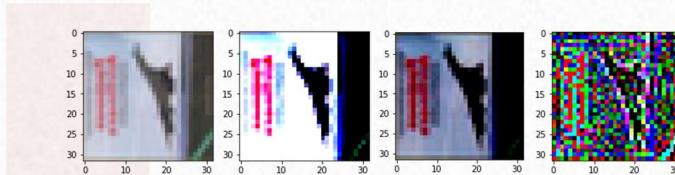


Figure 7 – Visualisation : image originale, puis normalisation standard, [-1,1] et ZCA.

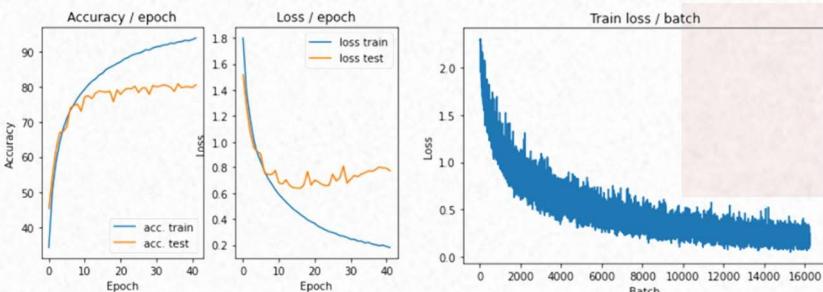
Le mode de normalisation ZCA ne renvoie donc pas une sortie très naturelle par rapport à l'original, mais contient plus d'information sur la distribution des données.

Une approche complémentaire sur les données est la *data augmentation*. On a expliqué en Q11 que le surapprentissage venait de la différence de grandeur entre le nombre de paramètres à optimiser et la taille de la base de *train* ; on cherche donc à augmenter artificiellement cette taille.

22 Décrire vos résultats expérimentaux après data augmentation et les comparer aux précédents.

moyenne finale	en train	en test
de la précision	93.91%	80.68%
de la loss	0.1811	0.7712

Figure 8 – Résultats CIFAR-10 post-normalisation et data augmentation. Même étude.



REMARQUES COTE TRAIN

- Au bout de 42 epochs, le graphique (c) montre que la loss par batch oscille et est encore loin de s'annuler ; cela conduit à une perte d'accuracy, le score final relevé sur (a) est d'environ 94%.
- La courbe d'apprentissage en (b) n'a pas atteint de plateau, on pourrait donc aller plus loin dans l'apprentissage.

REMARQUES COTE TEST

- Le surapprentissage se déclare à l'epoch 7 (b) mais est moins grave qu'en Q19 (écart plus faible).
- L'accuracy finale en test dépasse 80% (a), soit déjà 10% de plus qu'en Q18.
- Mais contrairement au cas du train, l'allure des courbes d'apprentissage (b) (stagnation) montre qu'on ne ferait pas mieux avec plus d'itérations.

CONCLUSION – Les observations en train montrent qu'on perd en vitesse d'apprentissage par rapport aux tentatives précédentes. La data augmentation pose des difficultés au modèle qui n'a pas eu le temps de converger alors qu'on a autant d'itérations (et qu'il y en avait auparavant 20 de trop sur 42). Les observations en test montrent que le surapprentissage est réduit, mais reste un facteur bloquant.

23 L'approche par symétrie horizontale vous semble-t-elle utilisable sur tout type d'images ?

Tout dépend de la sémantique des images. Si c'est la forme de l'objet qui est caractéristique (une voiture, un visage) appliquer une symétrie ne fera que redoubler sa présence dans le dataset et sera bénéfique. Typiquement, sur CIFAR-10, il n'y a aucun problème (avions, bateaux, chiens...) : ces objets peuvent être rencontrés dans n'importe quel sens dans le monde réel et sont toujours reconnus de même. Par contre, si l'orientation compte (comme pour une lettre, b, p, d, q, ou un chiffre, 6, 9), la symétrie la perd et il y a confusion entre plusieurs classes. Ce n'est donc pas recommandé sur MNIST par exemple.

24 Quelles limites voyez-vous à ce type de *data augmentation* par transformation du dataset ?

De façon générale, la transformation du dataset doit rester alignée avec la réalité. Ce qui est appris doit pouvoir être transposé au terrain ; or il arrive que les observations soient rencontrées uniquement dans les circonstances de prise de vue initiales.

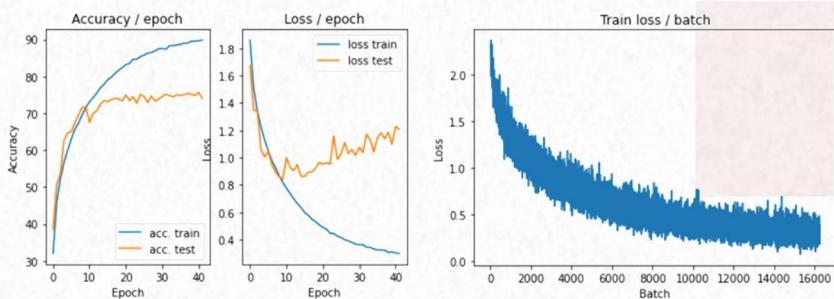
Dans ces cas-là, des images transformées ne seraient pas proches des données réelles et le biais appris dessus serait trop fort pour assurer de bonnes performances en test/en contexte. Si l'on veut plus de données, on ne peut alors que compléter le dataset avec des images originales.

25 Bonus : D'autres méthodes de *data augmentation* sont possibles. Chercher lesquelles.

Bien d'autres méthodes sont possibles. Nous avons essayé d'ajouter un flou gaussien et un changement de perspective et entraîné séparément un modèle qui les utilise, pour voir si cela avait de l'impact.

moyenne finale	en train	en test
de la précision	89.79%	74.00%
de la loss	0.2986	1.2059

Figure 9 – Résultats CIFAR-10 avec normalisation et une autre data augmentation. Même étude.



La data augmentation par symétrie et cropping de la Q22 était plus judicieuse : ici, on n'a rien gagné par rapport à la normalisation (Q19) en termes de score – on a juste ralenti l'entraînement. De fait, les images sont petites, et on dirait que le flou et la variation de perspective déforment la structure des objets de façon trop peu réaliste pour le test (on avait raisonné ainsi pour éliminer le cutout des transformations possibles). Une autre option prometteuse aurait donc été d'étudier des rotations ou d'autres types de cropping.

Idéalement, il faudrait ajouter à la base tous les types d'augmentation dont l'efficacité est prouvée. Mais élargir la base, c'est aussi devoir entraîner le modèle plus longtemps pour avoir de bons scores. Pour abréger les entraînements, nous continuons donc les tests avec les ajouts obligatoires seulement.

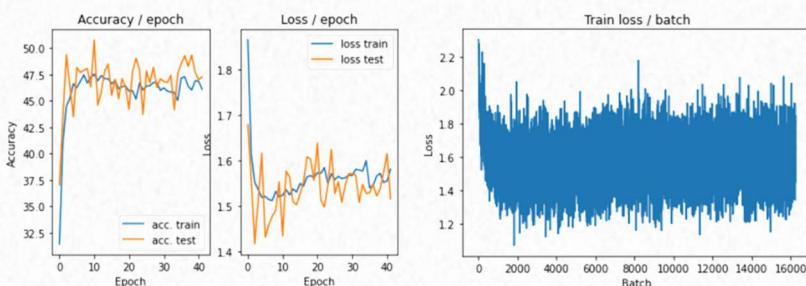
Améliorations des résultats : tuning de l'apprentissage

A cause des cas limite (Q24), et à cause de la lenteur qu'elle entraîne, on ne peut pas tout faire reposer sur l'augmentation des données. On essaie donc aussi d'optimiser l'apprentissage en soi : peut-on l'accélérer et faire en sorte que le modèle s'autopénalise s'il ne va pas dans le bon sens ?

26

Décrivez vos résultats après ajout d'un momentum sur l'optimiseur et d'un learning rate scheduler.

L'ajout du momentum correspond à une stabilisation de l'apprentissage. Lors de chaque mise à jour, la direction du nouveau pas de gradient est combinée avec « l'inertie » qui vient de celle du précédent. Cela accélère l'apprentissage lorsque les pas vont toujours dans la même direction, et cela permet aussi d'avancer plus prudemment si la direction qu'on vient de calculer est incohérente avec l'historique.



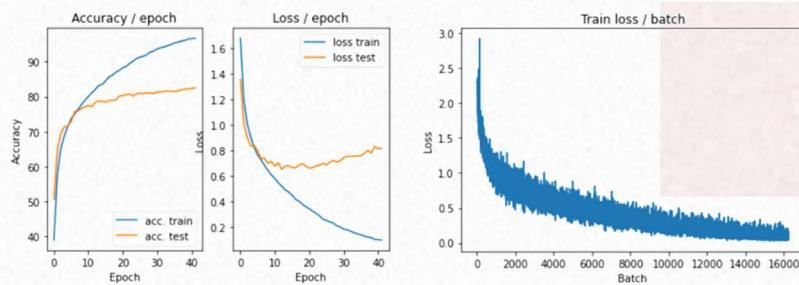
Mais l'ajout du momentum seul n'est pas une bonne idée. Avec le même paramètre η , constant et assez grand (0.1), il n'y a pas de convergence.

Figure 10 – Résultats CIFAR-10 après ajout d'un momentum seul (param 0.9).

De fait, les pas du début de l'apprentissage ne donnent pas forcément une direction compatible avec ceux qui suivent lorsqu'on est proche de l'optimum. Pour éviter qu'ils comptent trop, la solution est d'ajouter un *learning rate scheduler* qui va réduire le pas d'apprentissage au fil des itérations. En combinant les deux (et en abaissant aussi le momentum à 0.75), les résultats sont effectivement meilleurs :

moyenne finale	en train	en test
de la précision	96.75%	82.51%
de la loss	0.0964	0.8182

Figure 11 – Résultats CIFAR-10 avec normalisation, data augmentation, momentum et scheduling.



REMARQUES COTE TRAIN

- Par rapport à la data augmentation seule (Q22), au bout de 42 epochs, on a réussi à monter plus loin en termes d'accuracy : 97% (a). Même si on avait 100% en Q21, 97% est satisfaisant pour un dataset dont la taille a augmenté.
- La courbe de (b) ne montre aucun indice de convergence : il était possible d'aller plus loin.
- La loss par batch (c) oscille toujours, mais moins qu'en Q22 sur la fin de l'apprentissage. C'est un signe de stabilisation.

REMARQUES COTE TEST

- L'accuracy dépasse 80% ((a), +2% par rapport à Q22), et la courbe d'apprentissage (b) est lissée : on a une plus grande stabilité.
- L'écart train-test typique du surapprentissage reste cependant présent à partir de l'epoch 10.

CONCLUSION – La stabilisation de la descente de gradient par *momentum* et *scheduling* a augmenté les scores en train et en test, mais pas réduit leur écart : le surapprentissage demeure.

27

Pourquoi ces deux méthodes améliorent-elles l'apprentissage ?

La fonction de coût a une forme complexe en Deep Learning. En faisant un pas dans la direction du gradient, on peut glisser, dévier de l'orientation idéale et multiplier les oscillations autour du paramétrage optimum. La correction par momentum permet de mieux orienter les pas dans la direction du gradient, de réduire ces oscillations et d'aller tout droit (donc plus vite) vers l'objectif estimé.

Le scheduling se fait en deux temps : il permet d'approximer la région cible le plus vite possible au départ (et d'éviter les minima locaux de la fonction de coût) avec un grand pas d'apprentissage (1) ; puis, pour s'assurer de converger, on réduit ce pas en s'approchant du paramétrage optimal (2). On a donc au final une estimation vraiment fine des paramètres (2) pour un minimum qui a plus de chances d'être global (1).

Notons qu'on a parlé dans les deux cas de rapidité, de justesse et de stabilité de l'apprentissage : mais le surapprentissage, s'il existe, n'est pas affecté. C'est bien ce qu'on a observé pour nos résultats.

28

D'autres variantes de la descente de gradient et d'autres stratégies de planification existent. En tester quelques-unes.

L'optimiseur Adam est un choix par défaut en DL. Il ajoute un momentum sur l'algorithme de descente RMSprop, qui, lui, *normalise* le gradient en le comparant aux précédents avant de faire un pas.

RMSprop (Root Mean Square Propagation) réduit le pas qu'il va faire si le gradient lui semble trop grand par rapport aux précédents, et l'augmente sinon. On n'a donc plus besoin de *scheduling* car la valeur du learning rate devient adaptative.

Adam combine donc une optimisation sur l'orientation et sur l'amplitude des pas de gradient.

Les résultats avec Adam sont bons, mais il a fallu réduire le learning rate (à 0.005) pour éviter de diverger.

moyenne finale	en train	en test
de la précision	95.03%	82.32%
de la loss	0.1401	0.7505

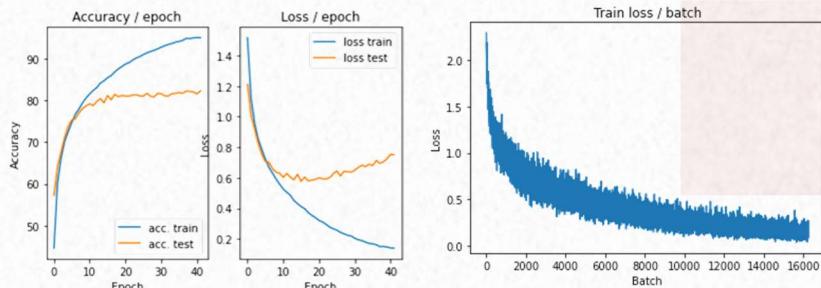


Figure 12 – Résultats CIFAR-10 avec normalisation, data augmentation et optimisation ADAM.

Le modèle obtient vite de bons résultats, mais sans amélioration du point de vue du surapprentissage. Celui-ci se déclare d'ailleurs très tôt (epoch 5) du fait de la vitesse de progression.

Améliorations des résultats : corrections sur le réseau

Pour finir, on essaie de modifier le réseau en gardant son architecture : on transforme les couches, comme avec un Dropout qui désactive aléatoirement quelques neurones à chaque epoch. On peut donc agir sur le défaut du surapprentissage du côté des paramètres et sans toucher aux données.

29 Décrire vos résultats expérimentaux après l'ajout d'une couche de dropout après FC4.

La couche FC4 est ciblée en raison du très grand nombre de neurones qu'elle contient.

moyenne finale	en train	en test
de la précision	89.03%	83.95%
de la loss	0.3119	0.5308

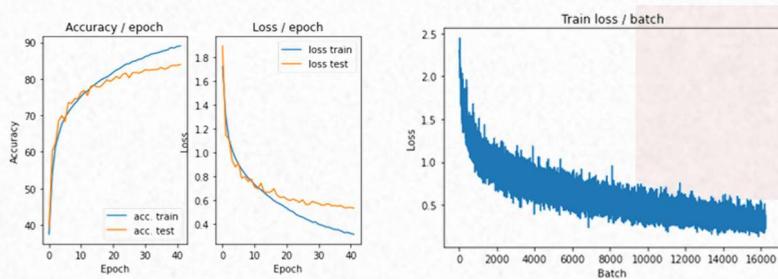


Figure 13 – Résultats CIFAR-10 avec normalisation, data augmentation, momentum, scheduling et Dropout.

Grâce au Dropout, le surapprentissage se déclare plus tard (epoch 15). L'accuracy finale en test a augmenté en conséquence. Par contre, l'accuracy en train a perdu par rapport à Q26, il y a sous-apprentissage (cf. Q32). Il est confirmé par l'allure de la courbe (c) : pas de convergence en vue.

Note : ici, l'hyperparamètre p vaut 0.5. On a essayé des p plus élevés qui réduisent davantage l'écart du surapprentissage mais restreignent beaucoup trop l'accuracy en test. Voir Notebook pour un exemple.

30 Qu'est-ce que la régularisation de manière générale ?

La régularisation permet de pénaliser les modèles complexes (enclins au surapprentissage), souvent en silenciant certains paramètres. En faisant cela, on espère apprendre des fonctions plus générales qui dépendent de moins de variables – donc en contexte de classification, des frontières de décision plus lisses.

31

Discutez des possibles interprétations de l'effet du dropout sur le comportement d'un réseau.

En désactivant des neurones *differents* à chaque epoch, la couche dropout empêche leurs successeurs de compter sur eux ; elle les force donc à fonder leurs calculs sur plusieurs parties de l'information reçue pour pouvoir décider sans si l'une des sources venait à manquer.

Comme chaque neurone dans une couche sert à détecter à une *feature*, cela force la couche suivante à être toujours réceptive à plusieurs de ces *features* et à ne pas se concentrer sur une seule. Cette flexibilité réduira le surapprentissage, car l'on pourra toujours reconnaître une image sur laquelle des features habituellement dominantes ne sont plus applicables (par exemple en cas de rotation ou de bruitage).

32

Quelle est l'influence de l'hyperparamètre de cette couche ?

L'hyperparamètre p correspond à la probabilité de désactivation de chaque neurone. Pour la totalité du réseau, il représente le pourcentage de neurones effectivement silencieux.

Un petit p fait peu contre le surapprentissage, car on ne change pas grand-chose à la structure du réseau. En revanche, si p est trop grand, le surapprentissage bascule dans l'excès inverse : on sous-apprend, car il n'y a pas assez de *features* retenues dans le réseau à chaque epoch. De plus, le réseau sera forcé de donner des poids très grands à celles qui restent, ce qui est source d'instabilité.

Il faudra trouver un équilibre, souvent pour $p \in [0.5, 0.8]$.

33

Quelle est la différence de comportement de la couche Dropout entre apprentissage et évaluation ?

La désactivation *Dropout* se fait pendant les epochs d'apprentissage ; mais en test, il faut conserver tous les neurones pour en observer l'effet. Les mentions respectives `model.train()` et `model.eval()` de PyTorch permettent de programmer le comportement du réseau en conséquence, selon la phase dans laquelle on est – soit supprimer aléatoirement des neurones, soit tous les utiliser.

Notons qu'en test, si l'on utilise les contributions de tous les neurones, on risque d'obtenir des valeurs de sortie très grandes par rapport à celles qu'on avait en train. En test, `model.eval()` remplace donc la couche *Dropout* par un ajustement des valeurs : elle multiplie les sorties des couches qui étaient élaguées en train par la probabilité de suppression p pour normaliser leur intensité.

34

L'apprentissage peut aussi être amélioré par une batch normalisation, qui est appliquée à la sortie de chaque couche pour chaque batch traité. Grâce à elle, les sorties des couches sont centrées réduites, leur distribution ressemble à celle de la toute première entrée. Les couches qui reçoivent ces valeurs peuvent donc agir comme si elles étaient situées au début du réseau et cela aide à généraliser.

Ajouter une batch normalisation et commenter les résultats.

On a conservé le même paramètre de `nn.Dropout` ($p=0.5$), mais *BatchNorm2d* accélère l'apprentissage en train par rapport à la Q29 ; par conséquent, le surapprentissage repart.

moyenne finale	en train	en test
de la précision	92.96%	85.47%
de la loss	0.1993	0.5042

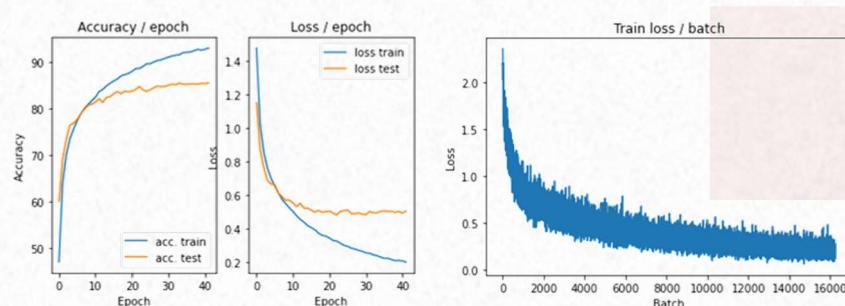


Figure 14 – Résultats CIFAR-10 avec tous les ajouts précédents + batch normalization.

L'effet de la batch norm est comparable à celui de la normalisation : une accélération de l'apprentissage (accuracy de 70% à 10 epochs avec Dropout seul, passage à 80% ici). La convergence n'est cependant pas finie. Malgré l'écart du surapprentissage, qui est un problème, on constate que le score en test est amélioré : ce sont nos meilleurs résultats depuis le début de ce TME.

Conclusions et résumé des méthodes utilisées

On a entraîné un réseau convolutionnel sur la base CIFAR-10, essayé d'optimiser ses performances, et tenté de pallier un problème récurrent de surapprentissage avec diverses méthodes :

DONNEES

- **Normalisation standard** : centrer et réduire toutes les données en partant de la distribution des données de train aide le modèle à généraliser en uniformisant l'apparence des images, et cela accélère la descente de gradient sans réduire le surapprentissage.
- **Normalisation [-1,1]** : même principe.
- **Normalisation ZCA** : il s'agit là aussi de centrer et réduire toutes les données, mais avec une condition supplémentaire sur la réduction qui dépend d'une PCA ; l'information dans la sortie est mieux ciblée, la généralisation est donc meilleure et accélère encore plus l'apprentissage.
- **Data augmentation par cropping et symétrie** : multiplier les données à l'aide de transformations légères permet au modèle d'être confronté à plus de cas, et même si l'apprentissage est ralenti à cause de la masse du dataset, le surapprentissage est réduit.
- **Data augmentation par floutage et changement de perspective** : même principe théorique, mais cette version change la structure des objets et ce n'était pas une bonne idée (pas ou peu d'impact).

DESCENTE DE GRADIENT

- **Optimisation de l'apprentissage par prise en compte du momentum** : la direction du pas de gradient est combinée avec les précédentes pour garder une cohérence – on a moins d'oscillations, l'apprentissage est donc stabilisé et par conséquent accéléré. Pas d'impact sur le surapprentissage.
- **Optimisation de l'apprentissage par scheduling du learning rate** : le pas de gradient est réduit au cours des itérations, on gagne donc en vitesse quand il est grand au début de l'apprentissage et en précision sur la fin, quand il est plus petit : le paramétrage appris est donc plus juste.
- **Optimisation ADAM** : prise en compte du momentum + normalisation du learning rate par rapport à l'amplitude des pas de gradient précédents, l'apprentissage est à la fois plus stable et plus fin.

RESEAU

- **Ajout d'un Dropout** : dans une couche ciblée, inactivation aléatoire de certains neurones à chaque epoch de train, de sorte à forcer l'apprentissage de tous au bout de quelques epochs ; permet de réduire le surapprentissage en évitant de fonder le modèle appris sur quelques features, ce qui le rendrait vulnérable aux changements de contexte.
- **Ajout d'une batch normalization** : renormalisation de la sortie de chaque couche, de sorte que la suivante puisse traiter son entrée comme si elle était au tout début du réseau ; permet une meilleure généralisation et donc une accélération de l'apprentissage.

Chacune de ces méthodes a demandé un paramétrage précis. Celles en noir ont été ajoutées les unes à la suite des autres. On aurait pu les combiner différemment et obtenir potentiellement des résultats encore meilleurs ; mais le nombre de combinaisons possibles est immense et le bon paramétrage difficile à trouver. Cela nous a donné un avant-goût du très long travail de fine-tuning d'un modèle.

Quelques sources :

https://medium.com/@penanklihicyrille_16953/introduction-aux-r%C3%A9seaux-de-neurones-de-convolution-b5da87a5321f
<https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c>
<https://distill.pub/2017/momentum/>
https://jermwatt.github.io/control-notes/posts/zca_sphering/ZCA_Sphering.html