Réseau de neurones : implémentation from scratch en Python

Grégoire GOUJON – Léo PORTE

21 mai 2023

Résumé

Les réseaux de neurones se sont désormais constitués comme une place forte du Machine Learning à l'état de l'art. Ils sont utilisés dans divers domaines appliqués au traitement de jeux de données conséquents (moteurs de recherche, natural language processing, traitement d'images, reconnaissance faciale...), nécessitant notamment de donner naissance à des modèles suffisamment expressifs pour extraire des connaissances fiables de ces données.
Dans le cadre de ce projet, nous avons donc implémenté en Python/NumPy lesdits réseaux de neurones, en nous inspirant de la désormais obsolète architecture Lua de PyTorch.
Nous verrons dans ce rapport les explications de certains choix, les performances de nos implémentations ainsi que des protocoles expérimentaux que nous avons jugés intéressants d'effectuer.

Table des matières

1 Mise en route de l'implémentation

Avec ce rapport est bien sûr fourni le code-source du projet. Pour tester les fonctionnalités implémentées, un notebook contenant plusieurs expérimentations est à disposition. C'est avec ce dernier que nous avons réalisé nos tests qui seront décrits plus en profondeur dans la suite du rapport.

2 Contenu du projet

Le projet est séparé en plusieurs modules, et ces derniers possèdent tous des classes qui soit implémentent des couches de réseaux de neurones (forward et backward inclus), soit définissent des classes plus générales pour le traitement de problèmes à plus haut niveau. Les différents modules de notre bibliothèque pour réseaux de neurones sont les suivants :

- Module.py, contenant le module linéaire, le module de convolution en une dimension, et une classe qui implémente une régression linéaire à l'aide du module linéaire (avec les méthodes fit, predict, score et la spécification des epochs et du pas de gradient).
- Loss.py, qui implémente différentes loss telles que la mean squared error (MSE), la cross-entropie (CE), la cross-entropie binaire (BCE), la cross-entropie conjuguée avec un softmax passé au log (CElogSoftmax) et la KL-divergence (que nous avons choisi d'introduire car permet de donner une notion de distance entre des classes).
- Activation.py, qui définit plusieurs les fonctions d'activation utiles pour les réseaux de neurones (tangente hyperbolique, sigmoïde, softmax, maximum pooling en une dimension pour la convolution, ReLU et enfin flatten qui permet de dimensionner des données 3D en données 2D).
- Network.py, qui introduit les deux classes permettent de faire fonctionner tout ce beau monde ensemble ; la classe Sequentiel qui permet de construire le réseau de neurones avec tous ses modules ainsi que la généralisation des passes forward / backward, et la classe Optim qui implémente la méthode step() permettant de réaliser une epoch de descente de gradient tout en renvoyant les valeurs utiles à la représentation en graphiques des résultats.

De plus, deux modules plus annexes prennent part au bon fonctionnement de notre solution, à savoir le module preprocessing.py, utile pour rendre utilisables des datasets génériques (tels qu'on peut en trouver sur Kaggle par exemple), et le fameux module mltools.py fourni avec les TMEs dans le cadre de l'UE.

3 Expériences, performances

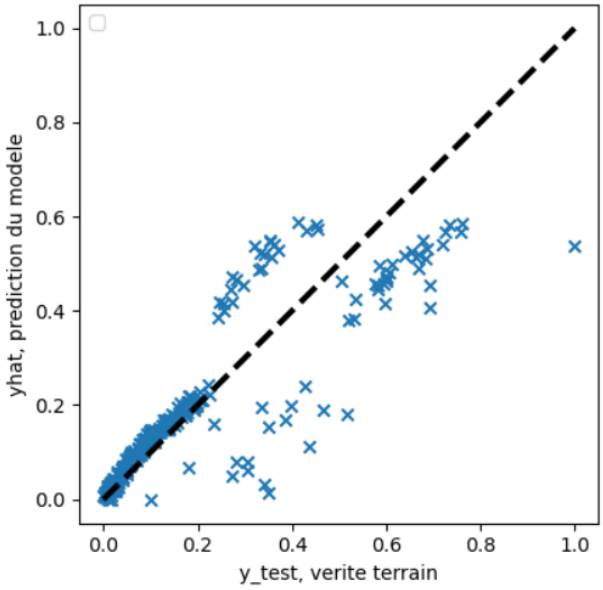
Toutes les expériences ont été réalisées avec une séparation train/validation/test répartie en 60/20/20 des données. Un biais est utilisé dans tous nos modules pour permettre la gestion du bruit, nous avons d'ailleurs estimé dans notre implémentation pertinent d'obliger l'utilisation du biais. Enfin, une corrélation de Pearson entre les courbes train/validation en loss et en accuracy/score sont toujours systématiquement calculés pour définir rigoureusement s'il y a overfitting ou underfitting, il en sera fait mention lorsque nécessaire (i.e. quand il ne sera pas clair que les deux courbes sont très similaires).

3.1 Performances génériques

Dans cette sous-section, nous allons simplement vous montrer rapidement quelques résultats attestant d'une bonne efficacité de notre solution sur des problèmes simples (ici des gaussiennes et de la régression linéaire).

3.1.1 Régression linéaire

Pour tester la régression linéaire, nous avons utilisé un dataset provenant de Kaggle (à l'adresse https://www.kaggle.com/datasets/mirichoi0218/insurance) dont nous avons voulu prédire des coûts médicaux selon différents critères. Après un preprocessing incluant normalisation des données numériques et encodage numérique des données autres, nous obtenons les résultats suivants :

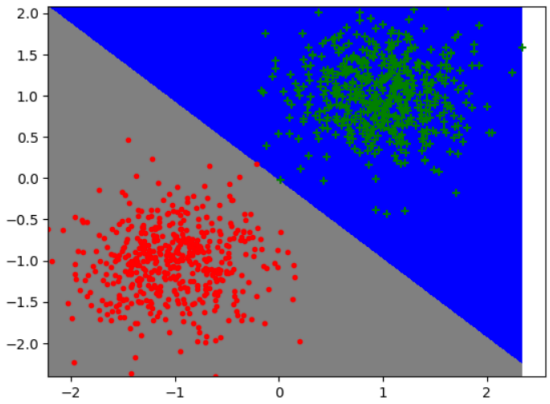


Plus numériquement, nous avons au final un R^2 d'environ 0.7 en apprentissage et 0.75 en test avec 300 epochs et un pas de gradient de 1e-4, qui se fait en temps négligeable.

3.1.2 Classification binaire de 2 gaussiennes

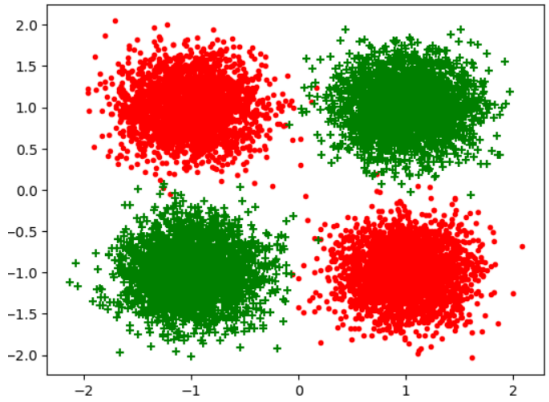
Dans ce problème, nous atteignons rapidement (4 epochs avec un pas de 1e-4) le score maximal (donc la loss minimale), et ce même si les données sont un peu bruitées.

En effet, il convient de préciser que notre module linéaire utilise obligatoirement un biais, ce qui permet une meilleure gestion de ce type de données. Ci-dessous, un aperçu de frontière de décision obtenable avec notre solution :



3.1.3 Classification de 4 gaussiennes non linéaires

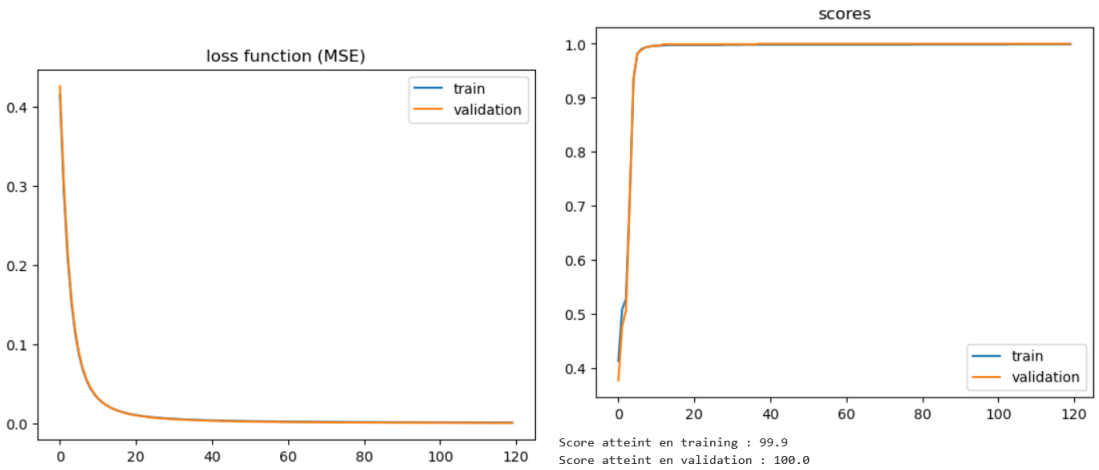
Le problème est que nous avons des données réparties en échiquier de deux classes comme suit :



Pour traiter ce genre de problèmes, un réseau de neurones s'avère nécessaire. Nous avons utilisé l'architecture suivante, tout en utilisant les classes nécessaires à l'encapsulage et à l'optimisation (ce qui permet de vérifier leur bon fonctionnement) :

```
Linear(2,16) -> TanH() -> Linear(16,1) -> Sigmoide()
```

Ce qui donne un nombre de poids à entraîner de 32 pour Linear(2,16) et 16 pour Linear(16,1).
Dans une boucle d'apprentissage à 120 epochs pour un pas de 1e-4, nous obtenons une accuracy en test autour de 99%.

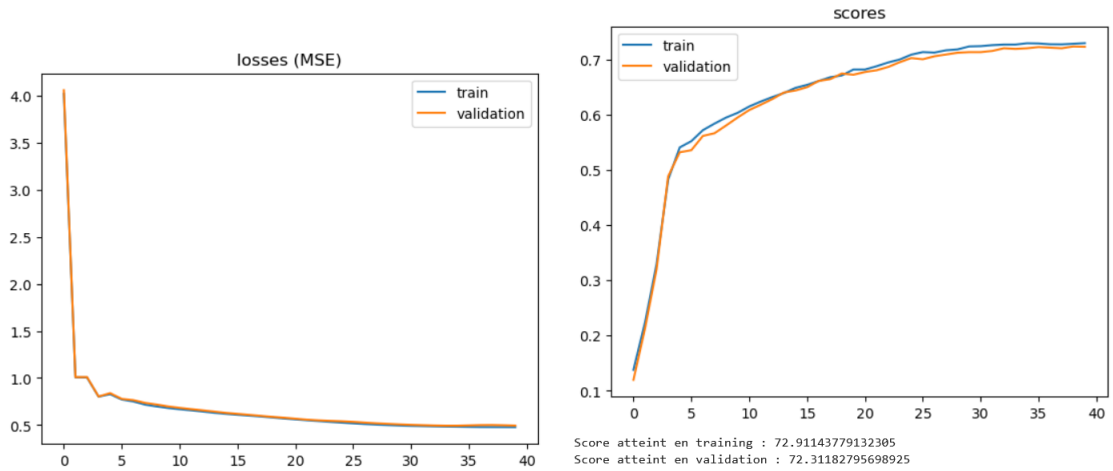


3.2 Performances en multi-classe

C'est ici que les choses sérieuses commencent. Nous utiliserons dans toute la suite le jeu de données USPS, qui contient des images de chiffres manuscrits de 0 à 9 (ce qui fait 10 classes de chiffres en tout).
Nous avons tout d'abord tenté une première approche en utilisant la MSE, qui devrait en théorie nous donner des résultats décevants car non adaptée à la classification (dont les sorties doivent tendance vers 0 ou 1, pas être une moyenne). Nous avons utilisé l'architecture suivante :

```
Linear(256,128) -> TanH() -> Linear(128,64) -> TanH() -> Linear(64,10) -> Sigmoide()
```

Ce qui nous fait un total (sans compter les biais) de 32768 poids à entraîner pour le premier module linéaire, 8192 pour le deuxième et 640 pour le troisième. Le module Sigmoide est mis en fin pour faire en sorte que les sorties soient comprises entre 0 et 1 (il convient de préciser que nous avons encodé en one-hot vector les labels des différentes images). Les résultats obtenus (avec un nombre d'epochs de 40 et un pas de 1e-3 et après 10 secondes d'apprentissage) sont les suivants :

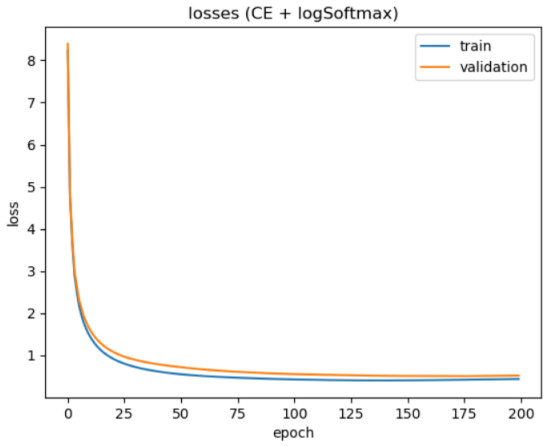


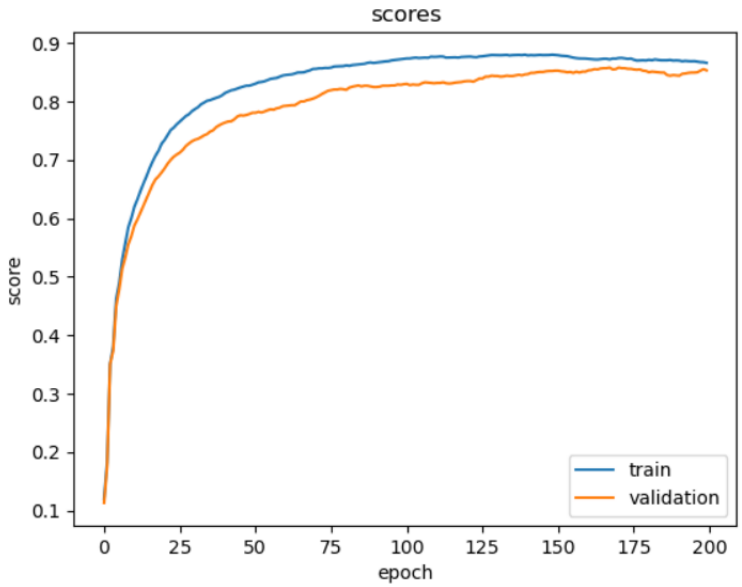
Ces résultats (relativement stables, toujours un score entre 65 et 75% en test) démontrent qu'il est possible de classer convenablement en multi-classe avec une MSE,

au moins dans le cadre défini par nos données.

Mais qu'en est-il avec la fonction de coût recommandée, à savoir une cross-entropie avec une fonction d'activation Softmax passée en logarithme ? Nous allons le voir maintenant.

En effet, en reprenant la même architecture de réseau de neurones à ceci près qu'on enlève l'activation Sigmoide et que la fonction coût fait elle-même cette cross-entropie avec le logSoftmax (ce qui fait que nous n'avons pas besoin de "normaliser" les sorties de la couche cachée Linear(100,10)), nous obtenons de meilleurs résultats (nombre d'epochs = 200, pas de gradient = 1e-4, temps d'apprentissage = 1min) :






```
Score (en %) atteint en training : 88.0064539261384
Score (en %) atteint en validation : 85.80645161290322
Score (en %) atteint sur les données de test : 83.54838709677419
```

```
print(pearson(losses_train, losses_valid))
```

```
0.9991495193800134
```

Nous remarquons donc que les scores sont un peu plus élevés y compris en test, qu'il n'y a que très peu d'overfitting (performances pratiquement équivalentes en train et en test), que les courbes de loss train/validation sont cohérentes entre elles (Pearson proche de 1), et enfin nous ajouterons que ces résultats sont beaucoup plus stables

qu'avec la MSE (toujours entre 83 et 86% d'accuracy en test).

La théorie s'est donc vue confirmée, la classification multi-classe se passe mieux avec une fonction cross-entropique couplée à un softmax passé au logarithme, y compris en gardant la même architecture de réseau de neurones.

3.3 Auto-encodeur

Nous voici dans la partie majeure du rapport, qui comprendra toutes les expérimentations et autres protocoles autour de l'auto-encodeur. En effet, ce type de réseau ouvrant à plein d'interrogations, nous nous sommes posés plusieurs questions que nous allons vous dévoiler et en prime ces questions s'accompagneront de réponses détaillées sur le sujet.

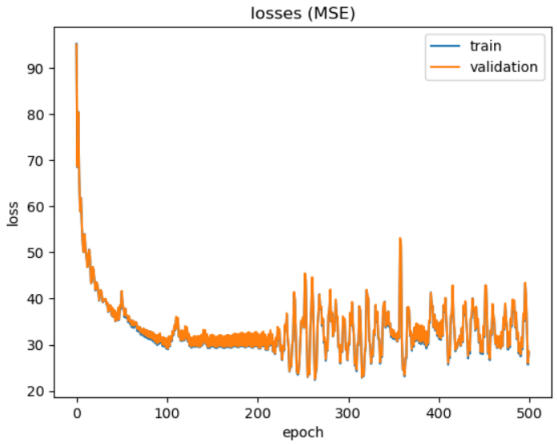
3.3.1 Premier test

Tout d'abord, nous allons vous montrer que notre implémentation fonctionne sur un exemple simple. Nous nous donnons le réseau suivant :

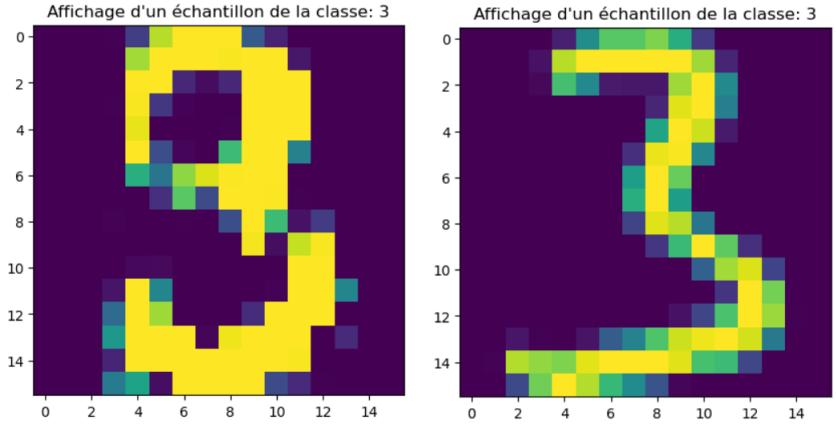
```
Linear(256,64) -> TanH() -> Linear(64,256) -> Sigmoide()
```

Nous avons utilisé une MSE pour la loss (qui s'est avérée plus intéressante que la BCE pour notre cas), et avons pris soin de normaliser les pixels entre 0 et 1 avant.

Avec un nombre d'epochs de 500 et un pas de 2e-4, nous obtenons la courbe d'évolution des loss suivanes :



Et voici un exemple de ce que peut donner cet auto-encodeur :

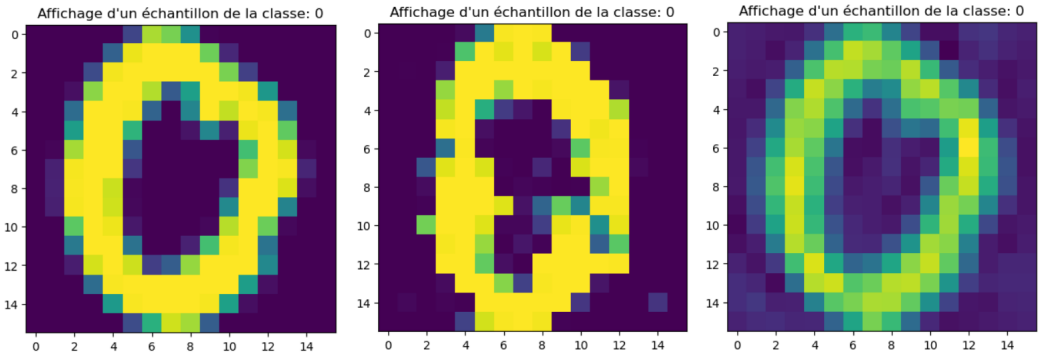


L'image à obtenir est à droite, celle que nous avons obtenue est à gauche. Bien que le tracé ne ressemble pas du tout à celui qu'il est censé être, nous pouvons cependant observer que le chiffre obtenu représente bel et bien un 3. Compte-tenu de la simplicité de l'auto-encodeur mis en place, nous pouvons en conclure que les auto-encodeurs sont fonctionnels dans notre code.

3.3.2 Encodeur/décodeur VS PCA

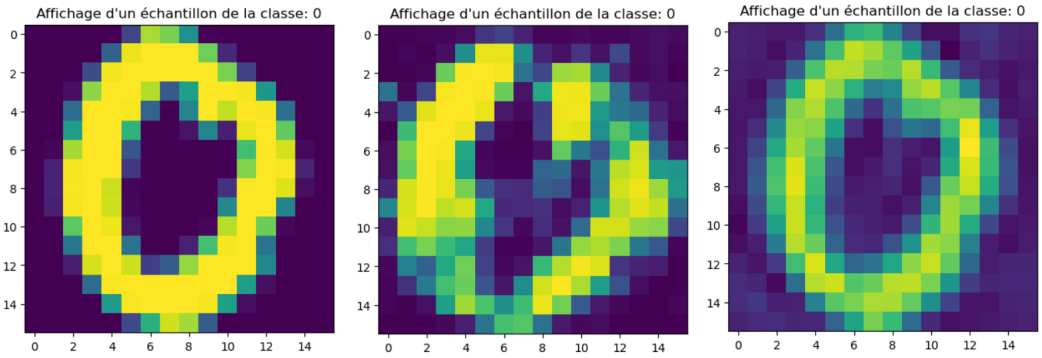
Dans cette sous-section, nous allons voir s'il est possible de substituer l'encodeur par celui de la PCA, idem pour le décodeur, et de manière plus générale de comparer les performances entre les différentes architectures que nous aurons obtenues. Nous garderons l'architecture décrite juste avant pour notre auto-encodeur.

Premièrement, comparons nos précédents résultats avec ceux obtenus par la PCA :



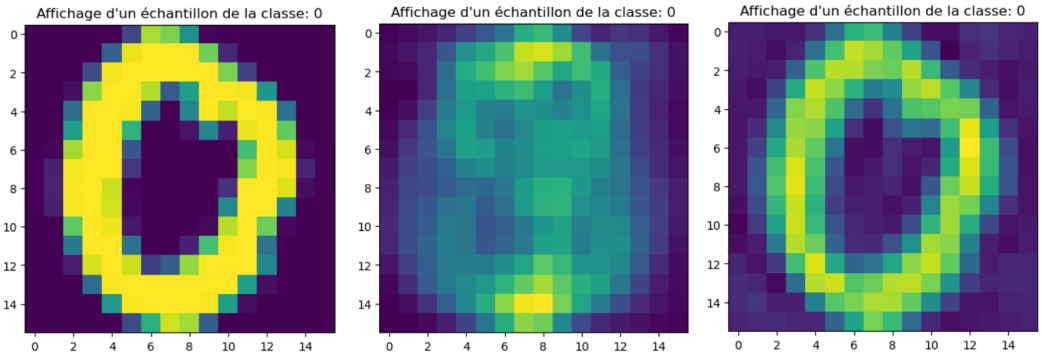
De gauche à droite : l'image originale, l'image obtenue avec auto-encodeur et enfin l'image obtenue par compression/décompression avec la PCA.
Nous remarquons alors plusieurs différences de traitement et plusieurs aspects notables. La PCA va avoir tendance à beaucoup moyenner les valeurs, ce qui fait que le fond noir ne l'est plus complètement et que les traits plus clairs sont moins marqués. Notre auto-encodeur, au contraire, va avoir tendance à mettre en valeur les traits marqués, quitte à donner moins de relief au nombre à prédire. Nous pouvons assigner ça au fait que nous n'avons pas régularisé notre résultat (pas de régularisation L1/L2 ou de matrice des poids transposée).

Maintenant, voyons voir ce qu'il en est lorsque nous utilisons la PCA comme encodeur puis un décodeur que nous avons façonné, ayant la même architecture que notre décodeur, en n'oubliant pas l'activation TanH au début :



Nous remarquons que l'image renvoyée par notre décodeur, au milieu, s'avère plus fiable et représentative que celle obtenue plus haut. Nous constatons cependant une petite différence au niveau de l'échelle, notre 0 éta,nt un peu plus écarté sur les côtés que l'original.
Cette expérience permet de se rendre compte que nous pouvons utiliser la PCA comme encodeur pour nos données, et que l'utiliser affecte positivement les performances de notre système.

Voyons voir si nous pouvons utiliser la PCA comme décodeur désormais, en gardant le même encodeur que celui décrit plus haut. Nous entraînerons notre encodeur sur les données compressées par la PCA en guise de modèle à reproduire. Voici nos résultats :



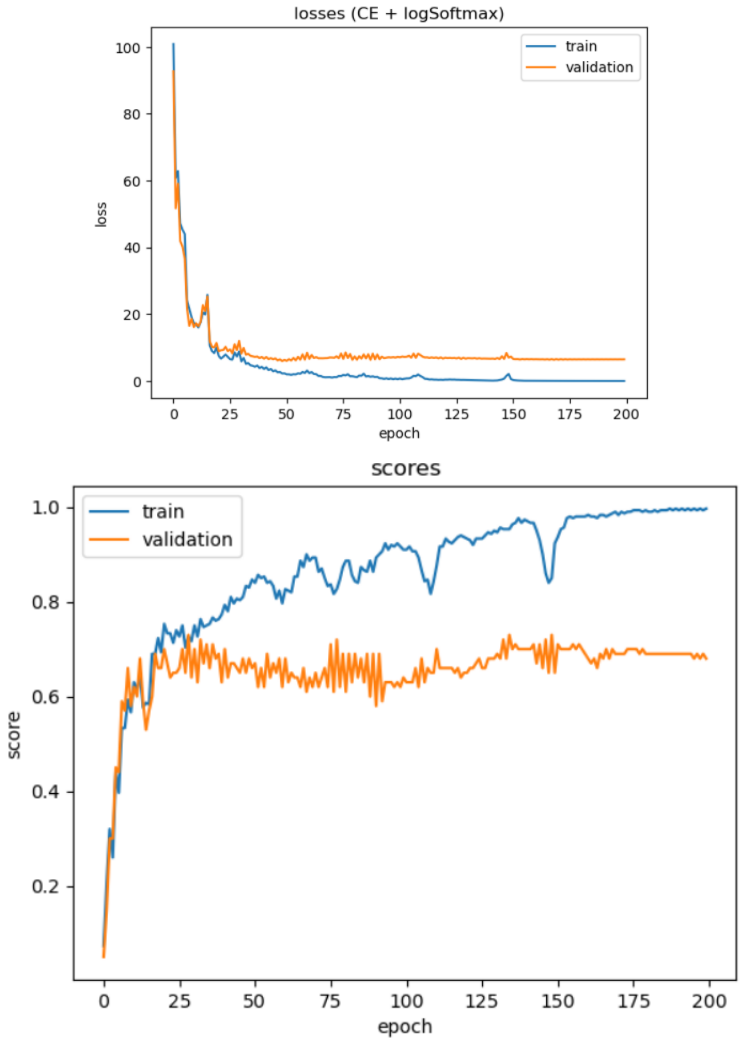
Nous remarquons alors que le résultat est beaucoup moins probant. Nous noterons par ailleurs la présence d'overfitting sur ce réseau... Nous avons essayé de le modifier en ajoutant des modules linéaires intermédiaires et des TanH, mais rien n'améliorait la sortie du réseau. Nous pensons que les résultats auraient pu être meilleurs si nous avions pu entraîner nos données sur des données compressées "officielles". Une solution plus discriminatoire (i.e. qui moyenne moins sur les valeurs des pixels) pourra être étudiée à l'avenir.

3.4 Convolution 1D

La conception du module de convolution aura nécessité divers tests et recherches dans le but d'optimiser un code naïf plus compréhensible mais nécessitant 3 boucles for sur plusieurs dimensions variables (le nombre de données, la dimension des données, le nombre de canaux de sortie, la valeur du stride).
Une solution viable aurait été d'utiliser numba pour écrire les boucles en C et ainsi optimiser le temps d'exécution, mais nous avons jugé la bibliothèque trop restrictive. Heureusement, la fonction einsum() de NumPy couplée à un "déplacement de fenêtre" nous permet de garantir un temps d'exécution convenable (5min pour un entraînement sur +9000 images taille de 16x16 sur 100 epochs).
Cependant, ce que nous avons choisi de mettre en lumière avec nos résultats ne nécessite pas beaucoup de données ; les résultats et expériences suivants sont réalisés avec 500 images.

3.4.1 Premier test et premières remarques

Tout d'abord, nous avons mis en place l'architecture préconisée par le sujet, couplée avec une cross-entropy loss avec une activation logSoftmax (en d'autres termes notre loss la plus efficace pour le multi-classe). Voici nos résultats, pour 200 epochs et un pas de gradient de 1e-4 :




```
Score (en %) atteint en training : 99.66666666666667
Score (en %) atteint en validation : 73.0
Score (en %) atteint sur les données de test : 74.0
```

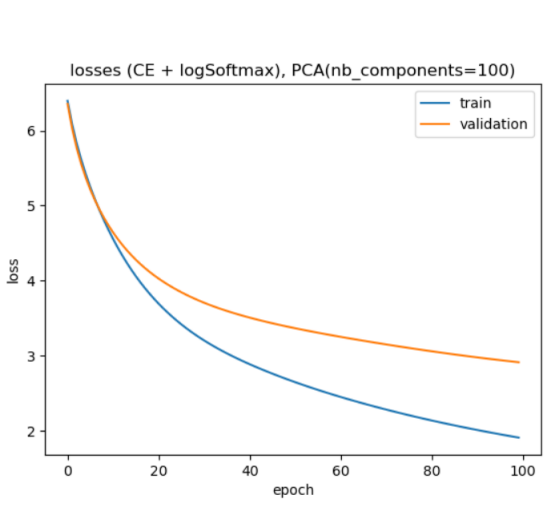
Comme nous pouvons le constater, ce réseau est plus expressif que ceux étudiés précédemment ; il est donc bien plus sujet à l'overfitting, et cela se ressent sur les courbes et sur les scores en accuracy (différence de plus de 25 points de pourcentage entre le training et la validation). Nous pensons que c'est dû au nombre faible de données utilisées.

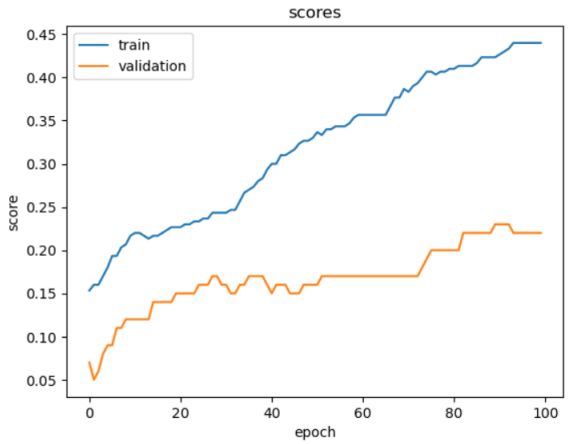
Dans notre cas, l'epoch optimale du modèle est à l'epoch 16, qui conserve des performances tout de même honorables bien que légèrement amoindries (autour de 70% d'accuracy en test pour environ 75 en apprentissage).

3.4.2 Autres architectures possibles et pistes d'amélioration

On l'a vu, ce réseau a une certaine propension à overfit sur notre faible jeu de données, et réduire le nombre d'epochs réduit aussi sa précision sur les jeux de tests.

Nous nous sommes d'abord tournés vers la fameuse PCA, pour compresser les images donc la dimensionalité et ainsi réduire le nombre de paramètres à apprendre, donc réduire la capacité du modèle. Les performances sont particulièrement exécrables, en voici un exemple :





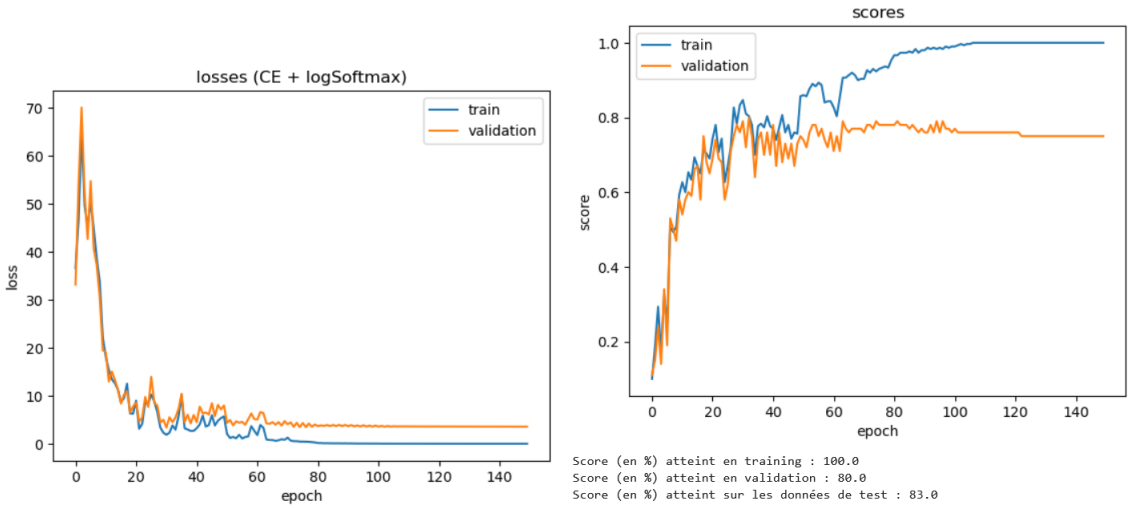

```
Score (en %) atteint en training : 44.0
Score (en %) atteint en validation : 23.0
Score (en %) atteint sur les données de test : 13.0
```

Ici, en réduisant les images à une dimension de 100 (la dimension maximum possible), nous remarquons que nous avons toujours ce souci d'overfitting (le gap entre score d'apprentissage et score de validation est toujours aussi important), mais qu'en plus un souci d'underfit est désormais présent. De plus, plus nous diminuons la dimension des photos, plus nous accentuons cet underfitting que nous ne pouvons pas réguler en augmentant le nombre d'epochs par exemple. Nous pouvons donc en déduire que le réseau doit avoir un minimum de capacité pour résoudre cette tâche de classification.

Pourtant, c'est bien en diminuant la capacité du réseau que nous arriverons à une conclusion intéressante... Peut-être faut-il réduire le nombre de modules ? Testons en optant pour un CNN minimisé par rapport à celui initialement préconisé :

```
Conv1D(3,1,32) -> MaxPool1D(2,2) -> Flatten() -> Linear(4064,10)
```

Ce qui réduit de 90% la partie linéaire du réseau (on passe de 407400 poids à 40640). Avec un nombre d'epochs de 150 pour un pas de 0.001, nous obtenons les résultats suivants :



Nous observons donc une amélioration assez nette sur les performances du réseau, ce qui semble logique puisque la réduction du nombre de poids permet une meilleure généralibilité du modèle, encore plus sur un jeu de 500 données au total.
D'un point de vue comparatif, nous nous rapprochons des performances de notre réseau non linéaire multi-classe, bien que soyions encore légèrement en-dessous (le score sur les données de test oscille entre 78 et 84% pour le CNN). En théorie, nous devrions arriver à dépasser ce réseau défini précédemment d'un point de vue performances. Continuons sur notre lancée.

On l'a vu, réduire le nombre de paramètres à apprendre semble être une bonne piste. Pour aller plus loin dans cette voie, nous pouvons rappeler que les paramètres du module de convolution et de maximum pooling permettent de définir la taille des fenêtres à étudier, donc des paramètres à apprendre. Nous avons essayé plusieurs possibilités pour réduire la matrice de poids (augmenter le kernel_size dans le module de convolution ou de max pooling, augmenter le stride du MaxPool, diminuer le nombre de canaux sortants...). La meilleure solution, qui s'avère aussi être la plus gourmande (1 epoch par seconde pour une dimension flattened de 2048, les autres sont plutôt à 4/5), est celle où l'on augmente le kernel_size dans le module de convolution en premier lieu. Nous noterons par ailleurs qu'augmenter le stride du max pooling a tendance à bruiter les courbes de loss/score en plus de baisser les performances, et que jouer sur le nombre de canaux sortants ne semble pas avoir de grand impact, tout comme le kernel_size du max pooling.

En tâtonnant et en réalisant plusieurs suites d'expériences, nous n'avons finalement pas trouvé de CNN suffisamment stable pour concurrencer celui trouvé plus haut. Il faut par ailleurs préciser que la performance dudit CNN est très intéressante pour un jeu de données si petit, nous nous en contenterons donc.

4 Conclusion

Au terme de ce projet, force est de constater que nous avons plusieurs choses à dire concernant nos difficultés à réaliser certaines tâches, et les points d'amélioration possibles de notre solution logicielle.
Nous tenons néanmoins à préciser que ce projet est une très bonne idée pour nous préparer en amont au M2 et à utiliser PyTorch convenablement lors des UEs dédiées (AMAL et RLD). Cela nous permet également de voir ce qu'il y a dans la boîte noire, de sorte à ne pas être un simple exécutif (ou encore le fameux "presta") sans profondeur ni capacité de réflexion en entreprise, et à nous préparer au mieux à une thèse dans l'autre cas principal d'orientation après le master.

4.1 Difficultés rencontrées

La première difficulté qu'il a fallu gérer était les risques d'overflow. En effet, nous nous y sommes retrouvés confrontés à pratiquement toutes les expérimentations un tant soit peu gourmandes. Pour les régler, il suffit de seuiller les valeurs absolues trop grandes.
Ensuite, nous avons évidemment un peu tâtonné sur le module de convolution, car nous ne pouvions pas nous permettre de laisser un code à 3 boucles for qui mettait plusieurs dizaines de minutes si ce n'est pire lorsqu'on lui passait un nombre convenable de données... le module d'Einstein sum de Numpy nous a ici été d'une grande aide, la seule difficulté subsidiaire qu'il restait était de bien avoir en tête les dimensions des matrices en entrée et en sortie de la fonction. Malgré cette optimisation, le module de convolution est celui qui met le plus de temps à finir son processus (si on rapporte ce temps au nombre de paramètres qui lui sont associés).
Enfin, l'autoencodeur est la partie qui nous aura mis le plus en bâtons dans les roues. En effet, utiliser la BCE n'a pas été bénéfique pour nos résultats, ce qui nous a fait tergiverser quelques temps étant donné que le sujet stipulait à juste titre qu'elle est plus performante que la MSE pour ce genre de tâches. Pour nous, la MSE s'est avérée plus fiable bien qu'imparfaite (les images retrouvées ne sont pas vraiment identiques aux originales).
Le reste des difficultés retrouvées sont inhérentes aux problèmes basiques du machine learner : fine-tuner un modèle, tâtonner sur les hyper-paramètres, changer les preprocessings... Et on se rend vite compte qu'au final la qualité et le pre-processing des données importe énormément, ce qui confirme les clichés sur les data scientists (90% du temps est passé sur le nettoyage et le pre-processing des données, 10% sur le reste...).

4.2 Points d'amélioration

Comme évoqué précédemment, nous pourrons dans le futur améliorer l'autoencodeur de notre système, que nous jugeons insuffisant étant donné ce qui peut se faire à l'état de l'art aujourd'hui. Ensuite, voici une liste de choses nouvelles à implémenter dans notre programme :

- Convolution 2D, average pooling pour la passe de convolution ;
- Implémentation d'autres fonctions de loss qui pourraient s'avérer utiles (nous avons déjà implémenté nous-mêmes la KL-divergence) ;

— Passage des calculs sur le GPU, possible avec pycuda ou numba (même si ce dernier est de plus en plus obsolète et inutilisé) si l'on ne veut VRAIMENT pas passer sur PyTorch ou TensorFlow ;
— Recoder tout le module en C pourrait être une bonne solution d'optimisation également ;
— Notre code n'est pas généralisé, repenser la structure de certains modules pourrait être bénéfique (surtout si l'on veut se rapprocher de l'ergonomie proposée par Pytorch) ;
— Toujours d'un point de vue pratique, automatiser l'optimisation des hyperparamètres pourrait s'avérer très utile ;
— implémenter des descentes de gradient plus rapides telles que la descente de gradient stochastique, en mini-batch ou en batch ;
— enfin, implémenter des outils pour régulariser les sorties des modules (L1 / L2 / matrice transposée des poids pour l'auto-encodeur).