SCIENCES SORBONNE UNIVERSITÉ

Master Données, Apprentissage et Connaissances-DAC

RAPPORT TME 3-4-5-6 RDFIA

Sujet : Introduction aux réseaux de neurones

Realisé par :

Liticia TOUZARI
Hanane DJEDDAL

Table des matières

1 TME 3-4 : Introduction aux réseaux de neurones **2**
1.1 Introduction . . . 2
1.2 Partie 1 – Formalisation mathématique . . . 2
1.2.1 Jeu de données . . . 2
1.2.2 Architecture du réseau (phase forward) . . . 2
1.2.3 Fonction de coût . . . 3
1.2.4 Méthode d'apprentissage . . . 3
1.3 Partie 2 – Implémention . . . 5
1.3.1 Forward et backward manuels . . . 5
1.3.2 Simplification du backward avec torch.autograd . . . 6
1.3.3 Simplification du forward avec les couches torch.nn . . . 6
1.3.4 Simplification de SGD avec torch.optim . . . 7
1.3.5 Application à MNIST . . . 7
1.3.6 SVM . . . 8

2 TME 5-6 : Réseaux convolutionnels pour l'image **8**
2.1 Introduction . . . 8
2.2 Partie 1 – Introduction aux réseaux convolutionnels . . . 9
2.3 Partie 2 – Apprentissage from scratch du modèle . . . 10
2.3.1 Architecture du réseau . . . 10
2.3.2 Apprentissage du réseau . . . 11
2.4 Partie 3 – Amélioration des résultats . . . 13
2.4.1 Normalisation des exemples . . . 13
2.4.2 Data augmentation . . . 14
2.4.3 Variantes sur l'algorithme d'optimisation . . . 15
2.4.4 Régularisation du réseau par Dropout . . . 16
2.4.5 Utilisation de batch normalization . . . 17

1 TME 3-4 : Introduction aux réseaux de neurones

1.1 Introduction

Dans ce tp on met en place un réseau de neurones simple en commençant par étudier d'un point de vue mathématique un perceptron à une couche cachée et sa procédure d'apprentissage. On implémente ensuite ce réseau avec la librairie PyTorch pour le tester sur un problème jouet, puis sur le jeu de données MNIST.

1.2 Partie 1 – Formalisation mathématique

1.2.1 Jeu de données

Réponse 1
L'interêt des ensembles d'apprentissage, test et validation :
Ensemble d'apprentissage : sous-ensemble destiné à l'apprentissage d'un modèle.
Ensemble de test : sous-ensemble destiné à l'évaluation du modèle.
Afin d'éviter le sur-apprentissage tout en convergeant vers le meilleur modèle possible, il est préférable d'utiliser non pas deux jeux de données seulement mais trois (apprentissage, test et évaluation).
L'ensemble de validation sert à évaluer les résultats de l'ensemble d'apprentissage et l'ensemble de test sert à vérifier l'évaluation après que le modèle a passé l'étape de validation.

Réponse 2
L'influence du nombre N d'exemples : Plus le nombre d'exemples augmente, plus on apprend mieux ; cependant augmenter N seulement ne suffit pas, il faudrait que les d'exemples soient répatis d'une façon homogénes entre les classes afin d'éviter que les données soient biaisées.

1.2.2 Architecture du réseau (phase forward)

Réponse 3
Il est important d'ajouter des fonctions d'activation entre des transformations linéaires puisque une succession de fonctions linéaires induit une fonction linéaire et le modèle reste simpliste, il est donc nécéssaire d'utiliser des fonctions d'activations non linéaires pour avoir des modèles plus compléxes pour un jeu de données non séparables linéairement.

Réponse 4
$n_x = 2$: représente la taille des données.
$n_h = 4$: la dimension de la couche cachée du modèle d'apprentissage, dépent des paramétres d'apprentissage. Quand h diminue on a un modèle peu complexe, risque de sous-apprentissage et quand h augmente on risque de sur-apprendre.
$n_y = 2$: représente le nombre de classes en sortie du modèle.

Réponse 5
$\mathbf{y}$: représente les classes réelles.
$\hat{\mathbf{y}}$: représente les classes prédites.

Réponse 6
On utilise une fonction SoftMax en sortie afin d'avoir un vecteur assimilable à une distribution de probabilité.

Réponse 7
Les équations mathématiques permettant d'effectuer la passe forward du réseau de neurones sont les suivantes :

— $\tilde{\mathbf{h}} = W_h x + b_h$
— $\mathbf{h} = tanh(\tilde{h})$
— $\tilde{\mathbf{y}} = W_y h + b_y$
— $\mathbf{y} = softmax(\tilde{y})$

1.2.3 Fonction de coût

Réponse 8
Pendant l'apprentissage, on cherche à minimiser la fonction de coût. Pour cela, pour un $y_{i*} = 1$, les $\hat{y}_i$ doivent varier de telle sorte que $\hat{y}_{i*}$ augmente tandis que $\hat{y}_j, j \neq i*$ doivent diminuer.

Réponse 9
L'entropie croiséee(cross-entropy) est plutôt adaptée aux problèmes de classification parce qu'elle minimise la distance entre deux distributions de probabilité - prédite et réelle.
Considérons un classifier qui prédit si l'animal donné est un chien, un chat ou un cheval avec une probabilité associée à chacun.
Supposons que l'image originale soit celle d'un chien et que le modèle prédit 0.2, 0.7, 0.1 comme probabilités pour trois classes. Comme les vraies probabilités ressemble à [1,0,0], ce que nous voulons idéalement, c'est que nos probabilités prédites soient proches de cet original distribution de probabilité. Donc, l'entropie croisée, assure que nous minimisons la différence entre les deux probabilités.

L'erreur quadratique(mean squared error, MSE) est plutôt adaptée aux problèmes de regression puisqu'elle indique à quel point une droite de régression est proche d'un ensemble de points. Pour ce faire, elle prend les distances des points à la droite de régression (ces distances sont les «erreurs») et les met au carré. La quadrature est nécessaire pour éliminer tout signe négatif. Cela donne également plus de poids aux différences plus importantes.

1.2.4 Méthode d'apprentissage

Réponse 10
Descente de gradien Batch : toutes les données d'entraînement sont prises en compte pour effectuer une seule étape. Nous prenons la moyenne des gradients de tous les exemples d'entraînement, puis utilisons ce gradient moyen pour mettre à jour nos paramètres. Ce n'est donc qu'une étape de la descente de gradient en une seule époque. Cette approche permet de mieux apprendre mais prend beaucoup de temps et presque impossible pour un jeux de données très grand.
Descente de gradien stochastique online : dans cette version un seule exemple est tiré à la fois pour mettre à jour les paramétres. Cette approche est moins couteuse en temps mais moins bonne en performance puisque elle introduit beaucoup de bruit et si les derniers exemples tirés sont divergents les performances peuvent être très mauvaises.
Descente de gradien Mini-Batch : un compromis entre les deux méthodes précédentes serait de prendre à chaque étape un petit ensemble tiré aléatoirement pour méttre à jour le gradient. Cela permet d'éviter les problémes des deux autres méthodes, c'est donc la meilleures approche.

Réponse 11
Le pas d'apprentissage est un hyperparamètre qui contrôle dans quel point le modèle doit être modifié en réponse à l'erreur estimée à chaque fois que les poids du modèle sont mis à jour. Le choix du taux d'apprentissage est important, car une valeur trop petite peut entraîner un long processus d'apprentissage qui pourrait rester bloqué, tandis qu'une valeur trop grande peut entraîner l'apprentissage d'un ensemble de poids sous-optimal trop rapidement ou un processus d'apprentissage instable.

Réponse 12
Pour calculer le gradient d'une couche d'un réseau de neurones avec la backpropagation, on a besoin du

gradient de la couche suivante ; et donc le temps de calcul du gradient dépend du nombre de couches dans le réseau (le gradient de la 1ère couche doit attendre que les gradients de toutes les autres couches soient calculés). Ce calcul se fait une seule fois, contrairement à la méthode naive qui doit parcourir toutes les couches précédentes et calculer le gradient, pour chaque couche, ce qui entraîne des calculs répétitives. Si n est le nombe de couches du réseau, la complexité de backpropagation est O(n), et celle de la méthode naive est O(n(n-1)(n-2)...)

Réponse 13
Pour permettre la backpropagation, les différentes fonctions qui forment les couches du réseau doivent être dérivables.

Réponse 14
La fonction SoftMax et la loss de cross-entropy sont souvent utilisées ensemble et leur gradient est très simple. Montrons que :

$$-\sum_i y_i log(\hat{y}_i) = -\sum_i y_i \tilde{y}_i + log(\sum_j \exp \tilde{y}_i) \tag{1}$$

Sachant que $\hat{y}_i$= Softmax($\tilde{y}_i$)= $\frac{\exp \tilde{y}_i}{\sum_j \exp \tilde{y}_j}$, on a alors :

$$\begin{aligned} -\sum_i y_i log(\hat{y}_i) &= -\sum_i y_i log(\frac{\exp \tilde{y}_i}{\sum_j \exp \tilde{y}_j}) \\ &= -\sum_i y_i(\tilde{y}_i + log(\frac{1}{\sum_j \exp \tilde{y}_j})) \\ &= -\sum_i y_i(\tilde{y}_i - log(\sum_j \exp \tilde{y}_i)) \\ &= -\sum_i y_i \tilde{y}_i + \sum_i y_i log(\sum_j \exp \tilde{y}_i) \\ &= -\sum_i y_i \tilde{y}_i + log(\sum_j \exp \tilde{y}_i) \end{aligned} \tag{2}$$

Réponse 15
Le gradient de la loss(cross-entropy) par rapport à la sortie $\tilde{y}$:

$$\frac{\partial l}{\partial \tilde{y}_i} = \hat{y}_i - y_i \tag{3}$$

$$\nabla_{\tilde{y}} l = \hat{y} - y \tag{4}$$

Réponse 16
Le gradient de la loss par rapport aux poids de la couche de sortie $\nabla_{W_y} l$:

$$\frac{\partial l}{\partial W_{y,ij}} = (\hat{y}_i - y_i) h_j \tag{5}$$

$$\nabla_{W_y} l = \nabla_{\tilde{y}} h^T \tag{6}$$

Le gradient de la loss par rapport aux biais de la couche de sortie $\nabla_{b_y} l$:

$$\frac{\partial l}{\partial b_{y,i}} = \hat{y}_i - y_i \tag{7}$$

$$\nabla_{b_y} l = \nabla_{\tilde{y}} l \tag{8}$$

Réponse 17
Calcule des gradients $\nabla_{\tilde{h}} l$, $\nabla_{W_h} l$, $\nabla_{b_h} l$:

$$\nabla_{\tilde{h}} l = W_y^T \nabla \tilde{y} \odot (1 - h^2) \tag{9}$$

$$\nabla_{W_h} l = \nabla_{\tilde{h}} x^T \tag{10}$$

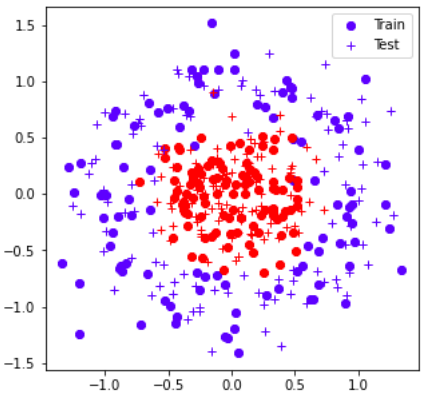
$$\nabla_{b_h} l = \nabla_{\tilde{h}} \tag{11}$$

1.3 Partie 2 – Implémention

Pytorch est une librairie pour l'apprentissage en profondeur qui optimise le calcul en exploitant les GPUs et CPUs. Le but de cette partie est de se familiariser avec cette librairie et de comprendre les différents outils qu'elle offre. On commence dans un premier temps, par une implémentation manuelle de l'architecture d'un réseau de neurones à une couche, qu'on améliore par la suite avec les modules de Pytorch.

L'architecture implémentée consiste de l'application successive d'une transformation linaire, suivi d'une tangente, suivi d'une autre transformation linaire suivi d'un SoftMax.

On applique le réseau de neurones, d'abord sur un jeu de données jouet *Circle* qui consiste des données non sépérable linéairement présentées sous forme d'un cercle (1ere classe) dans un autre cercle (2ieme classe) en 2D (voir figure 1a). Puis sur le jeu de données MNIST qui est un jeu d'images de chiffres manuscrits. voir (figure 1b)



(a) Circle

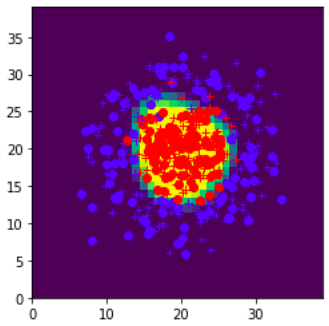
(b) MNIST

FIGURE 1 – Jeux de données

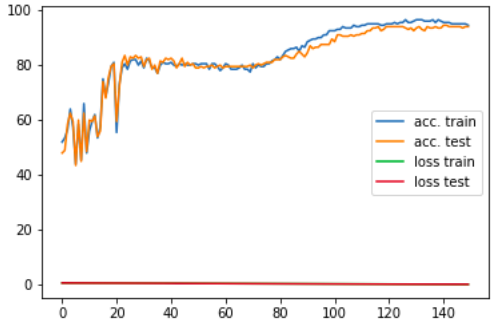
1.3.1 Forward et backward manuels

— **Forward** : Applique les transformations du réseau sur une entrée pour calculer les étapes intermédiaires et la sortie du réseau.
— **Backward** : Calcule les gradients des paramètres et de la loss selon les formules mathématiques trouvées précédemment.

Résultats
Au bout de 150 itérations, le modèle converge avec une accuracy de 94.5% en train et de 94.0% en test. La figure 5a montre la séparation non linéaire des points obtenue à la dernière itération..



(a) Display



(b) Accuracy en fonction de nombre d'itérations

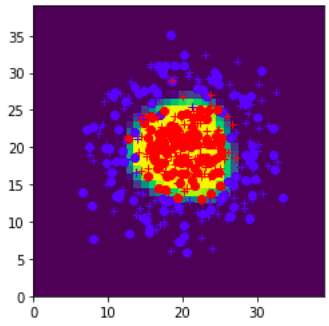
FIGURE 2 – Résultats de la dernière itération de l'algorithme d'apprentissage du réseau de neurones appliqué aux les données Circle, avec l'implémentation manuelle

1.3.2 Simplification du backward avec torch.autograd

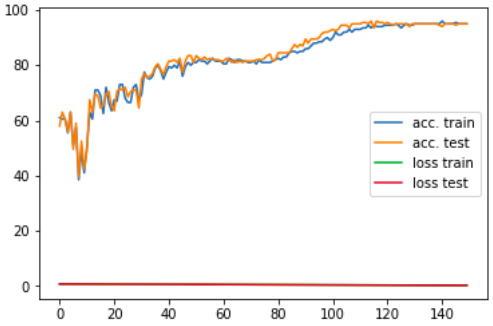
Pytorch offre une fonctionnalité 'torch.autograd' qui permet de calculer automatiquement le gradient des variables (tenseurs) pour lesquelles l'option 'requires_grad' est à True. Avec ce mécanisme, l'implémentation de 'backward' n'est plus nécessaire, il suffit d'applique 'loss.backward()) quand on veux calculer le gradient de la loss.

Résultats

On réexecute l'algorithme d'apprentissage avec cette modification et on obtient les résultats de la figure 3. L'accuracy à la dernière itération est de 95.0% pour le test et le train, qui est similaire à celle trouvée avec l'implémentation manuelle.



(a) Display



(b) Accuracy en fonction de nombre d'itérations

FIGURE 3 – Résultats de la dernière itération de l'algorithme d'apprentissage du réseau de neurones appliqué aux données Circle, avec autograd

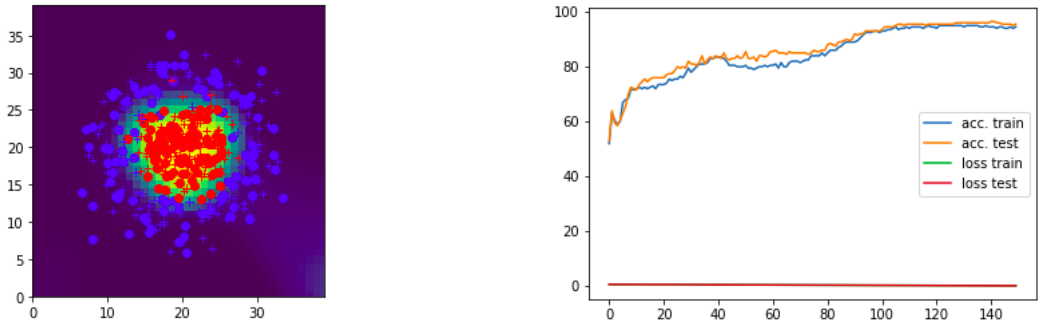
1.3.3 Simplification du forward avec les couches torch.nn

Le paquet nn de Pytorch définit un ensemble de modules qui permet de construire un réseau de neurones.Il contient des fonctions pour représenter les couches du réseau, fonctions d'activation, fonctions de coût,ainsi qu'un ensemble des outils nécessaires pour l'apprentissage.

En utilisant 'torch.nn' pour définir le réseau de neurones, on peut maintenant appliquer le forward directement en faisant appel au modèle, sans devoir implémenter la méthode forward.

Résultats

On obtient des résultats similaires à celles obtenues précédemment comme montre la figure 7 avec une accuracy de 94.5% pour le train et 95.5% pour le test, lors de la dernière itération.



(a) Display (b) Accuracy en fonction de nombre d'itérations

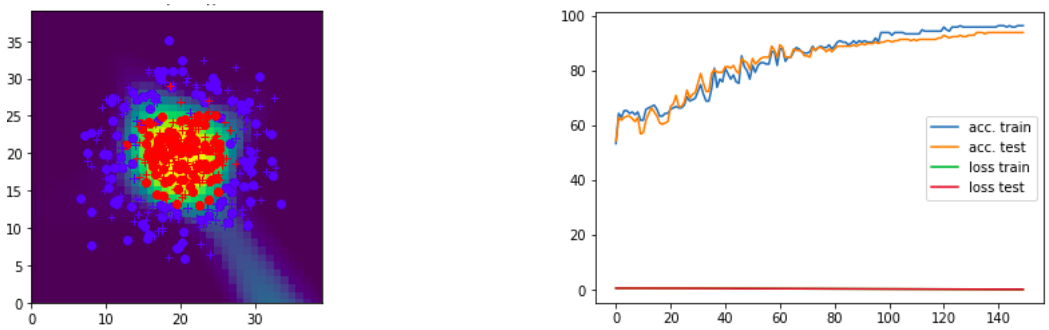
FIGURE 4 – Résultats de la dernière itération de l'algorithme d'apprentissage du réseau de neurones appliqué aux données Circle, avec torch.nn

1.3.4 Simplification de SGD avec torch.optim

Jusqu'à maintenant, on fait toujours la mise à jours des paramètres manuellement lors de l'algorithme d'apprentissage SGD. Torch.optim contient plusieurs optimiseurs, y compris SGD, qui s'occupent de la tâche de la mise à jour des paramètres avec la méthode optim.step().

Résultats

On obtient une accuracy de 96.5% pour le train et 94.0% pour le test à la dernière itération.



(a) Display (b) Accuracy en fonction de nombre d'itérations

FIGURE 5 – Résultats de la dernière itération de l'algorithme d'apprentissage du réseau de neurones appliqué aux données Circle, avec optim

1.3.5 Application à MNIST

On applique le réseau de neurones simplifié avec Module et Optim, au jeu de données MNIST. Le but est de prédire la classe (un chiffre) à partir d'une image manuscrite de ce chiffre.

Résultat

La figure 6 montre l'évolution de l'accuracy en fonction de nombre d'itérations. On remarque que le modèle converge rapidement avec une accuracy de 85.7% pour le train et 86.2% pour le test dès la première itération, et arrive à une accuracy de 93.0% pour le train et 92.9% pour le test à la dernière itération.

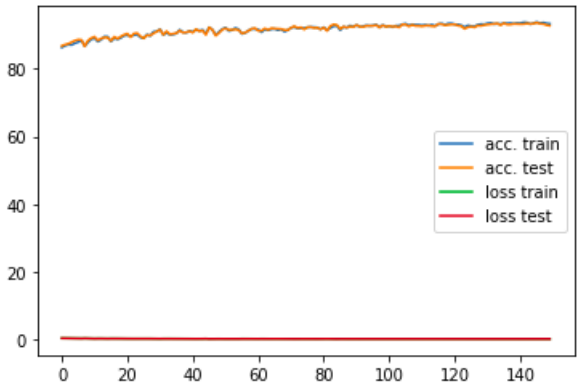


FIGURE 6 – Évolution de l'accuracy en fonction de nombre d'itérations du réseau de neurones appliqué à MNIST

1.3.6 SVM

On essaye, dans cette partie, d'entraîner un SVM sur le jeu de données Circle. On commence par un SVM linéaire qui donne le résultat montré dans la figure 7a avec une accuracy de 53.0%. Cette version de SVM fait une séparation linaire (avec une droite) qui n'est pas adaptée à ce problème et donc sa performance n'est pas forcément meilleure de celle d'un modèle aléatoire.

On utilise un kernel rbf (Radial Basis Function) dans la figure 7b et on obtient une séparation plus adaptée avec une accuracy de 95.0%. Le paramètre de régularisation C n'a pas beaucoup d'impact ici car les données des deux classes sont plus ou moins bien séparés.

(a) SVM Linéaire

(b) SVM avec un noyeau RBF

FIGURE 7 – Application de SVM linéaire (à gauche) et avec un noyeau (à droite) au jeu de données Circle

2 TME 5-6 : Réseaux convolutionnels pour l'image

2.1 Introduction

Dans cette partie, on étudie les réseaux de neurones convolutionnels qui permettent de classifier des images sans devoir effectuer des pré-traitements sur les données d'entrée. On commence par un réseau simple, qu'on développe par la suite pour inclure des techniques permettant d'améliorer le processus d'apprentissage. On applique le modèle sur le jeu de données CIFAR-10.

2.2 Partie 1 – Introduction aux réseaux convolutionnels

Réponse 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 × y × z, la taille de sortie est de :($\frac{x+2p-k}{s}+1$) x ($\frac{y+2p-k}{s}+1$) x z'.
Le nombre de poids à apprendre est de : (kernel_size * kernel_size * channels_in * channels_out) + (channels_out) = k*k*z*z'+z' avec z' le nombre de channels en sortie.
Le nombre de poids qu'il aurait fallu apprendre si une couche fully-connected devait produire une sortie de la même taille correspond à la taille de sortie du fully-connected.

Réponse 2

La raison pour laquelle les couches convolutives sont utilisées est qu'elles permettent d'extraire des caractéristiques significatives en préservant également les relations spatiales au sein de l'image.D'autre part, les fully-connected, bien qu'étant des approximateurs universels, restent médiocres pour identifier et généraliser les pixels bruts de l'image.
Un autre point important est que les couches convolutives résument les caractéristiques de l'image et produisent des cartes de caractéristiques plus concises pour les couches convolutives suivants consécutives sur le pipeline. À cette fin, les couches convolutives permettent de réduire la dimension et de réduire la complexité de calcul. Ceci n'est pas possible pour les fully-connected.
Le troisième point est que les fully-connected imposent des tailles d'image statiques en raison de leurs propriétés héritées, tandis que les couches convolutives nous permettent de travailler sur des images de taille arbitraire, en particulier dans des approches entièrement convolutives.
Cependant, nous utilisons toujours les fully-connected dans la dernière partie des réseaux de classification d'images car leur tâche principale est de fournir des approximations non linéaires universelles dans les tâches de classification / régression.
Une des limitations des convolutions c'est qu'elles manquent d'une compréhension globale des images car elles considère des zones locales.

Réponse 3

L'intérêt du pooling spatial :
Un objet, comme un visage, est toujours un visage s'il est agrandi (donc les traits sont plus éloignés) ou si le visage est tourné ou incliné (les traits comme les yeux, les oreilles, le nez et la bouche, sont compressés horizontalement mais pas verticalement, ou verticalement mais pas horizontalement) ; ou le visage est plus large ou plus étroit que d'habitude.
Un modèle de features spatialement rigide ne reconnaîtra pas les features s'il ne se trouve pas exactement à l'emplacement prévu.
Le pooling spatial rend chaque feature résiliente aux changements de position mineurs, car plusieurs positions sont regroupées. Si cette technique est appliquée de manière répétée et hiérarchique, alors un objet peut être reconnu malgré des distorsions spatiales importantes, ce qui rend la reconnaissance d'objets plus robuste et moins fragile.

Réponse 4

Supposons qu'on essaye de calculer la sortie d'un réseau convolutionnel classique (celui en Figure 2) pour une image d'entrée plus grande que la taille initialement prévue (224×224). On pourra dans ce cas appliquer les couches de convolution sans problèmes puisqu'elles ne dépendent pas de la taille de l'image en entrée (à condition que la padding match bien), cependant arrivé à la couche fully-connected il est nécessaire d'avoir les mêmes dimensions puisque ça dépends de la taille de l'image en entrée, il est donc nécessaire d'avoir une image 224x224 en entrée pour pouvoir appliquer les couches fully-connected.

Réponse 5

On peut voir les couches fully-connected comme des convolutions particulières de taille 1x1.

Réponse 6
Supposons que l'on remplace donc les fully-connected par leur équivalent en convolutions, on aura toujours le même problème de dimensions puisque qu'il faudra avoir la dernière couche de convolution à 1x1x512 mais dans notre cas c'est un 7x7x512 le moyen le plus effectif pour répondre à ce problème est d'utiliser un global pooling qui permet de ne pas dépendre de la taille de l'image.

Réponse 7
La taille des receptive fields des deux premières couches de convolution est de 3 x 3, calculé comme suit : Output shape = [(taille_image+2padding - receptive_field)/stride] + 1.

- Première couche de convolution : 64 channels, kernel 3x3, receptive field 3x3, padding 1 et stride 1. Sortie : 224x224x64.
- Couche pooling : stride 2, kernel 2x2, receptive field 2x2. Sortie : 112x112x64
- Deuxième couche de convolution : 128 channels, kernel 3x3, receptive field 3x3, padding 1 et stride 1. Sortie : 112x112x128.

On garde la même taille pour le receptive field qui est de 3x3.

2.3 Partie 2 – Apprentissage from scratch du modèle

2.3.1 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.

Réponse 8
Pour les convolutions, si on veut conserver en sortie les mêmes dimensions spatiales qu'en entrée dans nôtres cas 32x32. Pour une taille de kernel 5x5, les valeurs de padding et de stride devrait être :
Padding = 2, Stride = 1. Ainsi :
((taille_image+2padding - kernel)/stride) + 1 = (32 + 2x2 - 5)/1 + 1= 32.

Réponse 9
Pour les max poolings, on veut réduire les dimensions spatiales d'un facteur 2. Les valeurs de padding et de stride pour un kernel 2x2 sont les suivantes :
Padding = 0, Stride = 2. Ainsi :
((taille_image+2padding - kernel)/stride) + 1 = (32 + 2x0 - 2)/2 + 1= 16.

Réponse 10
La taille de sortie et le nombre de poids à apprendre pour chaque couche :
— conv1 : 32 convolutions 5×5, suivie de ReLU
sortie : 32x32x32, nombre de poids à apprendre est de 5×5x3x32+32=2432.
— pool1 : max-pooling 2×2
sortie : 16x16x32, nombre de poids à apprendre est de 2×2x32x32+32=4128.
— conv2 : 64 convolutions 5×5, suivie de ReLU
sortie : 16x16x64, nombre de poids à apprendre est de 5×5x32x64+64=51264.
— pool2 : max-pooling 2×2
sortie : 8x8x64, nombre de poids à apprendre est de 2×2x64x64+64=16448.
— conv3 : 64 convolutions 5×5, suivie de ReLU
sortie : 8x8x64, nombre de poids à apprendre est de 5×5x64x64+64=102464.
— pool3 : max-pooling 2×2
sortie : 4x4x64, nombre de poids à apprendre est de 2×2x64x64+64=16448.
— fc4 : fully-connected, 1000 neurones en sortie, suivie de ReLU

sortie : 1x1x1000, nombre de poids à apprendre est de 64x1000+1000=64064.
— fc5 : fully-connected, 10 neurones en sortie, suivie de softmax
sortie : 1x1x10, nombre de poids à apprendre est de 1000x10+10=10010.

Réponse 11
Le nombre total de poids à apprendre est de 267 258, plus petit que le nombre d'exemples qui est de 1 548 954.

Réponse 12
Le nombre de paramètres à apprendre avec l'approche BoW :
Avec l'approche BoW on doit générer un dictionnaire de mots de taille M à partir de la base, et pour chaque patch de taille d de l'image en entrée on génère son codage one-hot de taille nb_patchxM qui sont ensuite normalisé en un seul vecteur de taille M, ce vecteur est le descripteur de l'image. Les paramètres qui doivent être appris sont les mots du dictionnaire de taille M×d.
Le nombre de paramètres à apprendre avec l'approche SVM :
Pour classifier les images avec SVM, il faut commencer par une étape de feature-extraction. Chaque feature est par la suite présentée sous forme d'un histogramme. L'apprentissage s'applique sur l'ensembles des histogrammes et donc le nombre de paramètre à apprendre est f x h où f est le nombre de features et h est la taille de l'histogramme.

2.3.2 Apprentissage du réseau

Réponse 13
On teste le code fourni, en lançant l'entraînement avec les paramètres taille de batch= 128, pas d'apprentissage = 0.1 pour un nombre d'epochs de 100. La figure qui suit montre la fonction de loss et accuracy.

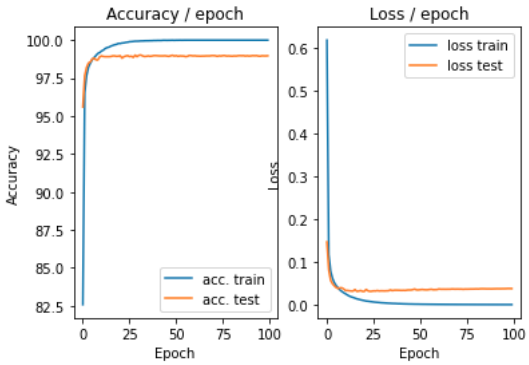


FIGURE 8 – Courbes de Loss et Accuracy pour MNIST

Réponse 14
Dans le code fourni, la loss et l'accuracy en train sont calculées à chaque apprentissage sur un batch tant dis qu'en test elle sont calculées à la fin de chaque epoch seulement, c'est pour ça qu'on remarque qu'au début la loss en test est meilleure que celle en train puisque celle en test est calculé après plusieurs itération sur les batchs (donc le modèle à pu apprendre) alors qu'en train c'est la moyenne des loss/accuracy obtenues après chaque train sur batch (somme des premières valeurs mauvaises avec les dernières valeurs qui sont meilleures).

Réponse 15
On implémente l'architecture demandée pour CIFAR-10. La figure ci-dessous représente le code d'initialisation des couches de notre modèle.

```
def __init__(self):
    super(ConvNetAlex, self).__init__()
    self.features = nn.Sequential(
        nn.Conv2d(3, 32, (5, 5), stride=1, padding=2),
        nn.ReLU(),
        nn.MaxPool2d((2, 2), stride=2, padding=0),
        nn.Conv2d(32, 64, (5, 5), stride=1, padding=2),
        nn.ReLU(),
        nn.MaxPool2d((2, 2), stride=2, padding=0),
        nn.Conv2d(64, 64, (5, 5), stride=1, padding=2),
        nn.ReLU(),
        nn.MaxPool2d((2, 2), stride=2, padding=0),
    )
    self.classifier = nn.Sequential(
        nn.Linear(1024, 1000),
        nn.ReLU(),
        nn.Linear(1000, 10), )
```

FIGURE 9 – Implémentation de l'architecture du modèle pour CIFAR-10

Réponse 16

Les effets du pas d'apprentissage (learning rate) : Quand le pas d'apprentissage est trop petit on converge lentement et en risque de bloquer sur un min local, et s'il est trop grand, il fait de plus grand saut et risque de divergé et de ne jamais atteindre le min global.
Les effets de la taille de mini-batch : Pour une taille de batch trés grande on a moins de bruit, donc un meilleur gradient mais moins de généralisation (over-fitting) de plus le calcul est long.
Et pour une taille petite on a plus de bruit (under-fitting).
Si on augmente la taille du batch on peut augmenter le learning rate.

Réponse 17

L'erreur au début de la première époque correspond à l'erreur d'une classification aléatoire puisque les paramètres ont été initialisé aléatoirement. Puisque la base CIFAR-10 contient 10 classes, on est sensé avoir une erreur au alentours de 1/10=0,1, ce qui est bien le cas.

Réponse 18

La figure ci-dessous représente les courbes de loss est accuracy, on remarque qu'en train la loss diminue puis augmente à partie de l'époque 12, pareil pour l'accuracy qui arrête d'augmenter ne dépassant pas la valeur 0.7. Le phénomène observé est de l'over-fitting, cela se produit qu'on on apprend par-coeur les données en train, on à alors une trés bonne accuracy et loss et train mais de moins bon en test.

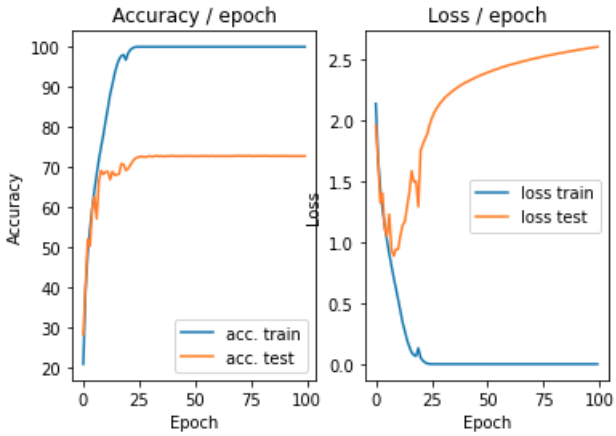


FIGURE 10 – Courbes de loss et accuracy pour CIFAR-10

2.4 Partie 3 – Amélioration des résultats

2.4.1 Normalisation des exemples

La normalisation des données permet de réduire l'écart entre les valeurs extrêmes et d'éviter les overflow dans les calculs car des nombres supérieurs à 1 peuvent rapidement tendre vers l'infini. Elle consiste à centrer et réduire les données. Cela se fait en soustrayant à chaque pixel la valeur moyenne correspondant à son channel et en le divisant par l'écart type correspondant : images =(images - moy)/std. La normalisation rend l'apprentissage des paramètres moins sensible à l'initialisation, et permet d'utiliser un taux d'apprentissage plus grand, ce qui accélère la phase d'apprentissage et améliore la précision.

Réponse 19
La figure 11 montre les courbes de la loss et l'accuracy pour un learning train de 0.1. On observe toujours le phénomène de sur-apprentissage : la loss (resp. accuracy) du train continue à diminuer (resp. augmenter) après 5 époques alors que la loss (resp. accuracy) du test augmente (resp. diminue). Par contre, le modèle converge plus rapidement avec la normalisation (au bout de 5 époques tandis que sans normalisation, il converge après 20 époques), et on attend une accuracy de 76.15 % en test.

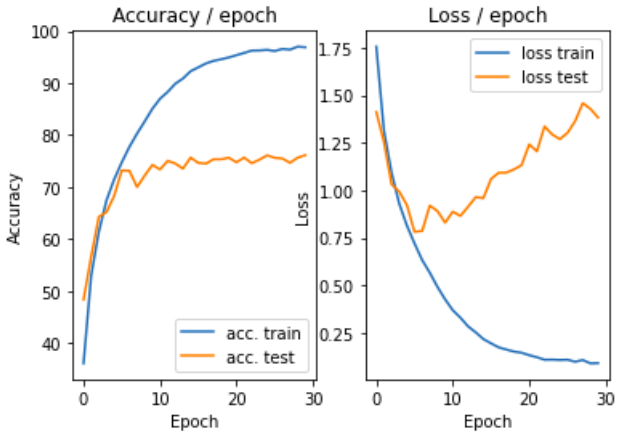


FIGURE 11 – Courbes de loss et accuracy avec normalisation

Réponse 20
On n'utilise pas la même image moyenne calculée sur les données d'apprentissage pour normaliser les exemples de validation car cela peut introduire un biais, c-à-d une information concernant les données d'apprentissage est introduite aux données de test/validation (data leakage), et cela peut influencer l'estimation de score.

2.4.2 Data augmentation

Data augmentation consiste à augmenter artificiellement la diversité et le nombre d'exemples d'apprentissage en appliquant des transformations aléatoires aux images existantes pour créer un ensemble de nouvelles variantes. Elle est est particulièrement utile lorsque l'ensemble de données d'apprentissage d'origine est relativement petit et elle permet de prévenir le sur-apprentissage lors de la construction d'un réseau de neurones convolutif.

On s'intéresse ici à deux transformation : un Crop aléatoire et une symétrie horizontale aléatoire.

Réponse 22
On applique les deux transformations à l'ensemble de train et on fait un crop centré sur les images de test. La figure 12 montre les courbes de la loss et l'accuracy obtenues. On remarque une amélioration des résultats du test avec une accuracy de 80.16 % et une loss inférieure à 1. Le modèle prend un plus du temps pour converger (15 époques) dû à l'augmentation de la taille du dataset.

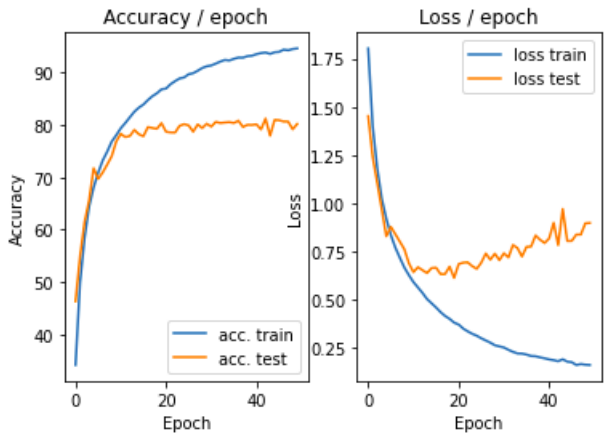


FIGURE 12 – Courbes de loss et accuracy en rajoutant Data augmentation

Réponse 23
La symétrie horizontale n'est pas utilisable sur tout types d'images. On peut l'appliquer seulement quand la symétrie des images est raisonnable et ne change pas la classe d'image. Cette transformation n'est pas applicable par exemple dans le cas des données textuelles (une transformation de la lettre 'd' change la classe à 'b').

Réponse 24
L'augmentation des données par transformation du dataset n'est pas sans limitations : pour une base de données de grande taille, ces transformations peuvent coûter cher en terme de temps d'exécution. Aussi, ce type de transformations peut être appliquer seulement aux données labellisées car elles peuvent introduire de nouveaux patterns dans le cas d'un apprentissage non-supervisé.

Réponse 25
D'autres méthodes du data augmentation existent tels que : Rotation, ColorJitter (change la luminosité,

contraste et saturation des images), Gaussian Blur, Redimensionner les images etc.
La figure 14 montre les courbes de la loss et l'accuracy du modèle en appliquant les transformations suivantes : Crop aléatoire, Rotation de 20° et un colorJitter. On obtient une accuracy de 76.98% en test qui est moins bonne que la séquence de transformations initiale et l'exécution prend un peu plus du temps.

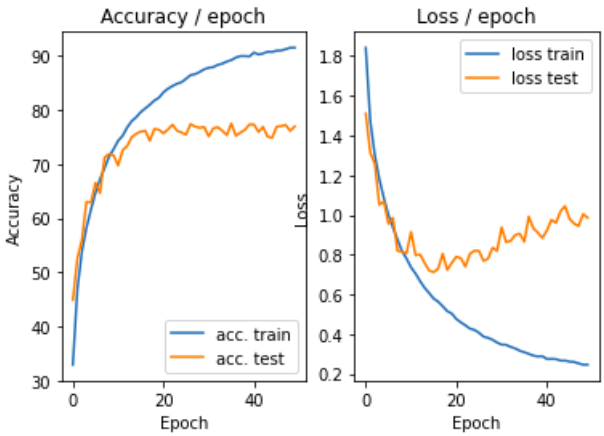


FIGURE 13 – Courbes de loss et accuracy en rajoutant Data augmentation

2.4.3 Variantes sur l'algorithme d'optimisation

On peut améliorer les performances en modifiant la méthode de mise à jour des paramètres. Une optimisation efficace permet de converger rapidement sans trop osciller. On teste ici le Momentum et un learning rate scheduler.

Réponse 26
La figure 12 montre les courbes de la loss et l'accuracy obtenues avec le momentum et learning rate scheduler. On remarque que les courbes sont plus lisses avec des petites variations entre les époques. On obtient une accuracy de 79.53% en test.

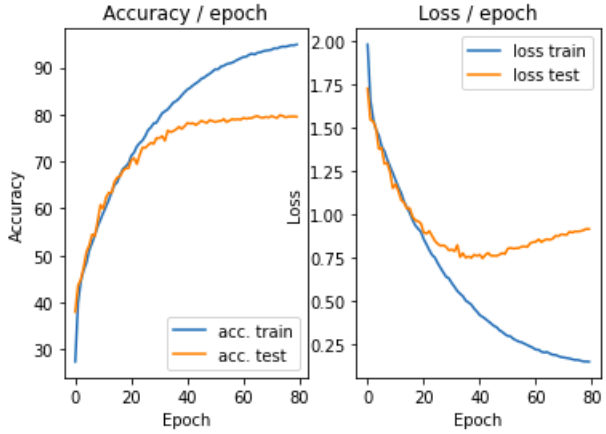


FIGURE 14 – Courbes de loss et accuracy avec optimisation

Réponse 27
Le momentum est une méthode d'optimisation des paramètres qui prend en considération les gradients précédemment calculés, ce qui permet d'évoluer dans la bonne direction et donc converger rapidement sans trop

osciller.
Learning rate scheduler permet d'ajuster le learning rate dynamiquement au cours d'apprentissage. En diminuant le learning rate quand le nombre d'époques augmente, le modèle sera capable de détecter des parties plus profondes et plus étroites de la fonction de la loss (minima locaux) et se stabiliser plus facilement.

Réponse 28
D'autres variantes de SGD existent, tels que : Adagrad (détermine automatiquement un taux d'apprentissage pour chaque paramètre), Adam (garde trace de decaying average du gradient, et gradient au carrée), AdaMax, Nadam etc. Pour le learning rate scheduler, il existe plusieurs stratégies : Time-Based Decay (diminue le learning rate selon le nombre d'itérations), Step Decay (diminue le learning rate à chaque époque par un facteur donné) et Exponential Decay (utilisée précedemenet.)
La figure 15 montre les courbes de la loss et l'accuracy obtenues avec un learning rate scheduler Step Decay avec un step size=10. Les coubres sont moins lisses et l'accuracy ne dépasse pas 55%.

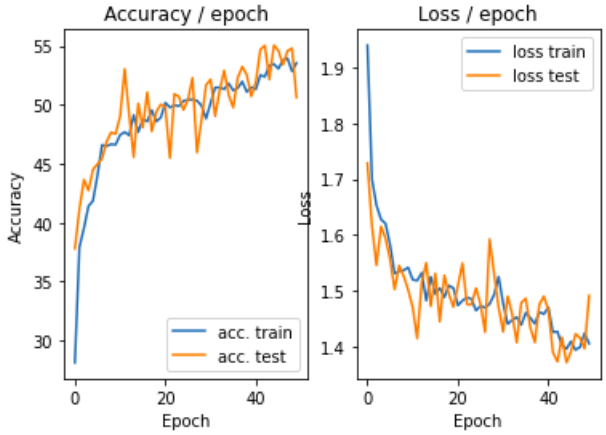


FIGURE 15 – Courbes de loss et accuracy avec Step Decay scheduler

2.4.4 Régularisation du réseau par Dropout

Avoir un très grand nombre de poids à apprendre dans un réseau peut conduire à un sur-apprentissage, la régularisation par Dropout supprime aléatoirement des unités du réseau de neurones lors d'une étape d'apprentissage pour diminuer artificiellement le nombre de paramètres.

Réponse 29
La figure 16 montre les courbes de la loss et l'accuracy obtenues avec la couche Dropout. On remarque une nette amélioration des résultats du test et le problème de sur-apprentissage est quasiment résolu. L'accuracy du test est 80.3%. Par contre, la convergence du modèle est plus lente (70 époques)

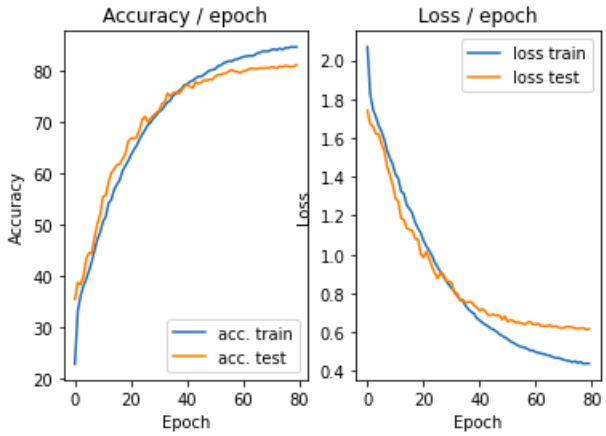


FIGURE 16 – Courbes de loss et accuracy en rajoutant Dropout

Réponse 30
La régularisation en général sert à diminuer la complexité d'un modèle pour résoudre le problème de sur-apprentissage. Elle consiste à ajouter de l'information à un problème. Une méthode généralement utilisée est de pénaliser les valeurs extrêmes des paramètres, qui correspondent souvent à un sur-apprentissage, en utilisant une norme sur ces paramètres, qui est ajouter à la fonction qu'on cherche à minimiser.

Réponse 31
Le dropout est une technique de régularisation qui permet de prévenir le sur-apprentissage. Un réseau de neurones profonds contient plusieurs couches cachées non linéaires, ce qui en fait un modèle très expressif qui peut apprendre des relations très compliquées entre les entrées et les sorties. Souvent, une partie de ces relations n'est qu'un bruit existant dans le training set et non pas dans les données réelles, ce qui conduit à un sur-apprentissage. Avec le Dropout, certaines unités du réseau seront déactivées lors une phase d'apprentissage et réactivées durant une autre. Cela crée plusieurs sous modèles et donc l'apprentissage du modèle principale revient à entraîner plusieurs sous réseaux de neurones. Cette approche peut être coûteuse en temps d'exécution mais elle garantie une bonne généralisation.

Réponse 32
L' hyper-paramètre de la couche Dropout qui est la probabilité d'activation des unités, détermine le nombre des unités actives à chaque phase et donc le nombre de sous-réseaux à entraîner. Une forte probabilité d'activation implique un petit nombre de sous-réseaux est donc une convergence rapide mais pas forcément des bonnes performances. De même une faible probabilité d'activation implique un grand nombre de sous-réseaux et donc une longue durée d'exécution mais une meilleure généralisation.

Réponse 33
Durant l'apprentissage, une unité est présente avec une probabilité p. Durant l'évaluation, une unité est toujours présente mais ses poids sont multipliés par p.

2.4.5 Utilisation de batch normalization

Batch normalization est une technique conçue pour normaliser automatiquement les entrées d'une couche dans un réseau neuronal d'apprentissage en profondeur. Elle a pour effet d'accélérer considérablement le processus d'apprentissage d'un réseau de neurones et, dans certains cas, améliore les performances du modèle via un effet de régularisation.

Réponse 34

La figure 17 montre les courbes de la loss et l'accuracy obtenues avec le batch normalisation. On obtient une accuracy de 85.03% en test et le modèle converge rapidement (après 15 époques).

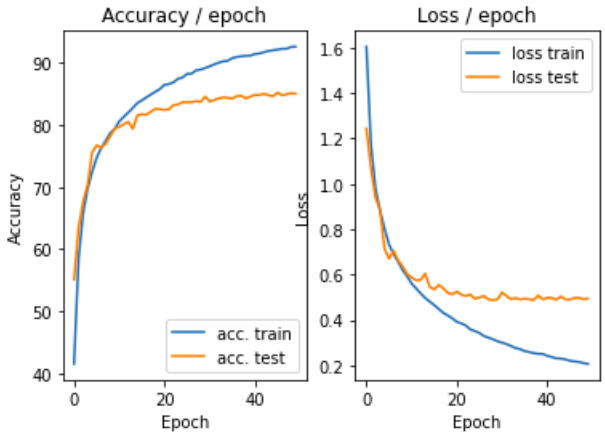


FIGURE 17 – Courbes de loss et accuracy avec Batch Normalization