Analyse CNN

Ismat Benotsmane

Novembre 2020

Table des matières

1 Réseux convolutionnels pour l'image

1.1 Introduction au CNN

(Q1) Si on considére un seul filtre de convolution de padding p, de stride s et de taille de kernel k pour une entrée de taille $x \times y \times z$, on a une sortie de taille :

$$\left(\left[\frac{x+2p-k}{s}\right]+1\right) * \left(\left[\frac{y+2p-k}{s}\right]+1\right)$$

Le nombre de poids à apprendre pour cette couche est $(k*k*z)+1$.
Avec la formule générale, $dim = (kernel*kernel*channel\_in*channel\_out)+channel\_out$

Dans le cas d'un réseau fully-connected, on devrait apprendre :

$$x*y*z*((x+2p-k)/s\ +1)*((y+2p-k)/s\ +1)*channel\_out$$

Avec ici channel_out = 1.

(Q2) Les principaux avantages de la convolution par rapport aux réseaux fully-connected c'est le nombre de parametres à apprendre qui est trés inferieur, et le fait de préserver la localité de l'information, c'est-à-dire qu'un pixel sera traité dans un contexte par rapport aux pixels qui l'entoure contrairement au MLP qui traite les pixels de façon indépendante.
L'autre avantage aussi, c'est d'être invariant à la translation, c'est-à-dire que si on a deux images qui ont un même motif localisé à deux endroits différents, elle seront considérées plus similaire.

Les inconvenients c'est cette localité de l'information qui entraine une perte d'information car on pourra pas détécter de grands patterns. Il y a également le temps d'apprentissage du réseau.

(Q3) L'intêret du pooling spatial est qu'on réduit la dimension de l'image en résumant l'information, ce qui a pour effet de réduire la complexité des calculs et de rendre le réseau plus robuste à la translation.

(Q4) On peut appliquer les couches de convolutions sur des images de taille différentes en entrée. De même, pour les couches d'activation et de pooling.
Le probléme va se situer au niveau des couches fully-connected, car la taille des sorties des couches de convolution seront plus grandes, et une fois qu'on arrive à la derniére couche de convolution on aura une matrice plus grande et donc incompatible avec le nombre de poids de la couche fully-connected.
Pour contourner ce problème, on peut utiliser un **Global Pooling**.

(Q5) On peut voir les couches fully-connected comme des convolutions particulières de taille de kernel k égale à la taille de la feature map en entrée, et le nombre de filtre C égale au nombre de neurones en sortie de la couche fully-connected.

(Q6) Si toutes les couches de notre réseau sont des couches de convolutions alors on peut prendre en entrée des images de tailles différentes.

La sortie du réseau dépend de la taille d'image en entrée, si cette derniére est grande alors on a une sortie en 2D où chaque élément est une classification sur une sous partie de l'image.

(Q7) La taille des receptive fields des neurones de la première couche correspondent à la taille du kernel $r_1 = k_1$, en conséquent la taille pour une deuxiéme couche est $r_2 = (k_2 - 1) * s_1 + r_1$.

Donc plus on avancera dans les couches plus on aura une vue d'ensemble sur toute l'image, dans les premières couches on va détecter des features bas niveaux comme des contours par exemple, et dans les dernières couches des features haut niveaux avec des formes plus complex telle qu'une tête par exemple.

1.2 Apprentissage from scratch du modèle

(Q8) Si on veut conserver les mêmes dimensions passée en entrée aprés une convolution, on applique une stride de 1 et un padding en fonction du kernel k :

- pour k impair, $p = \frac{k-1}{2}$.
- pour k pair, $p = \frac{k-1}{2}$ sauf qu'on va se retrouver avec une valeur non entière, pour cette raison il est préférable d'utiliser un kernel de taille impair pour pouvoir avoir des sorties de même taille qu'en entrée.

(Q9) Pour réduire les dimensions spatiales d'un facteur 2 avec le max pooling, il faut utiliser un kernel de 2, un padding de 0 et stride de 2.

(Q10)

Couche	Taille sortie	Nombre poids	Cumul nombre poids
Entrée	$32 \times 32 \times 3$	0	0
Conv1	$32 \times 32 \times 32$	$5 \times 5 \times 3 \times 32 + 32$	2432
Pool1	$16 \times 16 \times 32$	0	2432
Conv2	$16 \times 16 \times 64$	$5 \times 5 \times 32 \times 64 + 64$	53696
Pool2	$8 \times 8 \times 64$	0	53696
Conv3	$8 \times 8 \times 64$	$5 \times 5 \times 64 \times 64 + 64$	156160
Pool3	$4 \times 4 \times 64$	0	156160
fc4	1000	$4 \times 4 \times 64 \times 1000$	1180160
fc5	10	1000×10	1190160

Plus on avance dans le réseau, plus le nombre de poids augmente de façon conséquente. Notamment les couches fully-connected qui concentrent la majorité des poids du réseau.

(Q11) On a un nombre total de 1190160 poids à apprendre. Comparé au nombre d'exemple qu'on a en train qui est de 50000 images, cela semble beaucoup pour peu de data ce qui va provoquer un sur-apprentissage.

(Q12) Dans l'approche **BOW/SVM** si on prend 1000 descripteur SIFT et 10 classes, alors on a 1000+1 parametres par SVM et comme on considère 10 SVM(un par classe) cela nous fait 10010 parametres à apprendre.
On a donc 100 fois moins de paramètres à apprendre.

(Q14) En tain on doit faire une backpropagation pour corriger les paramètres, d'où la présence des fonctions *backward* et *step*(). En test, le mode *eval*() permet d'éviter la mise à jour des paramétres, de désactiver les couches dropout ou batch-norm.
De plus, en train on affiche l'erreur moyenne sur les batchs au fur à mesure qu'on itére.
On a remarqué que l'accuracy de train était bien inférieur à celle de test dans les premières epoch, ceci peut être expliquer par le fait que l'accuracy en train est calculé *online* pendant que le réseau apprend, hors dans les premières itérations de batch on est mauvais, et à la fin de l'epoch on est meilleur. En test l'accuracy est calculé avec les poids mis à jour par le dernier batch, c'est pourquoi on est meilleur qu'en train.

(Q16) Si le pas de gradient est trop petit on converge trés lentement ou on reste bloquer dans un minima local, s'il est trés grand on risque de divergé ou ne pas atteindre le minima local.

Un mini-batch trop grand, donne une loss plus précise et stable, mais l'incovenient c'est le temps de calcul trop lent voire les ressources hardware si on a beaucoup de données surtout pour les images c'est juste impossible à charger sur un GPU.
Un batch trop petit, on a un temps de calcul trés rapide mais une loss instable et bruité.

Généralement on travaille avec des mini-batchs de 64, 128, 256 voire 512, cela dépend évidemment du matriel qu'on posséde. Et on dit que si on augmente la taille du batch $\times 2$ on augmente le learning rate $\times 2$.

(Q17) L'erreur au début de la première époque correspond au fait que notre réseau a été initialisé avec des poids aléatoire. Cette erreur doit correspondre à une accuracy plus au moins de 10% car on a 10 classes et on tire aléatoirement. Ça sera notre baseline au début, et ainsi l'erreur devrait diminuer.

(Q18) Nous avons entraîné notre réseau avec un learning rate de **0.1**, un batch size de **128**.

En train, dans la figure 1 on constate qu'après 30 epoch la loss a bien convergé vers 0, car on a une loss de 0.1. On a bien réussi à entraîner le réseau.

À la figure 2, on observe que les loss en train et en test baisse au début, mais vers la 10ème epoch la loss en train continue à baisser mais celle de test augmente. On est dans un phénomène de sur-apprentissage, cela est confirmer par l'accuracy en train qui est de 100% et en test qui stagne à 70%. Même si l'accuracy en test stagne, on se doit de corriger ce problème.

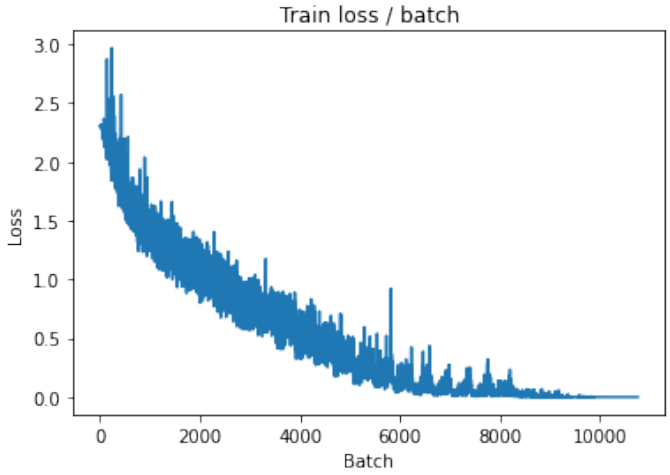

FIGURE 1 – Loss par batch en Train

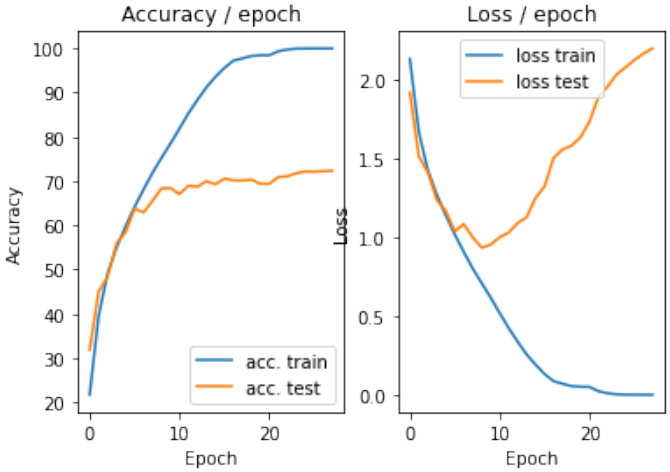

FIGURE 2 – Accuracy et loss en train/test

1.3 Améliorations des résultats

(Q19) Avec les mêmes hyperparamétres que l'étape d'avant, on a pu observer que la convergence de la loss de train et test est beaucoup plus rapide qu'avant. On peut noter que la loss minimale en test a également baisser que l'étape précédente ce qui a eu pour effet d'augmenter l'accuracy en test qui est passé de 70% à 75%.

On remarque toute fois qu'on est encore présence d'un sur-apprentissage.

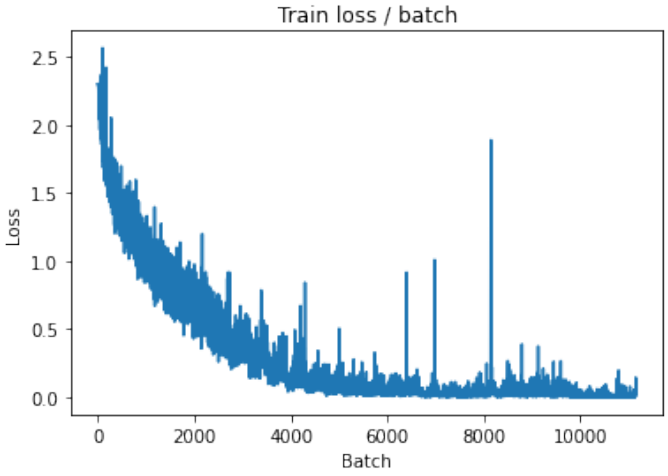

FIGURE 3 – Loss par batch en Train avec normalisation

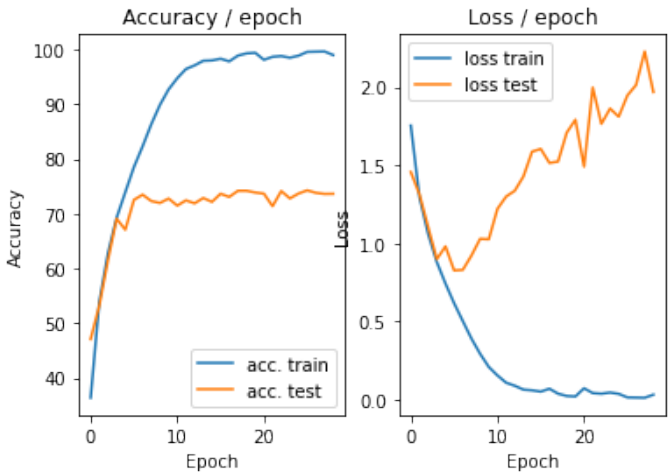

FIGURE 4 – Accuracy et loss en train/test avec normalisation

(Q20) Pour ne pas biaiser les résultats de notre modèle on calcule l'image moyenne que sur l'apprentissage, et on normalise les images de validation avec cette image. On procède ainsi car on ne connaît pas réellement les images tests en réalité.

(Q22) Après l'application d'une data augmentation avec un crop aléatoire de taille 28×28 et une symétrie horizontale de probabilité $p = 0.5$, on remarque que le sur-apprentissage arrive plus tardivement et il remonte plus lentement comparé à l'étape précédente.
On relève qu'on a amélioré la loss en test a été amélioré que précédemment, ce qui a eu pour effet d'améliorer l'accuracy en test qui est à 80%.
On peut souligner aussi qu'on a perdu légèrement en accuracy train pour gagner en plus en test.

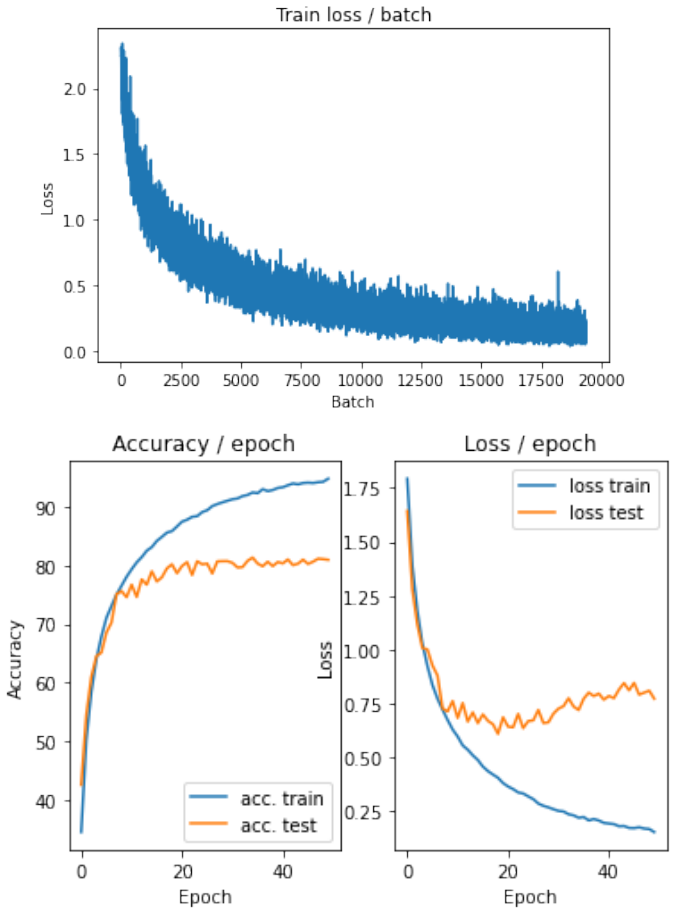

FIGURE 5 – Accuracy et loss en train/test après data augmentation

(Q23) Cette approche de symétrie horizontale elle ne peut pas être utilisable sur tout type d'images. Car par exemple, dans le cas de chiffres ou des lettres on en perd la sémantique.
Tandis que d'autres image comme les images satellites ou un verre renversé sont sémantique invariante par symétrie. Cela peut même participer à une robustesse et une généralisation du modèle.

(Q24) La limite de ce genre de data augmentation c'est comme on l'a mentionné ci-dessus, la perte de sémantique, la génération de pettern qui n'ont aucun sens comme un chien pivoté de 90° ce qui a pour effet de participer au phénomène de sur-apprentissage et donc un modèle moins précis.

(Q25) Il existe d'autres méthodes de data augmentation comme changement d'intensité de l'image, de saturation. On peut ajouter du bruit ou faire des rotations de différents angles, ou la suppression partielle de partie de l'image.

(Q26) Pour améliorer l'optimisation, on a introduit un momentum de 0.9 et un learning rate scheduler avec une décroissance exponentielle de coefficient 0.95.
On observe que l'évolution des différentes courbes que ça soit de loss ou d'accuracy se font de manière plus *smooth* et stable qu'auparavant mais en contre partie cela exige plus de temps d'entraînement. D'ailleurs, on remarque qu'il y a un décalage du sur-apprentissage à la 40ème epoch.

Cependant, on observe aucune amélioration de résultats comparé à précédemment.

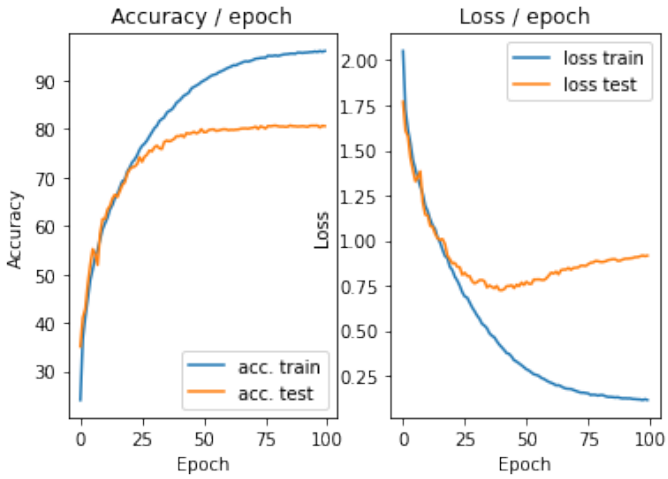

FIGURE 6 – Accuracy et loss en train/test après ajout de momemtum et LR Scheduler

(Q27) Le SGD avec un momentum permet d'accélérer la convergence et réduire les oscillations quand les données varient différemment selon les axes. Il conserve la mise à jour effectuer au pas précédent et il la pondère avec une valeur $\gamma < 1$.
Quand la nouvelle mise à jour est dans la même direction que la précédente on accélère la descente, dans le cas contraire on diminue la descente.

Le learning rate scheldurer est un pas d'apprentissage adaptatif. On l'utilise par exemple pour démarrer avec un grand pas de gradient pour s'extraire de minima locaux et le réduire au fur à mesure de l'apprentissage pour converger au mieux vers un minimum global.

(Q28) Il existe plusieurs variantes de SGD, comme **Adam** qui se base sur la moyenne et la variance pour adapter le pas de gradient ou encore **Adagrad**. On peut également cité **Nadam** ou **Adadelta**.

Il existe aussi d'autres stratégies de plannification du pas de gradient comme **StepLR** ou **MultiStepLR**.

(Q29) La première couche linéaire du fully-connected contient beaucoup de paramètres ce qui peut provoquer un sur-apprentissage de cette dernière.
En ajoutant une régularisation par *dropout* à la sortie de cette couche avec un probabilité $p = 0.5$ de désactiver un neurone, on voit qu'après 150 epoch il n'y a toujours pas de sur-apprentissage. Cela est confirmer aussi par l'accuracy en train qui a chuté de 10% comparé à l'étape précédente.
Par contre en test, on observe aucune amélioration significative.

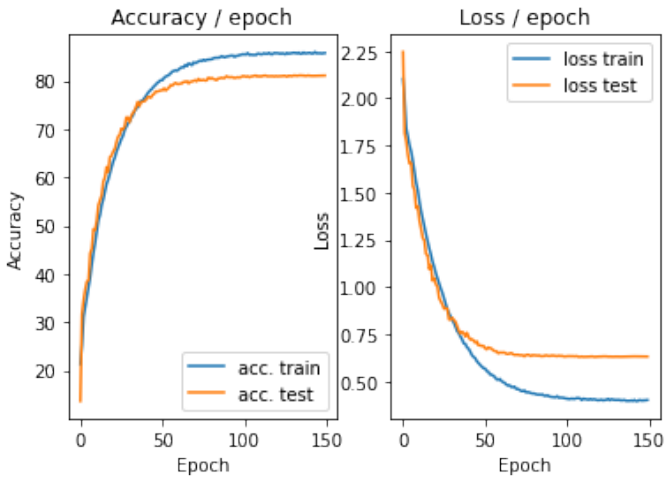

FIGURE 7 – Accuracy et loss en train/test après ajout de dropout

(Q30) La régularisation est un processus qui consiste à ajouter de l'information à un problème dans le but d'éviter le sur-apprentissage et une meilleur généralisation en test. Cette information se traduit sous une forme de pénalité envers la complexité du modèle.

(Q31) La technique du dropout permet de généraliser mieux les performances des neurones du réseau et ainsi éviter le sur-apprentissage. Le fait d'éteindre aléatoirement certains neurones oblige les autres neurones encore actifs à travailler plus. Ce qui va leur permettre d'avoir un impact sur la décision et éviter que tout le réseau soit exprimer que par quelque noeud.

(Q32) L'hyperparametre du dropout, c'est tout simplement la probabilité p de désactiver ou non un neurone. Si p est grand on risque de sous-apprendre, et si p est petit alors le dropout n'apporte aucune valeur ajouter et on risque potentiellement de sur-apprendre.

Dans le papier de recherche mis en référence, il est indiqué que p optimal est proche de 0.5, et dans le cas où on a des couches avec un nombre de neurones identiques alors $0.4 < p < 0.8$.

(Q33) En mode évaluation, le dropout est désactivé. Pour faire en sorte que l'intensité de la sortie soit proportionnelle au nombre de neurones non désactivés, on multiplie les sorties du réseau par le coefficient du dropout pour égaler l'espérance d'intensité du mode train.

(Q34) Le principe de la batch normalisation, c'est de normaliser chaque image non pas en fonction de tout le dataset mais en fonction des images contenu dans le même batch size qu'elle.
On remarque que la data normalisation a eu pour effet d'accélérer la convergence de la loss sans toute fois entrée en sur-apprentissage. Ça a permis également d'atteindre une accuracy optimal plus rapidement que ce qu'on a vu auparavant.

Quant à la performance, on a augmenté l'accuracy de 5% passant de 80% à 85%.

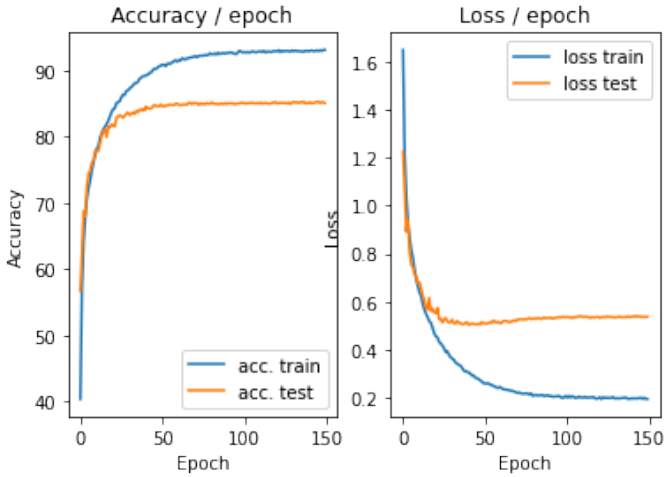

FIGURE 8 – Accuracy et loss en train/test après ajout de Data Normalisation

1.4 Optimisation des hyperparamètres

Pour l'optimisation nous avons considéré trois hyperparamétres, le **learning rate**, le **batch size** et le **ratio de dropout**.

```
    min_loss = 1e+10
    epoch_min_loss=1
    count_epochs = 0       # if after strait 10 epochs the test loss is growing, we stop learning
    # On itère sur les epochs
    for i in range(epochs):
        print("==================\n=== EPOCH "+str(i+1)+" =====\n==================\n")
        # Phase de train
        top1_acc, avg_top5_acc, loss = epoch(train, model, criterion, optimizer, cuda)
        # Phase d'evaluation
        top1_acc_test, top5_acc_test, loss_test = epoch(test, model, criterion, cuda=cuda)
        if loss_test.val < min_loss:
          min_loss = loss_test.val
          epoch_min_loss = i
          count_epochs = 0
        else:
          count_epochs +=1
          if count_epochs == 10:
            break
        # plot
        plot.update(loss.avg, loss_test.avg, top1_acc.avg, top1_acc_test.avg)

        lr_sched.step()

    return min_loss, epoch_min_loss
```

```
lr_grid = [1.0, 0.1, 0.01, 0.001]
  batch_size_grid = [64, 128, 256, 512]
  p_dropout_grid = np.arange(0.2,1,0.1)
  min_loss = 1e+10
  best_params={'learning_rate':0, 'batch_size':0, 'p_dropout':0, 'epoch':0}

  for lr in lr_grid:
    for bs in batch_size_grid:
      for p in p_dropout_grid:
        loss, min_epoch = main(bs, lr, epochs=150, proba_dropout=p, cuda=True)
        if loss < min_loss:
          min_loss = loss
          best_params['learning_rate'] = lr
          best_params['epoch'] = min_epoch
          best_params['batch_size'] = bs
          best_params['p_dropout'] = p
```

FIGURE 9 – Code du grid search

Les hyperparamètres optimaux sont : **learning rate**=0.01, **batch size**=512, **ratio dropout**=0.6, **epoch**=52.