Compte-Rendu des TMEs sur le Deep Learning - RDFIA

Basile Guerrapin & Timothée Poulain

17 décembre 2016

Table des matières

1 TP1 : Introduction aux Réseaux de Neurones

Séances du 19 et 26 octobre.

1.1 Objectifs et Rappels

L'objectif de ces deux premières séances est de prendre en main les réseaux de neurones : manipuler les équations de propagation et de rétro-propagation (descente de gradient) à travers un Perceptron Multi-couches puis implémenter ces équations sur deux types de données différentes :

- un problème "jouet"
- les données MNIST (caractères numériques manuscrits)

L'architecture du réseau est illustrée sur la figure 1. Les variables utilisés sont détaillés ci-dessous :

x un vecteur représentant une donnée d'entrée de taille n_x

y la vérité terrain associée à la donnée x

τ la fonction $tanh$

W_h les poids de la transformation linéaire de taille $n_h \times n_x$

b_h le biais de taille n_h associé à la transformation linéaire W_h

h le vecteur de la couche cachée de taille n_h

φ la fonction $SoftMax$

W_y les poids de la transformation linéaire de taille $n_y \times n_h$

b_y le biais de taille n_y associé à la transformation linéaire W_y

$\hat{y}$ la sortie calculée par le perceptron multi-couches de taille n_y

L la fonction de coût : l'entropie croisée entre la prédiction $\hat{y}$ et la vérité terrain y

La fonction de coût pour l'ensemble des données (X, Y) est

$$L(X,Y) = \sum_i l(y^{(i)}, \hat{y}^{(i)}) \quad avec \quad l(y,\hat{y}) = -\sum_j y_j log(\hat{y}_j) \tag{1}$$

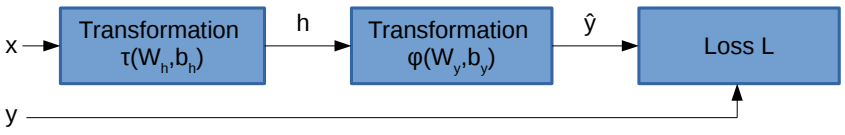

FIGURE 1 – Architecture du perceptron multi-couches utilisé

1.2 Pré-calculs mathématiques

L'objectif est d'exprimer la sortie et les couches intermédiaires du réseau de neurones en fonction des données d'entrée et de l'architecture du modèle (couches linéaires, $tanh$ et $softmax$).

Ensuite, nous exprimerons le gradient de la fonction de coût en fonction de chacun des paramètres du réseau en vue d'appliquer l'algorithme de rétro-propagation du gradient sur le réseau de neurones. Il s'agit de dériver l'erreur par "étapes" : La dérivée de l'erreur se compose par rapport à chacun des paramètres du réseau en fonction des sorties des couches intermédiaires. Ainsi, chacune des couches

peuvent calculer leur propre contribution aux gradients des couches précédentes et le calcul du gradient est délégué aux différents modules du réseau de neurones.

Enfin nous pourrons implémenter les équations de propagation et de rétro-propagation dans notre code Matlab. Les résultats sont analysés et discutés dans la section 1.3.

Afin d'expliciter au mieux les calculs, on notera $\tilde{h}$ et $\tilde{y}$ les valeurs des neurones avant l'application respective des fonctions $tanh$ et $softmax$.

1.2.1 Equations *forward*

La relation entre la couche d'entrée x et la couche cachée h est la suivante :

$$\tilde{h} = W_h \times x + b_h$$
$$h = \tau(\tilde{h})$$

La relation entre la couche cachée h et la sortie calculée $\hat{y}$ est :

$$\tilde{y} = W_y \times h + b_y$$
$$\hat{y} = \varphi(\tilde{y})$$

1.2.2 Equations *backward*

Nous commençons par définir l'expression du gradient de la prédiction $\hat{y}$ par rapport à $\tilde{y}$. La fonction qui lie ces valeurs est la fonction $Softmax$: $\hat{y}_i = \frac{e^{\tilde{y}_i}}{\sum\limits_j e^{\tilde{y}_j}}$

On distingue deux cas :

Pour $i = j$: $\frac{\partial \hat{y}_i}{\partial \tilde{y}_i} = \frac{e^{\tilde{y}_i} \sum\limits_j e^{\tilde{y}_j} - e^{\tilde{y}_i} \times e^{\tilde{y}_i}}{(\sum\limits_j e^{\tilde{y}_j})^2} = \frac{e^{\tilde{y}_i}}{\sum\limits_j e^{\tilde{y}_j}} \frac{\left(\sum\limits_j e^{\tilde{y}_j} - e^{\tilde{y}_i}\right)}{\sum\limits_j e^{\tilde{y}_j}} = \hat{y}_j(1 - \hat{y}_j)$

Pour $i \neq j$: $\frac{\partial \hat{y}_j}{\partial \tilde{y}_i} = \frac{-e^{\tilde{y}_j} \times e^{\tilde{y}_i}}{(\sum\limits_j e^{\tilde{y}_j})^2} = -\hat{y}_j \hat{y}_i$

On arrive ensuite à :

$$\frac{\partial l}{\partial \tilde{y}_i} = \frac{\partial}{\partial \tilde{y}_i}\Big(-\sum_k y_k \log \hat{y}_k \Big) = -\sum_k \frac{\partial}{\partial \tilde{y}_i}\Big(y_k \log \hat{y}_k \Big) = -y_i + y_i \hat{y}_i + \sum_{k \neq i} y_k \hat{y}_i = \hat{y}_i - y_i$$

Le résultat obtenu est nommé signal d'erreur de la couche y et s'écrit sous forme vectorielle :

$$\delta_y = \hat{y} - y$$

Ensuite, on souhaite calculer le gradient de l'erreur par rapport à l'entrée de la couche cachée $\tilde{h}$:

$$\frac{\partial l}{\partial \tilde{h}_i} = \frac{\partial l}{\partial \tilde{y}_i} \frac{\partial \tilde{y}_i}{\partial h_i} \frac{\partial h_i}{\partial \tilde{h}_i} = \Big(\sum_j \delta_y(j) W_y(j,i) \Big) \tanh'(\tilde{h}_i)$$

Ce résultat est appelé signal d'erreur de la couche h et s'écrit sous forme vectorielle :

$$\delta_h = \Big(W_y^T \delta_y \Big) \odot \tanh'(\tilde{h})$$

Enfin, on peut en déduire les équations de gradients par rapport à chacun des paramètres des couches h et y :

$$\frac{\partial l}{\partial W_y} = \delta_y h^T$$
$$\frac{\partial l}{\partial b_y} = \delta_y$$
$$\frac{\partial l}{\partial W_h} = \delta_h x^T$$
$$\frac{\partial l}{\partial b_h} = \delta_h$$

1.3 Analyse des Résultats

1.3.1 Application au problème jouet

Dans un premier temps, nous présentons la démarche générale de notre code. Puis nous discutons des résultats sur les données du problème "jouet". Les données du problème "jouet" sont en deux dimensions et il s'agit d'un problème de classification binaire. Nous disposons de 200 exemples d'apprentissage et 200 exemples de test.

Comme indiqué dans le sujet de TME, nous avons implémenté les fonctions suivantes :

- *initMLP(nx, nh, ny)* qui retourne le réseaux de neurones sous forme d'une structure dont les paramètres ont été initialisés en fonction de la taille de chacune des couches.
- *forward(net,X)* qui propage les données d'entrée à travers le réseau et renvoie les sorties intermédiaires ainsi que la prédiction.
- *backward(net,out,Y,eta)* qui rétro-propage les gradients de l'erreur calculé par rapport à chacun des paramètres du réseau.
- *loss_ accuracy(Yhat, Y)* qui permet d'évaluer la fonction de coût et la précision des prédictions par rapport aux vérités terrains.

Dans le but éventuel de paralléliser les calculs lors de l'utilisation de mini-batchs, nous n'avons utilisés que des opérations matricielles pour chacune des fonctions précédentes.

Quelques mots sur la phase d'apprentissage implémentée :
Tout d'abord, les données de test et d'apprentissage sont centrées puis réduites. Pour chaque époque, nous permutons aléatoirement les exemples d'apprentissage afin de ne jamais répéter la même séquence d'apprentissage d'une époque à l'autre. Ensuite, tous les mini-batchs sont propagés puis rétro-propagés à travers le réseau de neurones. Au début de chaque époque, la fonction de coût est évaluée sur toute la base d'apprentissage. A la fin de chaque époque, la précision (*accuracy*) des prédictions effectuées sur la base de test est évaluée.

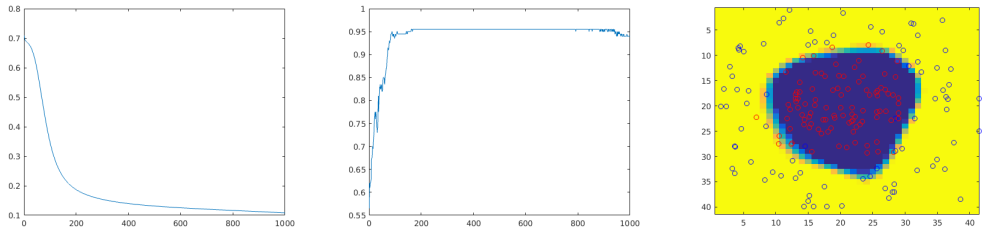

FIGURE 2 – Evolution de la fonction de coût sur la base d'apprentissage (à gauche), évolution de la précision des prédictions sur la base de test (au milieu) et frontière de décision dans l'espace des données (à droite) pour les hyper-paramètres suivants : $batch\_size = 50$; $nb\_epochs = 1000$; $eta = 0.1$; $nh = 10$.

On a ainsi une courbe indiquant l'évolution de la fonction de coût pendant l'apprentissage, une courbe traduisant l'évolution de la précision en inférence sur la base de test et (grâce au code fourni dans le sujet de TP) une représentation de la frontière de décision de notre algorithme dans l'espace des données en deux dimensions. Nous présentons différents résultats ci-dessous et analysons l'influence des hyper-paramètres de l'algorithme.

Les différents hyper-paramètres sont les suivants :

- *batch_ size* : la taille du mini-batch utilisé pour chaque itération d'apprentissage
- *nb_ epochs* : le nombre d'époques de l'algorithme d'apprentissage
- *eta* : le taux d'apprentissage

- nh : le nombre de neurones dans la couche cachée du perceptron multi-couches.

On constate que les hyper-paramètres choisis pour l'expérience présentée en figure 2 sont pertinents : la fonction de coût converge vers une valeur proche de zéro et la précision des prédictions atteint une valeur asymptotique proche de 0.95. De plus, la représentation de la frontière de décision dans l'espace des données correspond à la classification attendue.

Concernant l'influence des hyper-paramètres :

- la taille du mini-batch définit la variabilité du modèle par rapport aux données d'apprentissage : si l'on met à jour les paramètres pour chaque exemple, le modèle va fluctuer de manière plus importante que si les paramètres sont mis à jour après la propagation d'un groupe d'exemples. En revanche, la convergence de la courbe d'apprentissage sera plus rapide (c.f. figure 3).

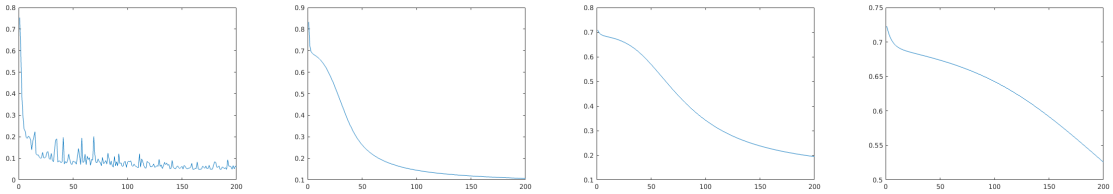

FIGURE 3 – Evolution de la fonction de coût sur la base d'apprentissage avec, de gauche à droite, $batch\_size = 1\,;\, 20\,;\, 50\,;\, 200$.

- le nombre de neurones dans la couche cachée définit à quel point le réseau de neurones va s'adapter aux données d'apprentissage. Sur la figure 4, nous constatons qu'il est nécessaire d'avoir au moins 3 neurones dans la couche cachée pour obtenir une frontière de décision acceptable. Aussi, nous pouvons penser qu'un grand nombre de neurones conduit au sur-apprentissage, mais pour ce problème, la figure de droite (nh = 1000) ne confirme pas cette idée. Enfin, une intuition qui se dégage de cette figure est que le nombre de neurones de la couche cachée correspond au nombre maximum de frontières linéaires agrégées pour générer la frontière de décision finale.

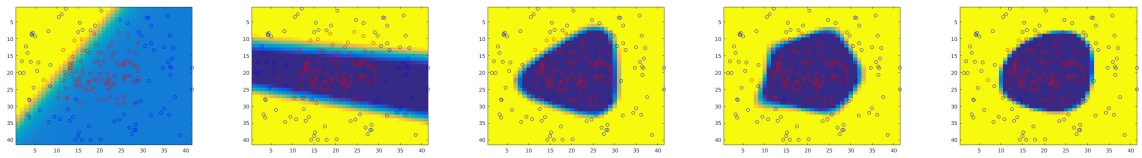

FIGURE 4 – Représentations de la frontière de décision dans l'espace des données avec, de gauche à droite, $nh = 1\,;\, 2\,;\, 3\,;\, 10\,;\, 1000$.

- le taux d'apprentissage correspond au coefficient accordé aux gradients de l'erreur par rapport aux paramètres du réseau à chaque mise à jour de ces derniers. Un taux d'apprentissage trop petit conduit à une phase d'apprentissage inachevée tandis qu'une valeur trop grande entraîne une divergence de l'algorithme. (c.f. figure 5)

1.3.2 Application aux données MNIST

Dans un second temps, nous appliquons notre perceptron multi-couches sur un autre jeu de données : MNIST. Il s'agit d'un problème de classification 10-classes. Les données sont des images de caractères numériques (0 à 9) de taille 28×28 pixels. Nous disposons de 60 000 exemples d'apprentissage et 10 000 exemples de test.

Les modifications par rapport au problème précédent :

- Normalisation des données : ne sont centrés et réduits que les pixels dont la variance n'est pas nulle (sinon division par zéro).

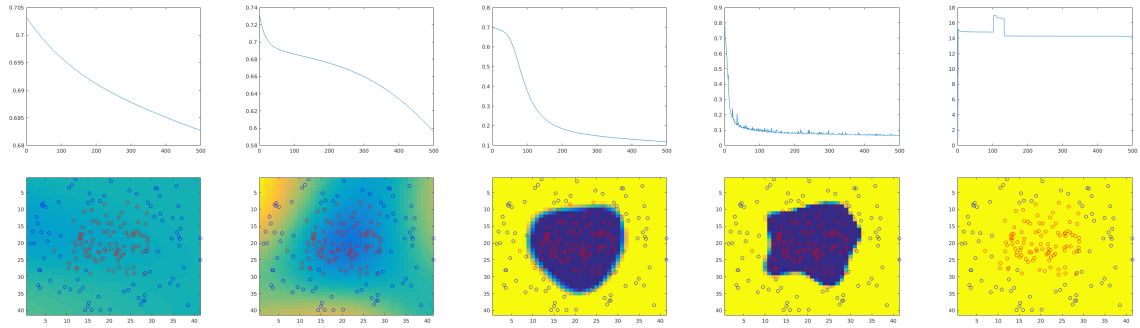

FIGURE 5 – Représentations de la fonction de coût en apprentissage et de la frontière de décision dans l'espace des données avec, de gauche à droite, $eta = 0.001\,;\ 0.01\,;\ 0.1\,;\ 1\,;\ 5$.

- Changement de la taille des couches d'entrée et de sortie : il y a maintenant 784 neurones d'entrée (nombre de pixels par image) et 10 neurones de sortie (nombre de classes).

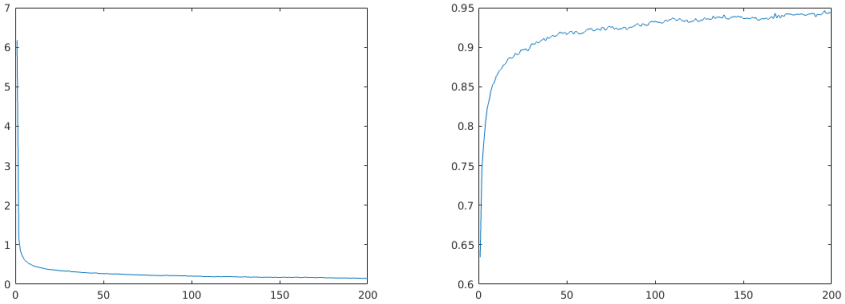

FIGURE 6 – Evolution de la fonction de coût sur la base d'apprentissage (à gauche) et évolution de la précision des prédictions sur la base de test (à droite) sur les données MNIST avec les hyper-paramètres suivants : $batch\_size = 1000\,;\ nb\_epochs = 200\,;\ eta = 0.1\,;\ h = 100$.

Sur la figure 6, nous constatons que l'apprentissage est correct puisque la précision des prédictions sur la base de test atteint un taux aux alentours de 0.94. Par rapport au problème précédent, les données ont beaucoup plus de dimensions et sont plus nombreuses, nous avons donc réduit le nombre d'époques d'apprentissage à 200 pour cet exercice. Par ailleurs, il n'est plus possible de visualiser la frontière de décision dans l'espace des données, ce dernier n'étant plus en deux dimensions.

Puisque les données d'entrée ont un nombre de dimensions supérieur au problème précédent, nous avons, par intuition, choisi d'utiliser une couche cachée de 100 neurones. Si l'on compare avec les résultats obtenus en utilisant une couche cachée composée de 10 neurones, comme présenté en figure 7, nous constatons que les performances atteintes sont moindres (inférieures à 0.9). Nous pourrons envisager d'utiliser une base de validation pour tester une grande quantité de valeur de nh et choisir celle qui maximise la précision des prédictions sur cette base de validation.

Enfin, nous choisissons de comparer l'initialisation habituelle des paramètres du perceptron multi-couches à une initialisation Xavier. Les poids des matrices sont toujours tirés selon une loi gaussienne mais sont divisés par la racine carrée du nombre de neurones d'entrée de la couche. Les biais sont initialisés à zéro. Les résultats sont présentés en figure 8. Nous constatons une légère amélioration de la précision des prédictions sur la base de test (de 0.94 à 0.96), cependant cette évolution est plus fluctuante que sans l'initialisation Xavier (c.f. figure 6).

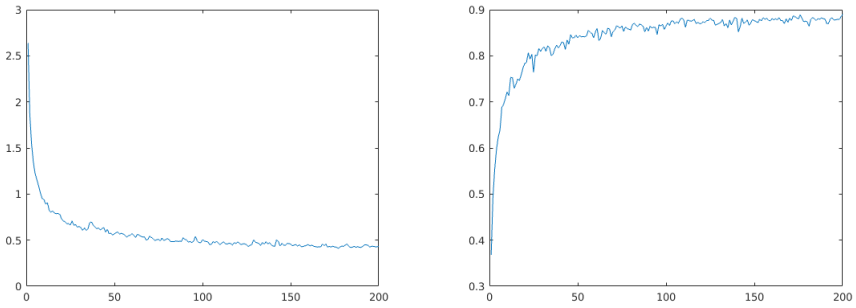

FIGURE 7 – Evolution de la fonction de coût sur la base d'apprentissage (à gauche) et évolution de la précision des prédictions sur la base de test (à droite) sur les données MNIST avec les hyper-paramètres suivants : $batch\_size = 1000$; $nb\_epochs = 200$; $eta = 0.1$; $h = 10$.

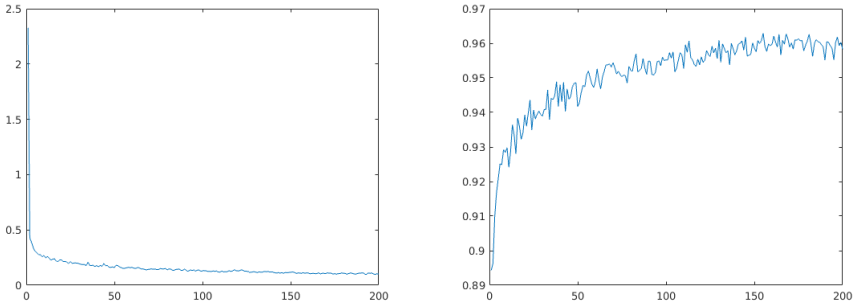

FIGURE 8 – Evolution de la fonction de coût sur la base d'apprentissage (à gauche) et évolution de la précision des prédictions sur la base de test (à droite) sur les données MNIST avec les hyper-paramètres suivants : $batch\_size = 1000$; $nb\_epochs = 200$; $eta = 0.1$; $h = 100$ et en effectuant une initialisation Xavier.

2 TP2 : Prise en main de MatConvNet

Séance du 2 novembre.

2.1 Introduction

Pour ce TP2, l'objectif est de comprendre et d'apprivoiser les différentes fonctions de MatConvNet. En effet, MatConvNet est une toolbox permettant l'implémentation de CNN (Convolutional neural network) pour la Computer Vision. Nous étudions une architecture de réseau de neurones convolutifs déjà implémentée.

2.2 Question 1

Dans le réseau "net" nous constatons différentes couches :

- Couche de Convolution : La couche de convolution est la couche indispensable dans la création d'un réseau convolutif, se composant d'un certain nombre de filtres. Chaque petit filtre est étendu au travers de la profondeur. Durant la phase de convolution, nous glissons chaque filtre sur les différentes dimensions de l'objet en entrée afin d'en obtenir une valeur . Nous produisons alors pour chaque filtre une nouvelle carte de l'image, que nous empilons. La couche de convolution dispose de quatre hyper-paramètres :
  - la taille spatiale du filtre
  - le nombre de filtres différents à appliquer

— le stride : le pas permettant de décaler le filtre à chaque itération

— le padding [1] permet d'ajouter à la bordure extérieure des valeurs de 0 permettant ainsi de ne pas perdre l'information.

- Couche de pooling : La couche de pooling permet de sous-échantillonner l'image. Elle réduit spatialement la taille de l'input. En effet, cette couche s'intéresse uniquement à la spatialité de l'entrée, la profondeur reste la même. Cela permet de créer de l'invariance par translation dans notre modèle. Il existe différentes fonctions de pooling utilisées pour cette couche :

  — "max pooling" : On effectue un chevauchement sur différentes régions et on ne conserve que la valeur maximale.

  — "sum pooling" : On fait de même en sommant tous les éléments.

  — "L2-norm pooling" , "avg pooling", etc.

- Couche de RELU : Elle permet de simplifier le calcul pour la back-propagation du gradient. On évite le "Vanishing" du gradient. Cette couche souvent appelée c̈ouche d'activation; ajoute une composante non-linéaire ce qui a pour but d'éviter une linéarisation complète du modèle. La RELU permet aussi de rendre l'apprentissage plus rapide.
- Couche de normalisation : Cette couche permet de normaliser. Elle permet d'amortir les réponses uniformément grandes en les diminuant et met en évidence les neurones stimulés.
- Couche de Softmax : Cette couche utilisée en fin du réseau, transforme nos données en une distribution probabiliste.

2.3 Question 2

Entre l'image originale est l'image en entrée du réseau, on effectue une opération de *resizing* et de normalisation (soustraction de l'image moyenne). En effet, un des inconvénients des réseaux de neurones profonds est que la taille d'entrée (nombre de neurones de la couche initiale) est fixe. Ainsi, on ne pourra pas utiliser une image $300 \times 300 \times 3$ sans pré-traitements.

2.4 Question 3

La figure 9 présente les tailles de toutes les structures du réseau.

2.5 Question 4

La relation entre les tailles des entrées et des sorties des couches de convolution est la suivante : **Partie entière (taille de l'objet - (taille du filtre - stride)) /stride = partie entière(taille de la prochaine couche)**

Dans l'expresison ci-dessus, la "taille du filtre" correspond à la hauteur ou/et à la largeur et non à la profondeur du filtre.

La profondeur de la couche est choisie arbitrairement sur les différentes couches.

En figure 10, nous présentons le schéma illustrant une opération de filtrage. Il s'agit de celle effectuée sur l'image initiale (input $224 \times 224 \times 3$).

2.6 Question 5

La figure 11 présente les différentes cartes obtenues pour les premières couches du réseau. Il est intéressant de constater que pour la couche de convolution, certains filtres se focalisent sur des motifs horizontaux tandis que d'autres se focalisent sur des motifs verticaux. C'est par composition de ces motifs au fil des différentes couches de convolution que le réseaux de neurones est capable d'identifier des motifs plus complexes.

1. http ://cs231n.github.io/convolutional-networks

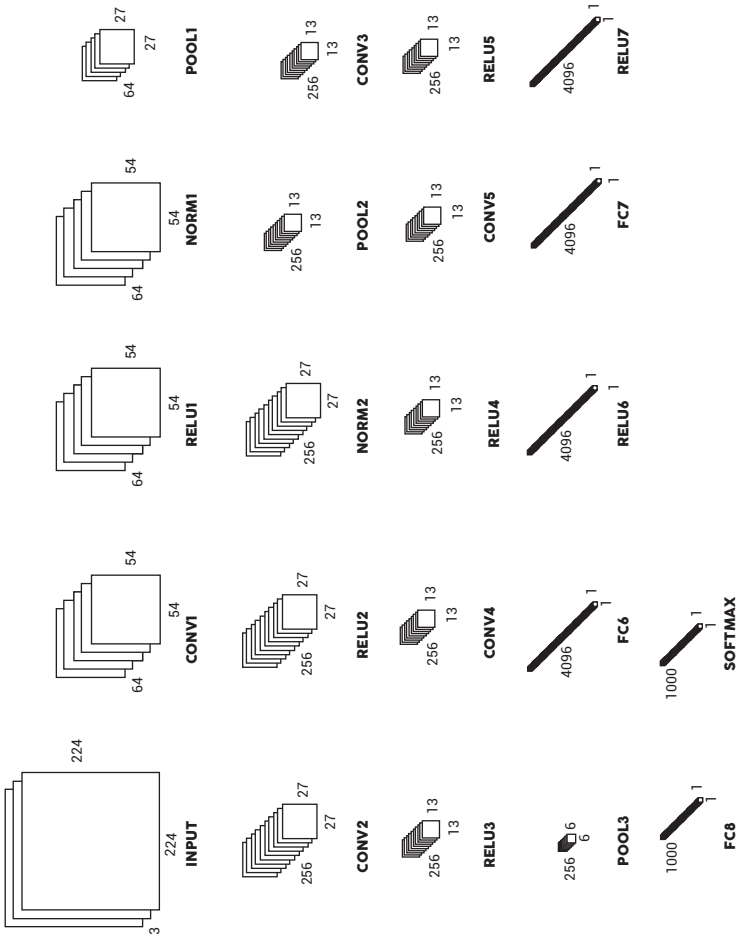

FIGURE 9 – figure
Tailles des différentes couches du modele

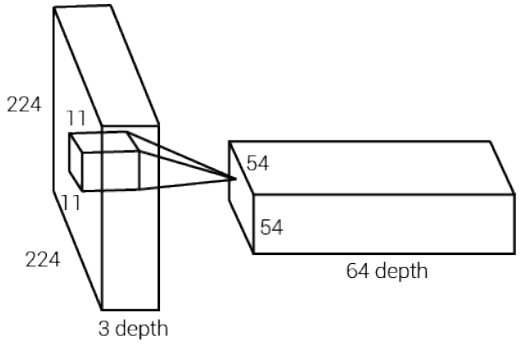

FIGURE 10 – Illustration d'une opération de filtrage sur une image d'entrée de taille $224 \times 224 \times 3$ pour un nombre de 64 filtres de taille 11×11

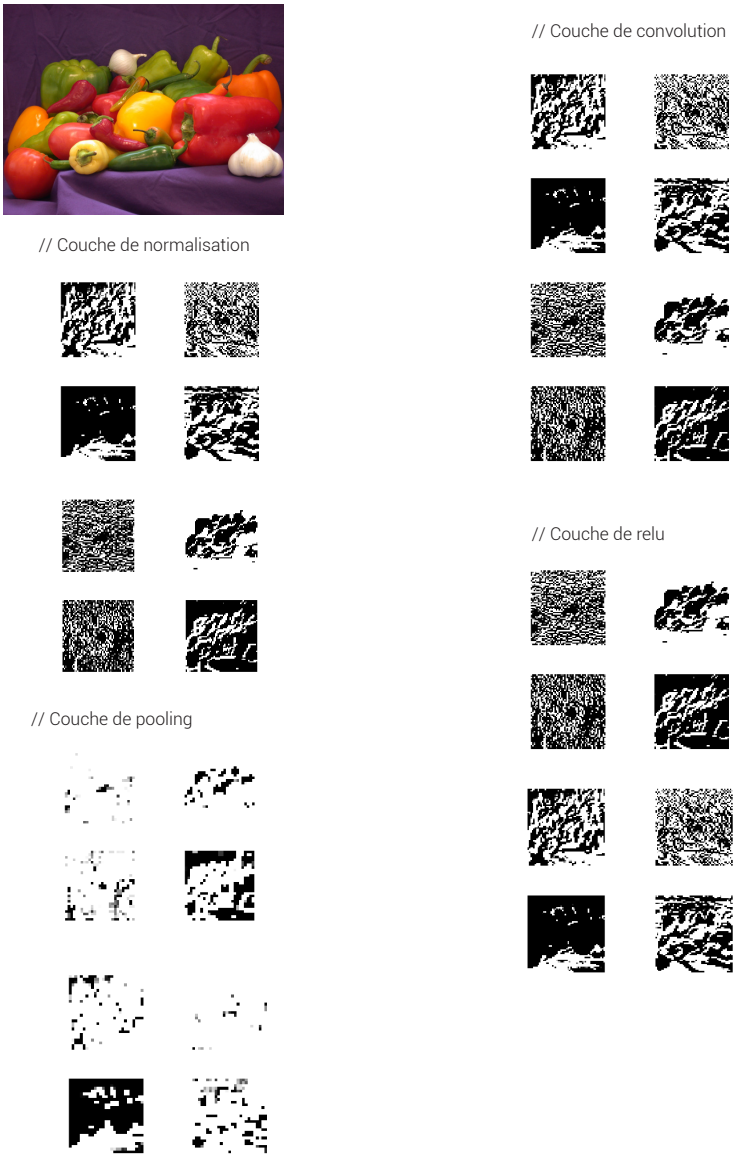

FIGURE 11 – Cartes obtenues à différentes couches

2.7 Question 6 & 7

Les couches de RELU permettent de convertir toutes les valeurs négatives en valeurs nulles. Elles ne modifient pas les résultats positifs. La couche de Softmax permet de transformer nos données en une distribution de probabilités : dans le premier exemple présenté dans le paragraphe suivant, on passe d'une valeur de 15.01 en couche fully-connected à une probabilité de 0.39% de classification correcte de l'image.

Les scores des dernières couches :

pears.npg

- Couches fc7 (18-layers) → 19.53 index 3130/4096 sur un vecteur de 4096
- Couches relu (19-layers) → 19.53 index 3130/ 4096 sur un vecteur de 4096
- Couches fc8 (20-layers) → 15.013 index 952 / 1000 classes

— Couches Softmax (21-layers) → 0,393 index 952 / 1000 classes (lemon)

peppers.png

— Couches fc7 (18-layers) → 12.44 index 2896/4096 sur un vecteur de 4096
— Couches relu (19-layers) → 12.44 index 2896/ 4096 sur un vecteur de 4096
— Couches fc8 (20-layers) → 13.51 index 946 / 1000 classes
— Couches Softmax (21-layers) → 0,7041 index 946 / 1000 classes (bell pepper)

La première image est mal catégorisée ("lemon" au lieu de "pears") tandis que la seconde est bien catégorisée. Il est intéressant de constater que la probabilité en sortie de softmax peut s'interpréter comme une "confiance" : dans le second exemple, la confiance dans la bonne classe est de 0.70 tandis que dans le premier (qui est faux) la confiance dans la classe prédite n'est que de 0.39.

2.8 Question 8

A l'aide de la documentation de MatConvNet, nous avons regardé plus précisément la passe backward. MatConvNet crée une seconde architecture (network) qui est l'inverse de la première afin d'avoir en mémoire directement les dérivées partielles pour la back-propagation. En effet, pour chaque couche de *forward*, il existe une couche correspondante dans le réseau "inverse" qui permet de propager les signaux d'erreurs afin de mettre à jour les paramètres. A l'opposé, dans notre cas, nous effectuons les passes forward et backward sur le même réseau.

meilleur

3 TP3 : Classification par feature extraction avec CNN pré-entrainé

Séance du 23 novembre.

3.1 Objectifs

Dans ce TP, nous utilisons un modèle de réseau de neurones convolutionnel déjà entraîné sur la base ImageNet (CNN-F) en vue de l'adapter à notre problème. On parle de *Transfer Learning*. Il existe deux principales méthodes : *finetuning* et *feature extraction*. Le *finetuning* consiste à ré-entraîner le réseau sur les données de notre problème ; ainsi, les paramètres vont s'adapter pour correspondre parfaitement à notre tâche de classification. Le *feature extraction* est une technique qui consiste à extraire d'une couche cachée du réseau pré-entraîné la représentation d'un échantillon de la base (cette représentation est appelée "descripteur *deep*").

Dans le cadre de ce TP, nous traiterons uniquement la méthode de *feature extraction*.

Le descripteur obtenu est une représentation de l'image d'entrée dont les composantes ont une plus forte valeur sémantique que les pixels de l'image. Contrairement au *finetuning*, les paramètres du modèle qui permettent de générer le descripteur ne sont pas du tout ré-appris. On suppose donc que la tâche pour laquelle le réseau de neurones a été pré-entraîné est une tâche similaire à la notre. Nous souhaitons adapter les descripteurs issus d'un problème de classification 1000-classes à nos images 15-Scene qui est donc un problème de classification 15-classes.

Enfin, nous utilisons les descripteurs comme données d'apprentissage d'un ensemble de SVMs linéaire en mode *One Against All*.

3.2 Analyse des Résultats

Dans un premier temps, nous devons extraire les descripteurs *deep* à la couche "relu7". Il s'agit de la couche 19 dans notre réseau de neurones (matlab étant indexé à 1, nous sélectionnons donc la couche d'indice 20). Cette couche intervient juste après la dernière couche "fully-connected" du réseau de neurones utilisé.
Nous utilisons ensuite les fonctions d'entraînement fournies : "train_classifier" permet d'entraîner 15 SVMs linéaires et "predict_classifier" qui permet d'inférer, pour chaque image, la classe prédite. Nous appliquons enfin la fonction d'*accuracy* de la même manière que dans le TP1.

Nous avons choisi de calculer l'*accuracy* pour chacune des classes. Sur la figure 12, nous avons représenté l'*accuracy* pour différentes valeurs de C. C est un hyper-paramètre lié à la régularisation du modèle. Aussi, nous avons représenté les valeurs obtenues lors de la série de TPs sur les descripteurs SIFT. Nous constatons qu'une valeur de C grande tend à "lisser" l'*accuracy* en fonction de chacune des classes. La variation de l'*accuracy* en fonction de la valeur de C est légère.
Par ailleurs, l'information essentielle de la figure 12 est que les résultats obtenus en utilisant les descripteurs *deep* (courbe jaune, rouge, bleue-ciel) sont bien meilleurs que ceux obtenus avec les descripteurs SIFT (courbe bleue). Cela confirme notre intuition de départ et indique que la valeur sémantique des descripteurs *deep* est plus importante que celle des descripteurs SIFT.

Dans un second temps, nous choisissons de comparer nos résultats en utilisant des descripteurs *deep* extraits d'une autre couche du réseau de neurones : la couche "relu6" qui correspond donc à la couche d'indice 16. Les résultats sont comparés pour $C = 1000$ avec les descripteurs extraits à la couche "relu7" sur la figure 13. Nous observons que les résultats ainsi obtenus (courbe bleue) sont en

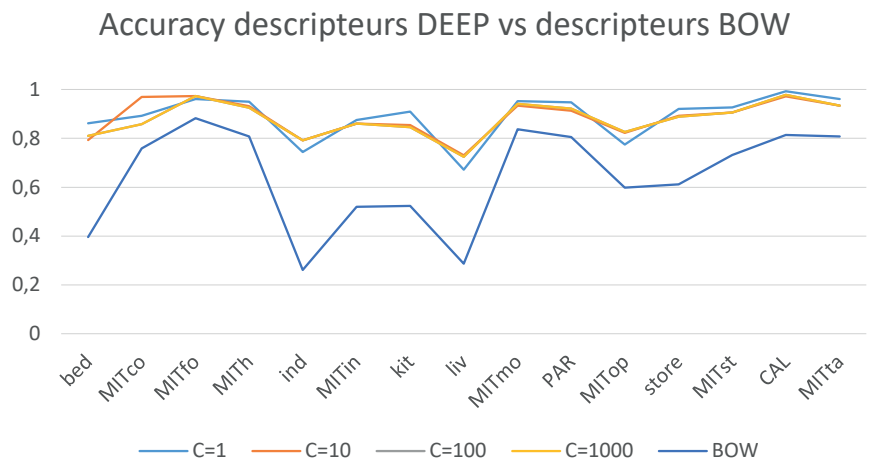

FIGURE 12 – Comparaison de l'*accuracy* en utilisant des descripteurs BOW (avec SIFT) et des descripteurs *deep* pour différentes valeurs de C.

moyenne légèrement inférieurs : il est donc plus pertinent d'extraire les descripteurs *deep* à la couche "relu7".

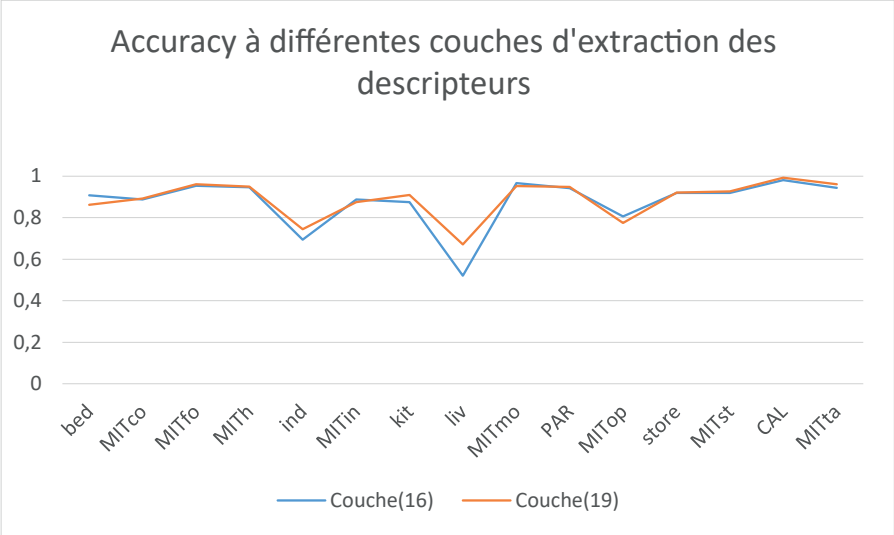

FIGURE 13 – Comparaison de l'*accuracy* sur les descripteurs *deep* extraits sur deux couches cachées différentes.

4 TP4 : Apprentissage d'un réseau de convolution sur 15-Scene

Séances du 30 novembre et 7 décembre.

4.1 Objectifs

Après avoir expérimenté des techniques de *transfer learning* dans le TP précédent, nous allons maintenant définir nous même l'architecture du réseau de neurones et l'entraîner *from scratch* sur la base 15-Scene.

Dans un premier temps, nous concevons le réseau de neurones comme indiqué dans le sujet de TME. Dans un second temps nous étudierons différentes méthodes de régularisation pour améliorer les performances du modèle.

4.2 Apprentissage *from scratch* du modèle

4.2.1 Réponses aux questions

- Afin d'évaluer la capacité de généralisation d'un modèle, il est important de découper les données en deux sous-ensembles : l'ensemble d'apprentissage (train) et l'ensemble de test (test). Le premier permet d'apprendre le modèle et le second d'évaluer la capacité de généralisation. Le troisième sous-ensemble, appelé ensemble de validation (val), n'est pas forcément nécessaire, il intervient lors de l'utilisation de techniques de *cross validation* qui ont pour but de pré-évaluer la capacité de généralisation de différents modèles afin de sélectionner celui qui généralise le mieux.

  En général, le sous-ensemble de train est plus important que les deux autres (en terme de quantité de données), car il est bien souvent nécessaire d'avoir le plus de données possibles pour effectuer l'entraînement d'un réseau de neurones.

- Le fait de réduire les images en 64×64 diminue le nombre de neurones sur chacune des couches du réseaux de neurones. Cela a pour effet de diminuer la place utilisée en mémoire mais aussi le nombre de calculs à effectuer, l'apprentissage sera donc plus rapide.

- Les contraintes sur la couche 'fc2' sont les suivantes :

  1 - nous devons utiliser des opérations de filtrage pour générer une couche fully connected

  2 - il est nécessaire de connaître la dimension spatiale d'entrée pour adapter la taille du filtre en conséquence.

  En entrée, on a un vecteur de taille $8 \times 8 \times 10$, on souhaite connecter chacune des composantes avec les composantes du vecteur de sortie de la couche (de taille 1×15, le nombre de classes). On applique donc 15 filtres de taille 8×8 au vecteur d'entrée.

- Le réseau contient : 820 paramètres $(10 \times 9 \times 9 + 10)$ pour la première couche et 9750 paramètres $(10 \times 8 \times 8 \times 15 + 10 \times 15)$ pour la deuxième couche. Au total, cela fait 10570 paramètres. Une intuition générale est qu'il faut plus d'exemples d'apprentissage que de paramètres pour éviter de sur-apprendre (ce qui n'est donc pas notre cas).
- En comparaison avec le réseau CNN-F, ce dernier possède entre 20 millions et 30 millions de paramètres (en fonction de la taille d'entrée de l'image définie). Notre réseau de neurones dispose donc de 2000 à 3000 fois moins de paramètres.

  On souhaite maintenant comparer à l'approche BOW+SVMs effectuée dans les précédents TPs. Il n'y a pas de paramètres pour la partie "génération des BOWs", il y a seulement des paramètres dans l'étape de classification avec les SVMs. On avait 15 SVMs possédant chacun 1002 paramètres (1001 pour les poids, plus 1 pour le biais). Au total l'approche BOW+SVMs utilise donc 15030 paramètres. Notre réseau de neurones dispose donc d'un nombre assez similaire de paramètres, même si ce nombre est légèrement inférieur.

- La *backpropagation* a pour objectif de minimiser une fonction de coût. Lorsque l'on utilise un modèle paramétré, le principe de la backpropagation est de propager le gradient de l'erreur afin de mettre à jour les différents paramètres du modèle. Pour les réseaux de neurones (dont l'architecture est sous forme de couches), on applique alors le théorème de dérivation des fonctions composées pour chacune des couches.
- Le pas d'apprentissage permet, pour chaque exemple (ou groupe d'exemples) d'apprentissage, de pondérer la mise à jour des paramètres. Le choix d'un pas d'apprentissage adapté permet d'assurer la convergence du modèle, si bien sur le modèle est efficace. On peut choisir une méthode "Grid Search" afin de trouver le pas d'apprentissage "optimal" pour notre modèle.

  Le *mini-batch* est un compromis entre la méthode de descente de gradient *batch* et le gradient stochastique. En effet, l'idée est d'effectuer une mise à jour des paramètres pour l'ensemble des exemples contenus dans ce *mini batch*. Son effet est de conserver une vitesse de convergence égale à celle du SGB et une stabilité de la convergence améliorée grâce au gradient *batch*.

- L'erreur à la première époque correspond à l'erreur calculée au dernier *batch* aussi bien pour le "train", 3.59, que pour la "validation", 2.707.

4.2.2 Analyse des résultats

Sur la figure 14, on constate que les courbes d'apprentissage suivent une évolution monotone et décroissent au fil des itérations. En revanche, l'évaluation de la fonction de coût sur l'ensemble de test augmente en continue après la dixième itération. Les courbes d'évaluation "top-1 error" et "top-5 error" sont constantes sur l'ensemble de test et ne suivent plus celles de l'ensemble d'apprentissage. Le phénomène observé est le sur-apprentissage : le nombre de paramètres étant supérieur au nombre de données d'apprentissage, le modèle apprend "trop" les données d'apprentissage et perd la capacité de généralisation.

On note que le top-1 error sur l'ensemble de test est aux alentours de 0,78.

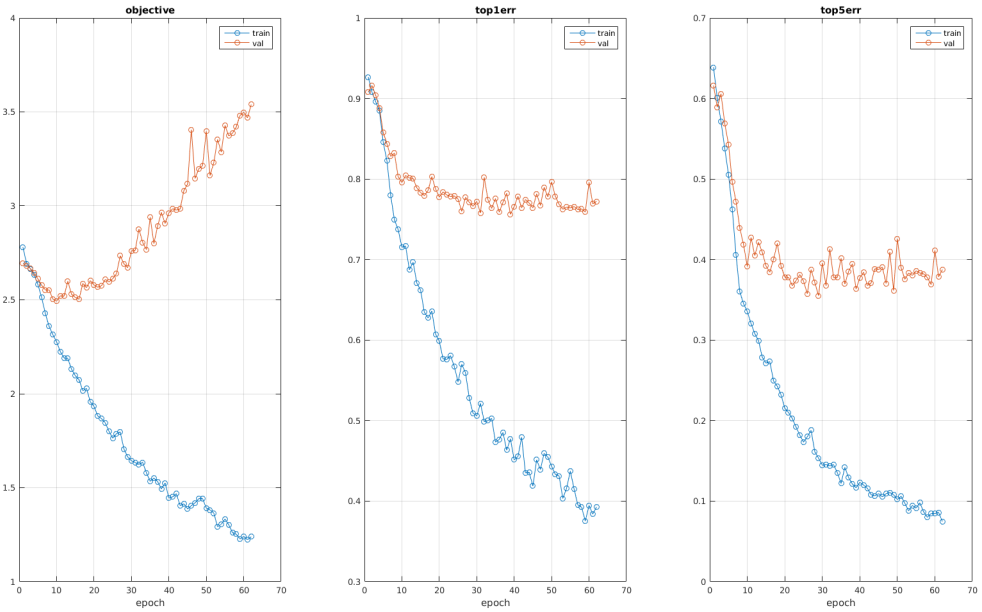

FIGURE 14 – Représentation de la fonction de coût (à gauche), du *top-1 error* (au milieu) et du *top-5 errors* (à droite) du réseaux de neurones convolutif "from scratch".

Dans la section suivante, nous allons voir différentes méthodes pour contrer le phénomène de sur-apprentissage.

4.3 Améliorations des résultats

4.3.1 Augmentation du nombre d'exemples d'apprentissage par data augmentation

Afin d'augmenter le nombre de données, nous effectuons lors de la constitution du *mini-batch* une opération de symétrie horizontale pour chacune des images avec une probabilité de 0,5. Cela a pour conséquence d'augmenter la diversité des images de l'ensemble d'apprentissage : le réseau de neurones s'entraîne maintenant sur deux fois plus de données qu'auparavant. Cependant, une image et sa symétrie possèdent de fortes similarités entre elles (distribution des pixels par exemple). En revanche, si l'on compare avec un ensemble d'apprentissage composé d'autant d'exemples, mais tous originaux, ce second cas de figure serait plus efficace car des exemples originaux offrent une meilleure diversité.

En terme de performances, sur la figure 15, on constate une amélioration du "top-1 error" sur l'ensemble de test d'environ 10% (on passe de 0,78 à 0,68. Cela signifie qu'augmenter les données d'apprentissage par symétrie horizontale permet effectivement d'améliorer les performances du réseau. Mais ce n'est pas suffisant, le phénomène de sur-apprentissage est toujours présent.

On note que cette approche par symétrie est pertinente pour note type de données (des images dites "naturelles"). En revanche, ce ne serait pas pertinent sur des images représentant du texte ou des nombres (MNIST par exemple) pour lesquelles la symétrie horizontale ferait perdre le sens de l'information : un 6 transformé par symétrie horizontale n'est plus un 6.

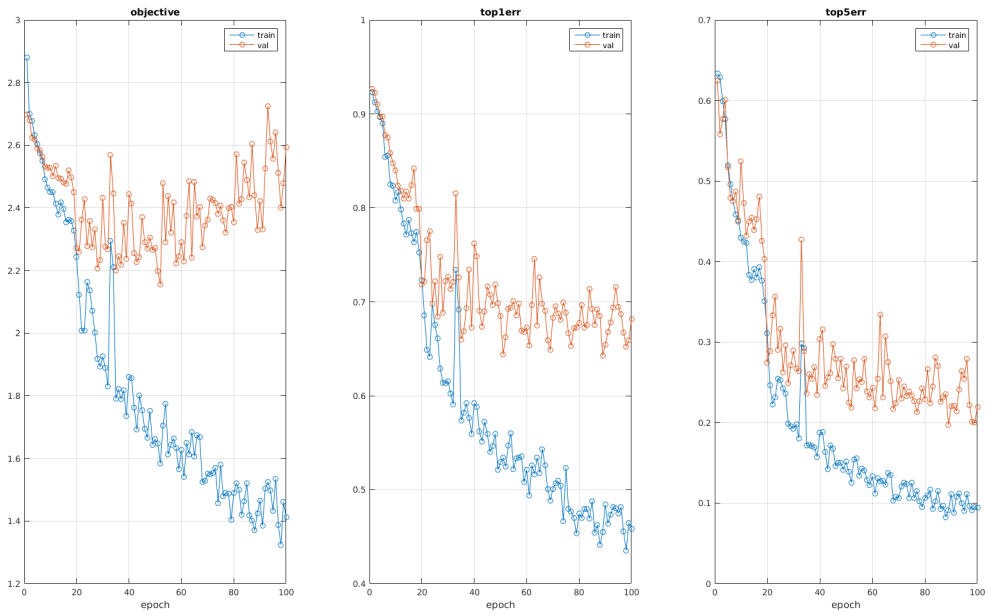

FIGURE 15 – Représentation de la fonction de coût (à gauche), du *top-1 error* (au milieu) et du *top-5 errors* (à droite) du réseaux de neurones convolutif "from scratch" avec augmentation des données.

On peut envisager d'autres méthodes pour augmenter les données : des rotations, des zooms et des perturbations des couleurs (luminosité, contraste, sous-échantillonnage).

Aussi, nous avons choisi d'évoquer la méthode "Fancy PCA" [2]. L'auteur explique qu'il augmente les données en projetant l'image sur les principaux composants : on obtient alors une image dégradée de

2. http ://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

l'image originale (en fonction du nombre de principaux composants gardés). Cela permet d'augmenter l'invariance au "bruit" du modèle. Dans l'article, cette méthode permet de réduire le top-1 error de 1%.

Enfin, nous avons expérimenté une autre méthode. De manière équiprobable, l'image subit l'une des transformations suivantes :

1 - l'image reste la même

2 - l'image subit une symétrie horizontale

3 - l'image subit une rotation de 120 degrés

4 - l'image subit une rotation de 240 degrés

Nous obtenons ainsi un ensemble d'apprentissage équivalent à quatre fois l'ensemble de départ. Les résultats incluent la normalisation et le *dropout* présentés et discutés dans les sections suivantes, nous présentons donc les résultats de cette expérience après avoir discuté du *dropout*, dans la section 4.3.4.

4.3.2 Normalisation des exemples d'apprentissage

La normalisation permet d'utiliser un taux d'apprentissage plus grand et réduit les contraintes sur l'initialisation des paramètres du réseau.

L'image moyenne est calculée seulement sur les exemples d'apprentissage car pour le cas dans lequel l'ensemble de test n'est constitué que d'une image, soustraire l'image moyenne revient à transmettre au réseau une image nulle. C'est donc l'image moyenne de l'ensemble d'apprentissage qui est soustraite aux images de test. L'hypothèse que nous faisons dans ce cas est : l'ensemble d'apprentissage et l'ensemble de test sont issus d'une même base d'images et par conséquent, l'image moyenne de l'ensemble d'apprentissage est très proche de celle de l'ensemble de test.

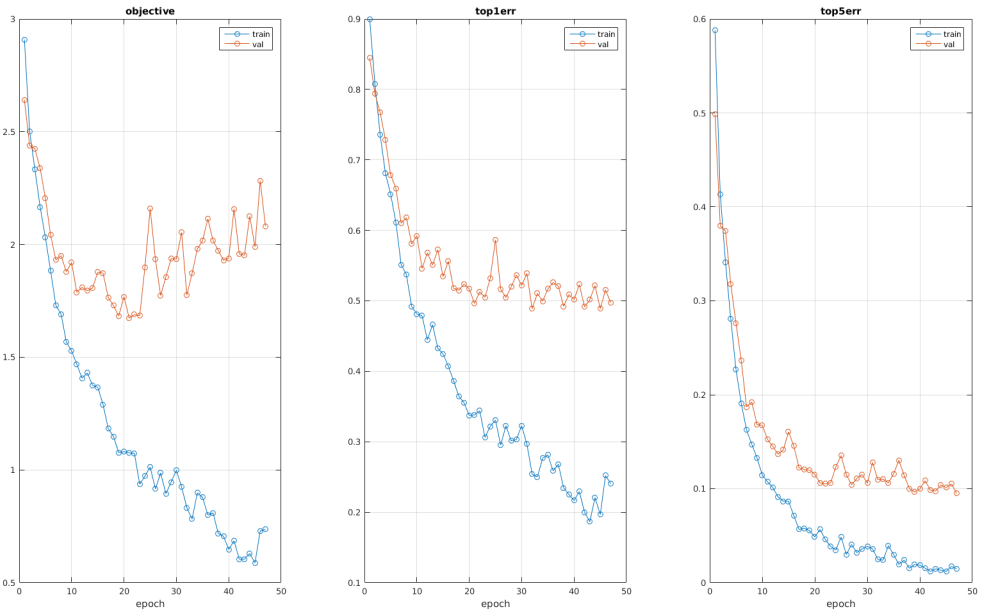

FIGURE 16 – Représentation de la fonction de coût (à gauche), du *top-1 error* (au milieu) et du *top-5 errors* (à droite) du réseaux de neurones convolutif "from scratch" avec augmentation des données et normalisation des exemples.

Dans l'architecture CNN-F, parmis les deux premières couches du réseau, une couche de normalisation "LRN" est appliquée et a pour but de normaliser la couche "cachée" du réseau.

En terme de performances, on constate en observant la figure 16 que le "top-1 error" sur l'ensemble de test a diminué d'environ 17% (de 0,68 à 0,51)

4.3.3 Régularisation du réseau par dropout

La régularisation a pour but de contrer le sur-apprentissage en évitant au modèle de devenir "trop" expert sur les données d'apprentissage (au détriment des données de test).

Implicitement, dans la fonction `cnn\_train` et plus précisément au travers de la descente de gradient par *mini-batch*, la méthode de *batch normalisation* est utilisée. Cela permet l'utilisation de taux d'apprentissage plus élévés (et donc un apprentissage plus rapide) et également d'être moins attentif à l'initialisation des paramètres du réseau de neurones [3].

Le principe d'une couche de *dropout* est d'omettre chaque connexion de manière aléatoire selon une probabilité p. L'intuition derrière cette technique est d'empêcher le réseau de neurones de "trop" apprendre les données d'apprentissage en "oubliant" une partie des informations à chaque itération (on peut y voir une analogie aux méthodes ensemblistes de *bagging*). Cela a donc pour effet d'augmenter la capacité du réseau à généraliser.

L'hyperparamètre `rate` est la probabilité de rejet d'un lien de la couche de *dropout*. On choisit de fixer cet hyperparamètre à 0,5. Cependant, certaines études [4] indiquent que la valeur optimale se situe dans un intervalle de valeurs comprises entre 0,4 et 0,8.

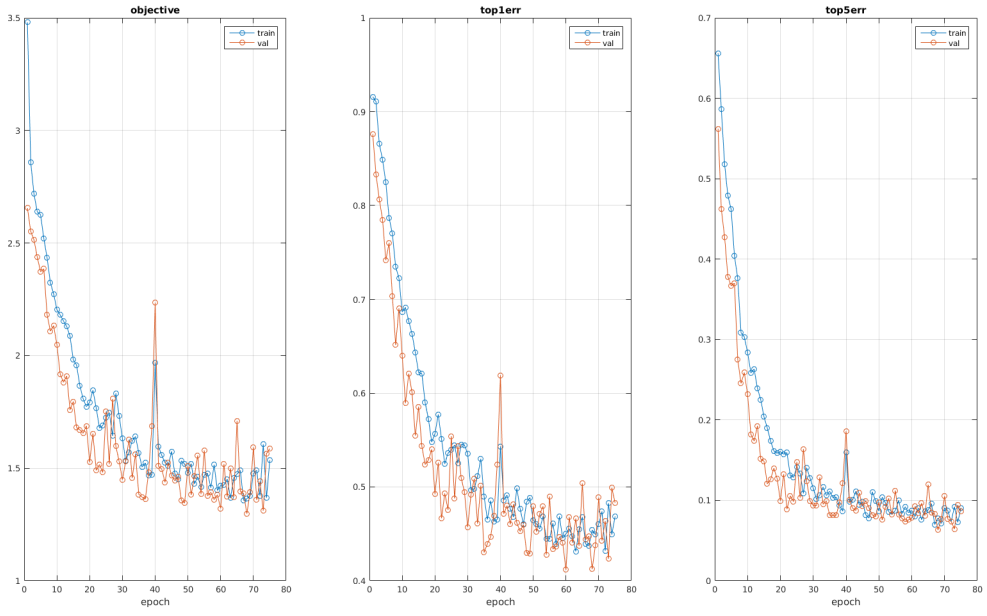

FIGURE 17 – Représentation de la fonction de coût (à gauche), du *top-1 error* (au milieu) et du *top-5 errors* (à droite) du réseaux de neurones convolutif "from scratch" avec augmentation des données, normalisation des exemples et régularisation (dropout).

Grâce à la régularisation *dropout*, on constate sur la figure 17 que la courbe "top-1 error" sur l'ensemble de test diminue de 7% (de 0,51 à 0,44). Par ailleurs, on constate que le phénomène de

3. https ://arxiv.org/pdf/1502.03167v3.pdf
4. http ://www.cs.toronto.edu/ nitish/msc_thesis.pdf

sur-apprentissage a disparu : le réseau de neurones ainsi entraîné produit des résultats aussi bons sur l'ensemble d'apprentissage que sur l'ensemble de test.

Il y a une différence dans le comportement d'une couche de *dropout* entre la phase d'apprentissage et la phase d'inférence. En effet, pendant la phase d'apprentissage, la probabilité de rejet est appliquée sur chacun des liens afin d'effectuer la régularisation. Au contraire, en inférence, la régularisation n'a plus lieu d'être (on ne cherche plus à corriger les paramètres du réseaux). Ainsi la probabilité de rejet est considérée à 1 et tous les liens sont conservés : la couche de *dropout* n'effectue alors aucune transformation sur les neurones qu'elle reçoit en entrée.

4.3.4 Présentation des résultats pour une méthode d'augmentation des données alternative

Comme indiqué en fin de section 4.3.1, nous avons expérimenté une autre méthode d'augmentation de données impliquant à la fois une symétrie horizontale, une rotation de 120 degrés et une rotation de 240 degrés. L'objectif est de comparer les deux méthodes afin de savoir si plus d'augmentation de données permet de diminuer encore l'erreur. Nous devons donc comparer la figure 18 avec la figure 17 car pour cette expérience, la normalisation et la régularisation sont également utilisées.

Nous constatons que le nombre d'itérations nécessaires à la convergence de la fonction de coût est plus grand. Cela est prévisible car il y a beaucoup plus de données différentes à traiter et ces données varient d'une époque à l'autre. Le réseau de neurones met donc plus de temps à "s'approprier" les différentes transformations des images que nous avons choisi d'effectuer. Le score en "top-1 error" obtenu est d'environ 0,6. C'est donc moins bien que le score obtenu en appliquant seulement la symétrie horizontale. On en conclue que les rotations appliquées aux images ne sont pas pertinentes pour augmenter la capacité de généralisation du modèle.

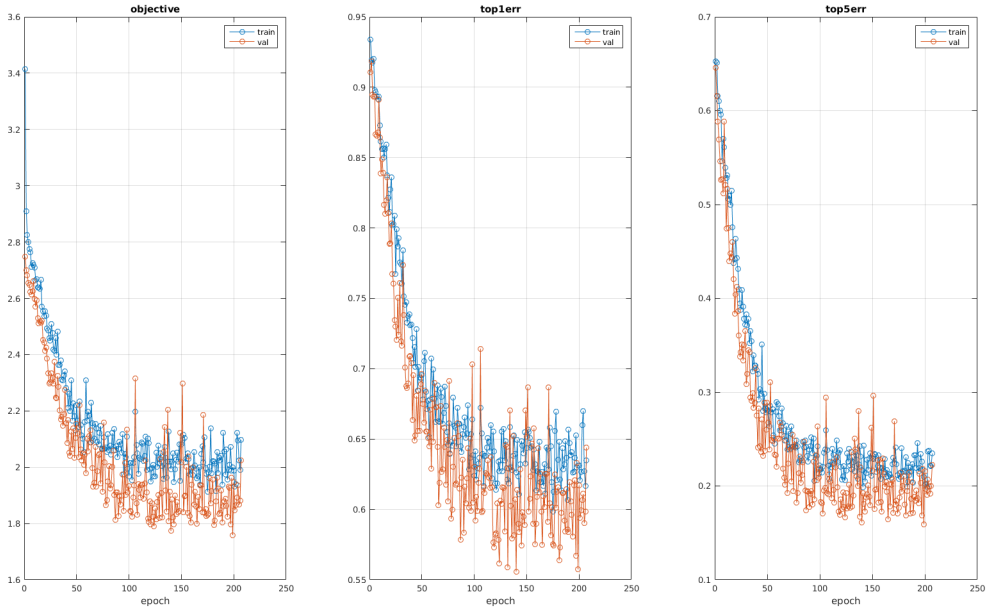

FIGURE 18 – Représentation de la fonction de coût (à gauche), du *top-1 error* (au milieu) et du *top-5 errors* (à droite) du réseaux de neurones convolutif "from scratch" avec augmentation des données par la méthode décrite à la fin de la section 4.3.1, normalisation des exemples et régularisation (dropout).