École Centrale Lyon

MOD 3.2
TD1
Rapport

Kppv et réseaux de neurones pour la classification d'images

Élève :
Maxime Clément

Enseignant :
Alberto Bosio

21 décembre 2021

Table des matières

Introduction

Ce TD à pour objectif de mettre en place deux méthodes de classification d'images sans utiliser de bibliothèques dédiées. L'approche de classification par l'algorithme de K plus proches voisins sera traitée dans la partie 1 alors que la partie 2 traitera de l'approche par un réseau de neurones dit "fully connected". On étudiera en particulier les limites et avantages de chaque techniques.

Présentation des données pour les expérimentations

Les données utilisées pour ce TD sont les images de la base **CIFAR-10**. Il s'agit d'une base de 60000 images RGB de taille 32x32 réparties en 10 classes. Toute image possède un label lui étant associé. Ces données ainsi que les instructions pour les utiliser sont disponibles à l'adresse : https ://www.cs.toronto.edu/ kriz/cifar.html.

Lecture et traitement des données

Les données fournis sont contenues dans 6 fichiers batch différents. Chaque fichier correspond à la sérialisation d'un dictionnaire de 10000 images accompagnées de leur labels et de leurs noms. Les fonctions dédiées à la lecture des images (A.1), leur représentation par des descripteurs (A.4, A.3) et le découpage aléatoire en deux sous ensemble d'apprentissage et de test (A.2) sont disponible dans le fichier prepareData.py.

On note en particulier que quoi qu'il arrive les données sont représentées sous forme de matrice, pour lesquelles chaque ligne correspond au vecteur descripteur d'une image. Par exemple, sans conversion une image en RGB est représentée un vecteur ligne de 3072 valeurs flottante.

Ces fonctions seront utilisées pour les deux parties suivante (1 et 2)

On définie aussi la fonction *evaluation_ classifieur* (D.1) prenant un vecteur de labels prédit et le vecteur des labels de référence afin de calculer la précision (accuracy) de la prédiction.

1 Système de classification à base de l'algorithme des kppv

L'objectif pour cette partie est d'utiliser la puissance des calculs matriciels pour limiter au maximum le temps d'exécution. On doit donc restreindre au plus l'utilisation de boucles.

1.1 Calcul des distances

Soit deux matrices A et B de dimensions respectives (a,n) et (b,n). On souhaite obtenir la matrice des distance D de dimension(a,b) tel que Dij corresponde à la norme L2 de la différence entre les vecteurs Ai et Bj. Pour ce faire on utilise la fonctionnalité de broadcasting des matrices numpy. En effet numpy permet de "répéter" les matrices selon l'un des axes pour faire correspondre les dimensions de matrices lors de certaines opérations (C'est par exemple le cas lorsque l'on multiplie une matrice par une constante). Ici on

souhaite passer par l'intermédiaire d'une matrice de dimensions (a,b,n) correspondant à la différence terme à terme de chaque élément des matrices A et B. (Voir Figure 1). On peut alors sommer les éléments de cette matrice selon l'axe 2 pour obtenir la matrice des normes L2 au carré. En appliquant une racine carré sur chaque terme on obtient enfin la matrice D voulue initialement.

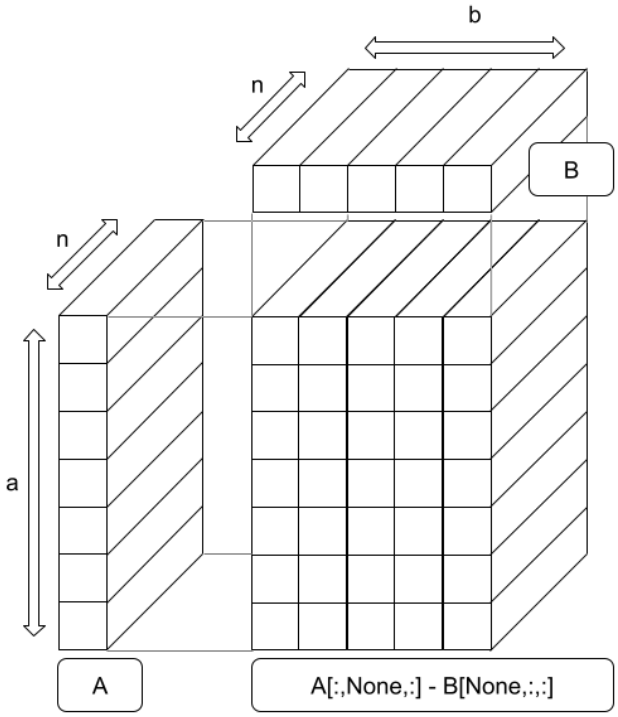

FIGURE 1 – Différence terme à terme des matrices A et B. Le vecteur en (i,j) de cette matrice correspond au vecteur Ai - Bj.
A est répétée selon l'axe 1 (horizontal) alors que B est répétée selon l'axe 0 (vertical)

En pratique le nombre de donnée est bien trop important pour pouvoir être réalisé en une seule fois. En effet même si on ne prend en compte que 10000 images séparées en 8000 images d'apprentissage et 2000 images de test, la matrice intermédiaire aurait un dimension de 8000x2000x3072, soit environ 180Go. On procède donc plutôt par batch pour calculer la matrice D en plusieurs fois. On se fixe une taille t de batch correspondant aux nombre de vecteurs images pris en compte en même temps pour calculer D (Voir Figure 2), ceci permet alors de calculer une portion carré de taille $t*t$ de la matrice D. Il est alors facile à l'aide d'une boucle de traiter l'ensemble des données.

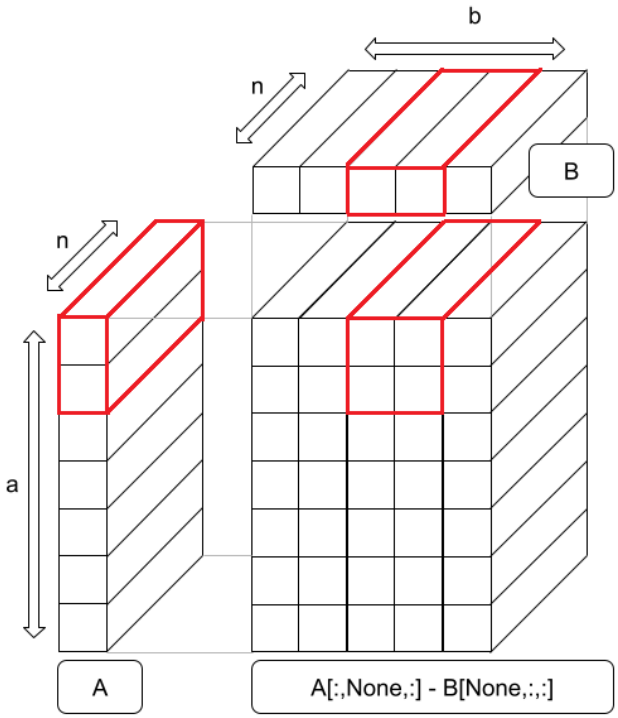

FIGURE 2 – Illustration de l'utilisation de batchs pour calculer la matrice des distances D. (Voir Figure 1)

1.2 Prédiction des labels à partir de la matrice des distances

Une fois la matrice des distance calculé (Voir Partie 1.1), il suffit de trouver les k plus proches voisins de chaque donnée de test pour obtenir le vecteur des labels prédits. C'est le rôle de la fonction *kppv_predict* (Voir B.2). On utilise la fonction *argpartition* sur la matrice D selon l'axe 1 permettant d'arranger les indices de chaque ligne de façons à ce que le k premières valeurs correspondant à ces indices soient les plus petites de la ligne. Pour chaque ligne on ne récupère alors que les k plus petites valeurs. En calculant l'histogramme entre 1 et 10 pour chacune de ces lignes on peut alors facilement retrouver le label majoritaire parmi les k plus proches voisins. Cette fonction renvoie alors un vecteur correspondant aux labels prédits par la méthodes des k plus proches voisins.

On peut alors aisément évaluer la prédiction en faisant appel à la fonction *evaluation_classifier* (D.1).

1.3 Expérimentation

Dans un premier temps on explique le fonctionnement de l'évaluation de l'influence de K sur l'efficacité du classifieur (1.3.1) puis le fonctionnement de la validation croisée à N répertoires (1.3.2). Dans les parties suivantes (1.3.3, 1.3.4 et 1.3.5) on étudiera les

résultats et notamment les différence dues à l'utilisation de descripteurs d'images à la place des données brutes.

1.3.1 Influence du nombre de voisins K sur l'efficacité du classifieur

Après avoir calculé la matrice des distances D (1.1) il est rapide de calculer l'efficacité du classifieur en fonction du nombre de voisins pris en compte (Voir D.3). En effet en faisant une bloucle sur k allant de 1 à 100 on peut facilement tracer la courbe représentant la précision en fonction de k.

Il est cependant impoossible d'utiliser cette méthode directement pour ajuster l'hyper-paramètre k. en effet on a besoin de connaître les labels des données de test. La partie suivante présente la Validation croisée à N répertoires permettant de s'affranchir de ce problème.

1.3.2 Validation croisée à N répertoires

Le principe est le même que pour la partie précédente mais cette fois-ci on va chercher à ajuster les hyper-paramètres uniquement sur les données d'apprentissage avant de faire une prédiction sur les données de test. On sépare donc les données d'apprentissage en N sous-ensemble qui serviront à tour de rôle de données test lors de l'apprentissage.

Pour tout les N répertoires, on réalise alors le calcul de l'influence de k sur la précision. Il est alors possible d'obtenir N courbes de précision en fonction de k, en prenant le k tels que la précision moyenne soit la plus grande on peut alors espérer obtenir de bons résultats lors de la réelle phase de test sur des données n'ayant jamais servit lors de l'apprentissage.

1.3.3 Utilisation des données brutes

Les données brutes présente l'inconvénient majeur d'avoir une dimension très grandes menant à des calculs excessivement longs (Calcul de 12h environ pour effectuer les calculs sur l'ensemble des données disponibles). On se limitera alors pour cette partie à l'étude n'utilisant que 18000 images d'apprentissage et 2000 images de tests.

On choisi d'utiliser 9 répertoires pour l'estimation du meilleur k possible. La figure 3 représente l'évolution de la précision pour chaque répertoire. La figure 4 quand à elle, correspond à l'évolution de la valeur moyenne de la précision sur l'ensemble des répertoires en fonction de k. Sur cette dernière figure on se rend alors compte que k=7 semble le plus optimisé pour ce jeu de données, permettant d'obtenir une précision moyenne de 29,9%.

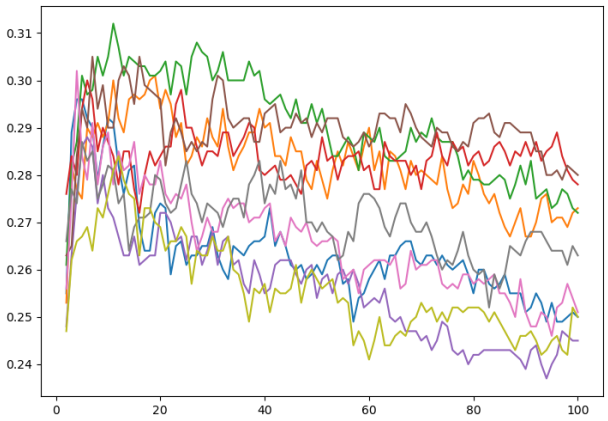

FIGURE 3 – Évolutions des précisions de prédiction pour les 9 répertoires en fonction du nombre K de voisins que l'on considères, dans le cas où on utilise les données brutes.

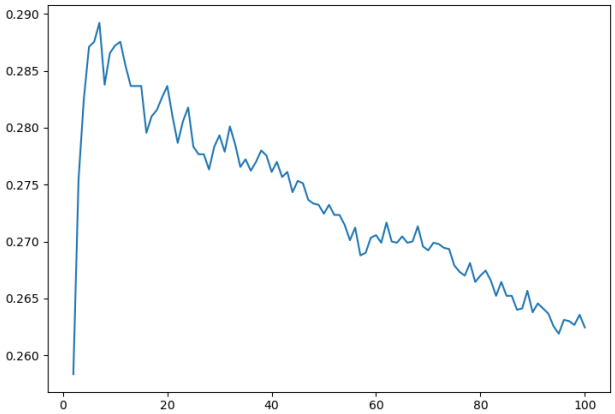

FIGURE 4 – Évolutions de la précision de prédiction moyenne sur les 9 répertoires en fonction du nombre K de voisins que l'on considères, dans le cas où on utilise les données brutes.

Enfin on réalise une prédiction sur les données tests en se fixant k = 7. Avec ces conditions on obtient alors une précision d'environ 30%.

1.3.4 Utilisation du descripteur HOG

On utilise ici l'histogramme des gradients orientés (HOG) comme descripteur d'image. Celui-ci beaucoup plus compacte que les données brutes permet d'effectuer les calculs sur l'ensemble des données de façon bien plus raisonnable. En effet ce descripteur a une dimension de 128 valeurs par images (contre 3072).

On utilise encore 9 répertoires ce qui permet d'obtenir les évolutions de précision figures 5 et l'évolution de sa moyenne figure 6.

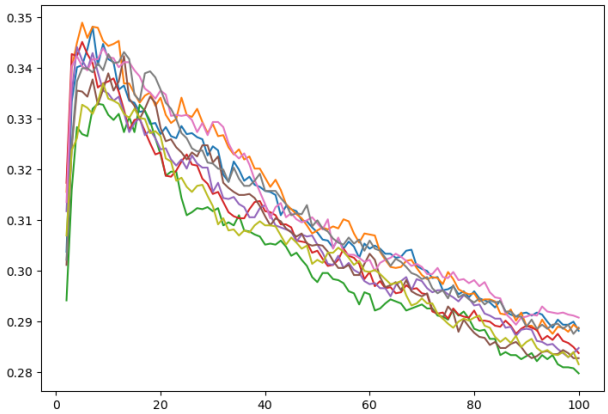

FIGURE 5 – Évolutions des précisions de prédiction pour les 9 répertoires en fonction du nombre K de voisins que l'on considères, dans le cas où on utilise le descripteur HOG.

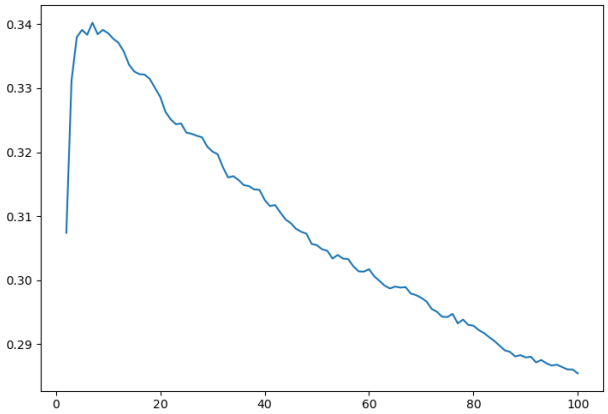

FIGURE 6 – Évolutions de la précision de prédiction moyenne sur les 9 répertoires en fonction du nombre K de voisins que l'on considères, dans le cas où on utilise le descripteur HOG.

On se fixe alors k = 7 pour obtenir une précision sur l'ensemble de test de 34,2%.

Notons tout de même qu'il aurait été intéressant de faire varier les paramètres de calcul du HOG dans une autre validation croisée à N répertoires. En effet les paramètre choisis pour cette expérimentation ont été déterminés pour que cela fonctionne relativement bien, cependant il est presque certain qu'il existe un arrangement de paramètre plus performant que celui-ci.

1.3.5 Utilisation du descripteur LBP

Ce descripteur a une dimension de 18 valeurs par images. Avec 9 répertoires on obtient les évolutions de la précision figures 5 et l'évolution de sa moyenne figure 6.

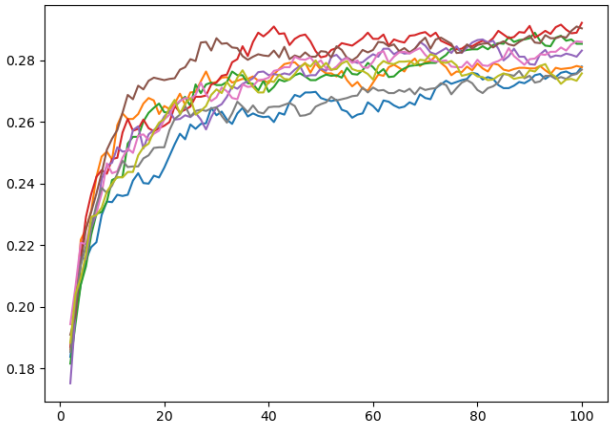

FIGURE 7 – Évolutions des précisions de prédiction pour les 9 répertoires en fonction du nombre K de voisins que l'on considères, dans le cas où on utilise le descripteur LBP.

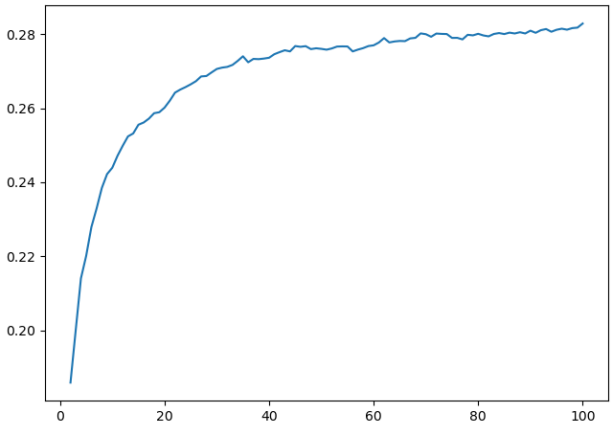

FIGURE 8 – Évolutions de la précision de prédiction moyenne sur les 9 répertoires en fonction du nombre K de voisins que l'on considères, dans le cas où on utilise le descripteur LBP.

On note de façon assez surprenante que la précision semble atteindre un palier pour rester à un valeur presque constante de 28% à partir de k = 80. En réalité on constate que pour des valeurs très grandes de k (supérieurs à 500) on observe encore la décroissance de la précision avec l'augmentation du nombre de voisins. Cette différence avec les représentations précédentes est sûrement due à la très faible dimension de ce descripteur par rapport aux autre.

On se fixe alors k = 100 pour obtenir une précision sur l'ensemble de test de 28,4%.

Ici aussi il aurait été intéressant de faire varier les paramètres de calcul du descripteur dans une autre validation croisée à N répertoires.

2 Système de classification à base de réseaux de neurones

2.1 Classes Reseau et Layer

Afin de permettre d'effectuer les expérimentations plus facilement, on crée un classe *Reseau* (C.1) permettant modifier les fonctions d'activation, le nombre de couche cachée, le taux d'apprentissage ou la fonction de perte. Un *Reseau* est composé de plusieurs objet *Layer* (C.2).

Deux fonctions de la classe Reseau sont particulièrement importantes, il s'agit de *forward* et *backward*. Elles sont chargés d'exécuter respectivement la passe avant et arrière de chaque couche dans le bon ordre afin de calculer Y_pred en fonction de X (resp. les gradients en fonction de la loss). Les fonction appelés par *forward* et *backward* dépendent de la fonction d'activation choisie ("*sigmoïd*", *leaky-relu*" ou "*relu*"). Aussi, pour essayer d'initialiser au mieux les poids de chaque couches, on utilise une fonction d'initialisation différente selon la fonction d'activation.

L'utilisation de cette classe est alors facilement modifiable dans la cellule de code "Reseau de neurone" (D.4).

2.2 Calcul des fonctions backward

2.2.1 Perte de type MSE

L'erreur MSE est calculée comme suit :

```
loss = np.square(Y_pred - Ynn).sum() / (2*mini_batch_size)
```

Ainsi en notant :

$$f(u) = \frac{\|u - \hat{y}\|}{2}$$

la fonction permettant de calculer la loss en fonction de la prédiction u. Où $\hat{y}$ est le vecteur de référence. On peut facilement calculer :

$$\nabla f = u - \hat{y}$$

En pratique on calcul le gradient à partir d'un minibatch de donné, on considère donc ce dernier comme étant la moyenne des gradients de chaque données du minibatch, c'est pourquoi on ajoute une division par N dans le calcul de la perte et dans le calcul du gradient. Cela permet aussi de travailler avec des valeurs de taux d'apprentissage plus proche lorsqu'on change la taille des minibatch.

2.2.2 Perte de type cross-entropy

Soit :

$$p_i = \frac{e^{u_i}}{\sum_j e^{u_j}}$$

la probabilité normalisée associée au label i. Où $\hat{y}$ est le vecteur de référence et u et le vecteur de prédiction. On peut alors calculer l'erreur de type cross-entropy comme suit :

$$f(u) = -\sum_i \widehat{y_i} \times log(p_i)$$

Or $\hat{y}$ est nul partout sauf pour l'une de ses coordonnées qui est égale à 1, ce dernier agit donc comme un sélecteur sur le vecteur de prédiction. Soit k l'indice pour lequel $\hat{y}_k = 1$. On a alors :

$$f(u) = -log(p_k) = -log(\frac{e^{u_k}}{\sum_j e^{u_j}}) = log(\sum_j e^{u_j}) - u_k$$

Or.

$$\frac{\partial}{\partial u_i} log(\sum_j e^{u_j}) = \frac{1}{\sum_j e^{u_j}} . \frac{\partial}{\partial u_i} \sum_j e^{u_j} = \frac{u_i}{\sum_j e^{u_j}} = p_i$$

Enfin si i=k :

$$\frac{\partial}{\partial u_i} f(u) = p_i - 1$$

Sinon :

$$\frac{\partial}{\partial u_i} f(u) = p_i$$

On peut alors utiliser $\hat{y}$ et $1 - \hat{y}$ pour distinguer ces cas. D'où le code suivant (ici aussi on fait la moyenne sur l'ensemble des données du minibatch d'où la division par la taille de ces derniers :

```
#Cross entropy loss
Y_pred_exp = np.exp(Y_pred)
Y_pred = (Y_pred_exp.T/np.sum(Y_pred_exp,axis=1)).T
loss = - np.sum( Ynn*np.log(Y_pred) ) / mini_batch_size

grad_Y_pred = (Ynn * (Y_pred - 1) + (1-Ynn) * Y_pred) / mini_batch_size
```

2.2.3 Fonction d'agrégation

Toute les fonction forward on la forme suivante :

```
#Commun
self.m_Xi = Xi
self.m_Ii = Xi.dot(self.m_Wi) + self.m_bi
# Activation
...
```

Où la partie annotée "*Commun*" est commune à toutes les fonctions forward. Souvent nommé fonction d'agrégation, elle permet de calculer la matrice *Ii* correspondant aux données en sortie de la couche avant la fonction d'activation. La partie annotée "*Activation*" dépends alors de la fonction d'activation.

Pour la passe arrière il faut donc déterminer comment calculer les gradients des poids *Wi*, des biais *bi* et des données d'entrée *Xi* à partir du gradient de la fonction d'activation par rapport à la perte.
D'autre part, la présence éventuelle d'un régularisation impose une modification du gradient. On a donc la code suivant devant être exécute après le calcul du gradient de la fonction d'activation. Et ce pour chaque couche du réseau.

```
# Activation
...

#Commun
self.m_grad_bif = np.ones((len(self.m_Xi),1)).T.dot(self.m_grad_Iif)
self.m_grad_Wif = self.m_Xi.T.dot(self.m_grad_Iif)

#Regularisation
if(self.m_taux_regul):
    self.m_grad_bif += self.m_bi * self.m_taux_regul
    self.m_grad_Wif += self.m_Wi * self.m_taux_regul

self.m_grad_Xif = self.m_grad_Iif.dot(self.m_Wi.T)
return self.m_grad_Xif
```

2.2.4 Sigmoïd

On a pour la passe avant :

```
# Commun
...

# Activation
self.m_Oi = 1/(1+np.exp(-self.m_Ii))
```

Il faut donc déterminer comment calculer le gradient de la fonction sigmoïd par rapport à sa sortie puis utiliser la propagation du gradient pour obtenir les gradients des poids *Wi*, des biais *bi* et des données d'entrée *Xi*.

En notant :

$$sigmoid(v) = \frac{1}{1+e^{-v}}$$

On a :

$$\nabla sigmoid = g(v)(1-g(v))$$

D'où le code :

```
# Activation
self.m_grad_Iif = (1-self.m_Oi)*self.m_Oi * grad_Oif

# Commun
...
```

2.2.5 Relu et Leaky-relu

On a pour la passe avant :

```
# Commun
...

# Activation
self.m_Oi = np.maximum(self.m_Ii,0)
return self.m_Oi
```

Pour chaque composante yi du vecteur Ii en sortie de la fonction d'agrégation, la dérivée partielle est nulle si $yi < 0$ et elle est égale à 1 sinon. D'où le code suivant, "(self.m_Oi>0)" sert de "masque" que l'on applique sur le gradient de sortie de la couche.

```
# Activation
self.m_grad_Iif = (self.m_Oi>0) * grad_Oif

# Commun
...
```

2.3 Expérimentation préliminaires pour trouver une structure capable d'overfitter 512 images

Il est difficile de tester indépendemment les effet de chaque hyper-paramètre, en effet une structure donnée (nombre de couche et nombre de neurones par couche) avec un taux d'apprentissage donné, une fonction de perte donnée, etc.. peut très bien fonctionner avec une fonction d'activation de type *sigmoïd* mais donner des résultats inexploitables avec une fonction d'activation de type *relu*. Les expérimentations présentées dans les parties suivantes sont donc loin d'être exhaustives et on se contentera de faire varier les hyper-paramètres de façon arbitraire en essayant de trouver à chaque fois ce qui fonctionne le mieux **pour overfitter un petit ensemble de 512 images**.

2.3.1 Une seule couche cachée

Pour cette première partie on commence avec un cas proche du cas de départ, les paramètres sont les suivants :

- N = 512
- mini_batch_size = 512
- N_epoch = 20000
- D_layers = [512]
- learning_rate = 2e-1
- loss = 'MSE'
- activation='sigmoid'
- taux_regul = 0

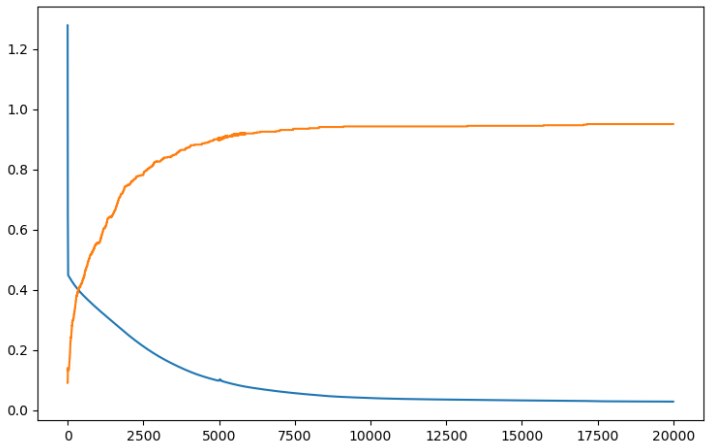

FIGURE 9 – Évolutions de la perte (en bleu) et de la précision de prédiction (en orange) en fonction du nombre d'itérations

On commence par tester l'entraînement du réseau sur un nombre réduit de donnée en considérant un batch les comprenant toutes. On effectue alors 20000 cycles forward-

backward sur l'ensemble de ces données. On mesure alors l'évolution de la perte moyenne ainsi que la capacité de ce réseau à (over)fitter les données. En effet si le réseau n'est déjà pas capable d'atteindre des taux de prédiction relativement bon sur les donnée d'apprentissage, on ne peut pas espérer avoir de bons résultats sur des données de test jamais utilisées. Les résultats sont représentés sur la figure 9

Plusieurs point sont intéressant à noter :

— Cette structure de réseau est capable de relativement bien (over)fitter les données
— Le temps de convergence est très long pour un cas aussi simple
— Après une décroissance extrême de la loss à la première itération, la décroissance suit l'allure d'une exponentielle.

On pourrait alors être tenté d'augmenter le taux d'apprentissage, par exemple à 8e-1 au lieu de 2e-1. Cependant comme on peut le voir sur la figure 10, le résultat obtenu suit d'abord un pallier (probablement du à une explosion du gradient sur les premières itérations) puis est très bruité. Si la loss suit une allure décroissance globalement il est difficilement envisageable d'utiliser ceci pour entraîner le réseau. Aussi le temps de convergence est ici encore bien trop grand pour un cas aussi simple.

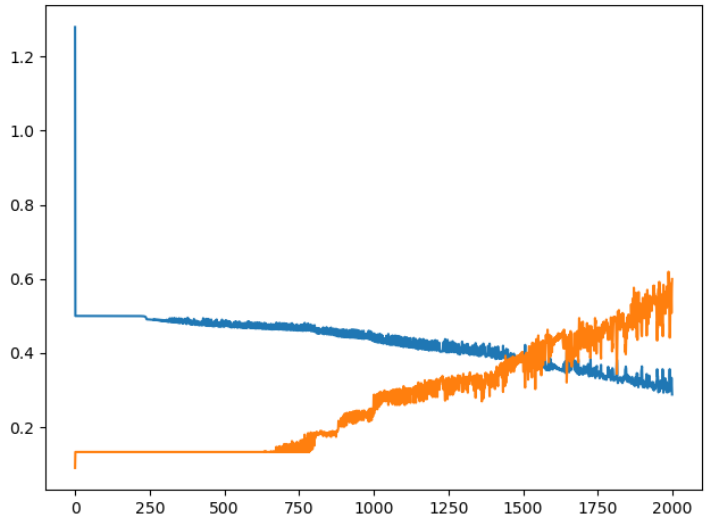

FIGURE 10 – Évolutions de la perte (en bleu) et de la précision de prédiction (en orange) en fonction du nombre d'itérations

2.3.2 Utilisation d'une fonction de perte de type cross entropy

Dans cette partie on s'intéresse a l'influence de la fonction de perte et de la fonction d'activation sur la vitesse de convergence du réseau. On remplace donc la perte de type MSE (2.2.1 par une perte de type cross entropy (2.2.2). On utilise alors les paramètres suivant :

— N = 512
— mini_batch_size = 512
— N_epoch = 2000
— D_layers = [512]
— learning_rate = 1e-1
— loss = 'cross entropy'
— activation='sigmoid'

— taux_regul = 0

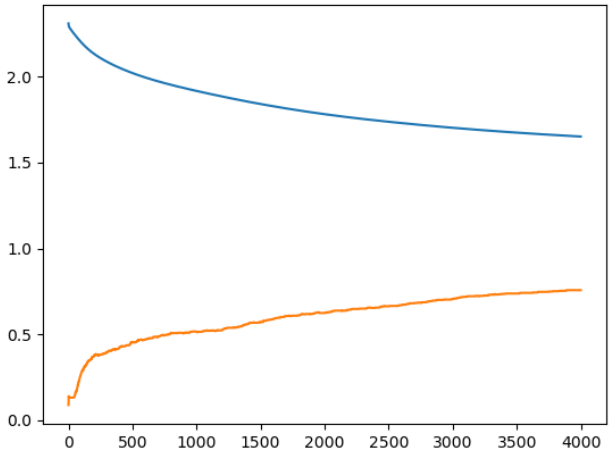

FIGURE 11 – Évolutions de la perte (en bleu) et de la précision de prédiction (en orange) en fonction du nombre d'itérations

On constate (Figure 11) que l'utilisation de cette fonction de perte permet une décroissance plus fluide de la perte, elle permet aussi d'atteindre des taux de précisions plus grand en un temps identique cependant on est encore loin d'un résultat satisfaisant.

2.3.3 Utilisation d'une fonction d'activation de type relu

Maintenant on va tester l'influence de la fonction d'activation en replaçant la fonction d'activation de type "sigmoïd" (2.2.4) par un fonction de type relu (2.2.5). Les paramètres utilisés sont alors les suivants :

— N = 512
— mini_batch_size = 512
— N_epoch = 2000
— D_layers = [512]
— learning_rate = 1e-1
— loss = 'cross entropy'
— activation='relu'
— taux_regul = 0

On obtient alors les résultats suivants présentés sur la figure 12.

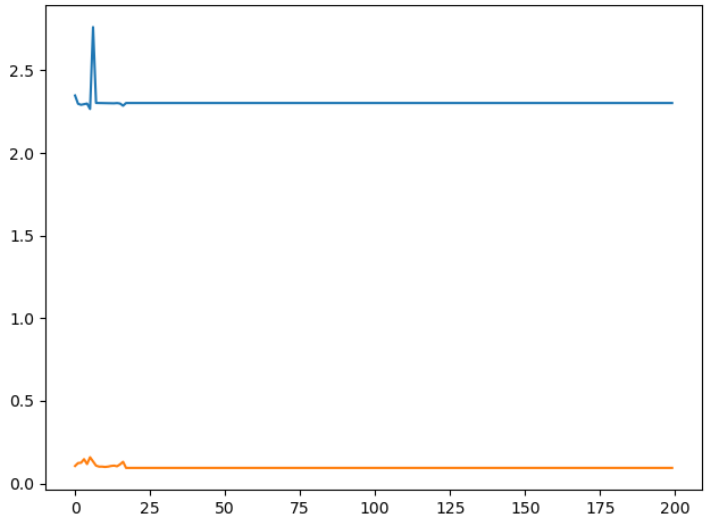

FIGURE 12 – Évolutions de la perte (en bleu) et de la précision de prédiction (en orange) en fonction du nombre d'itérations

On remarque qu'avec ce taux d'apprentissage les neurones se retrouvent directement saturés, le gradient est alors nul partout et on observe un palier constant à partir de la 5ème itération. On réalise alors la même expérience avec une fonction d'activation de type **leaky** relu au lieu de relu simple qui devrait permettre d'éviter de se retrouver dans des situations où les neurones sont saturés et le gradient annihilé lorsqu'on se trouve dans le domaine négatif.

On obtient alors les résultats suivants présentés sur la figure 13.

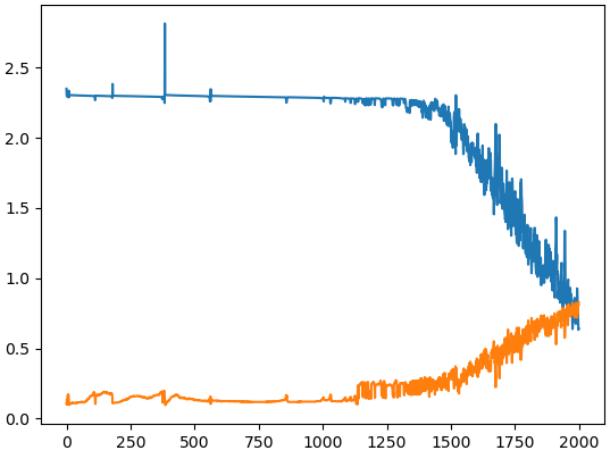

FIGURE 13 – Évolutions de la perte (en bleu) et de la précision de prédiction (en orange) en fonction du nombre d'itérations

Utiliser cette fonction d'activation est donc un peu mieux car elle permet d'éviter ce phénomène de saturation complète du réseau, cependant on constate que le taux d'apprentissage est toujours trop grand et engendre des très grandes variations de la perte d'un

itération à l'autre. En utilisant les paramètres suivant on peut peut-être espérer avoir un décroissance moins chaotique :

— N = 512
— mini_batch_size = 512
— N_epoch = 2000
— D_layers = [512]
— learning_rate = 1e-2
— loss = 'cross entropy'
— activation='leaky_relu'
— taux_regul = 0

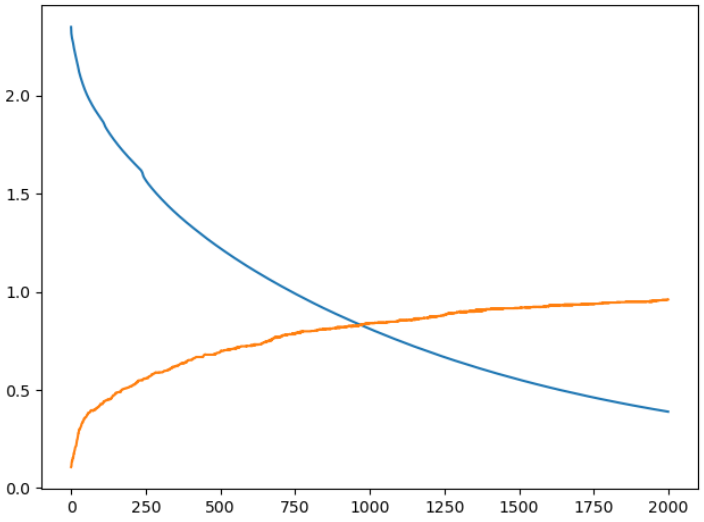

FIGURE 14 – Évolutions de la perte (en bleu) et de la précision de prédiction (en orange) en fonction du nombre d'itérations

On obtient des résultats suivants qui sont bien plus satisfaisants (Voir Figure 14). En effet la décroissance de la perte est bien moins chaotique et on atteint une perte relativement satisfaisante par rapport aux expérimentations précédentes.

2.3.4 Ajout de couches cachées

Dans cette partie on réalise plusieurs expériences avec un nombre de couches variable. En effet l'ajout de couches devrai permettre de pouvoir extraire des motifs de l'image et de les combinés afin d'en trouver une signification. Dans un premier temps on se contente d'ajouter une couche de 256 neurones après la première couche de 512. On utilise donc les paramètres suivants :

— N = 512
— mini_batch_size = 512
— N_epoch = 2000
— D_layers = [512,256]
— learning_rate = 2e-1
— loss = 'cross entropy'
— activation='leaky_relu'
— taux_regul = 0

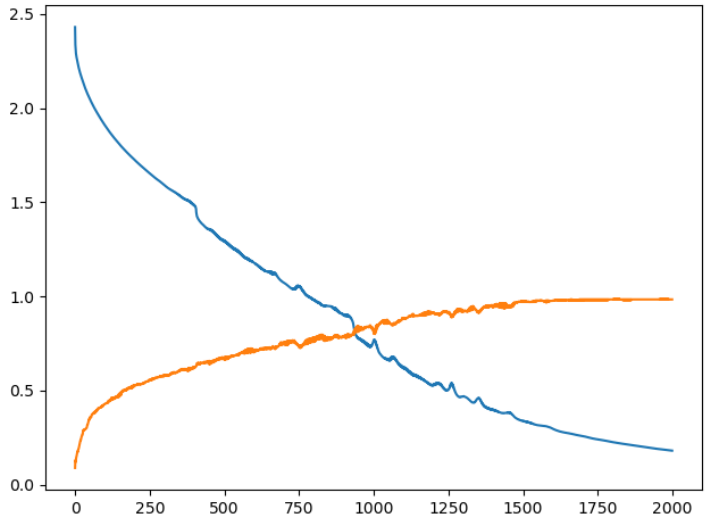

FIGURE 15 – Évolutions de la perte (en bleu) et de la précision de prédiction (en orange) en fonction du nombre d'itérations

On obtient alors des résultats un peu meilleurs que précédemment, en effet on observe une convergence plus rapide, un perte finale inférieure et un meilleur taux de prédiction (504 images sur 512). Cependant on remarque de la décroissance est un peu moins stable qu'avec une seule couche. Cette instabilité est probablement du à multiplication des erreurs du à la modification simultanée de tout les poids.

2.4 Expérimentations

Maintenant qu'on a une structure capable d'overfitter de façon satisfaisante les données d'apprentissage on va commencer à essayer de réaliser de vraies prédictions. Dans un premier temps on entraîne le réseau sur 9000 images d'apprentissage (voir les paramètres ci-dessous) et on test cet apprentissage sur 1000 images de test.

- N = 9000
- mini_batch_size = 9000
- N_epoch = 2000
- D_layers = [512,256]
- learning_rate = 1e-2
- loss = 'cross entropy'
- activation='leaky_relu'
- taux_regul = 0

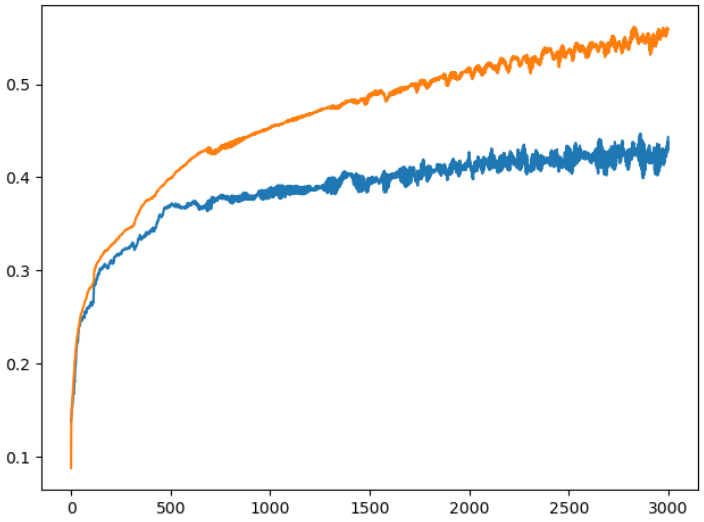

FIGURE 16 – Évolutions des précisions de prédiction par rapport aux données d'entraînement (en orange) et par rapport au données de test (en bleu)

On constate qu'on arrive relativement vite sur un plateau de précision et que ce dernier est très bruité. Aussi, même si on ne le constate pas ici, en attendant suffisamment longtemps on devrait commencer à observer une décroissance de la précision sur les données de tests alors que la précision sur les données d'apprentissage continue d'augmenter. L'instant au on constate cette décroissance est le moment à partir du quel le réseau commence à overfitter. Les parties suivantes présentes des amélioration pour réduire ce défaut.

2.4.1 Ajout d'un terme de régularisation dans la fonction de perte

Pour réduire le bruit précèdent probablement du à des poids trop importants entre les couches, on ajoute cette fois un terme de régularisation L2 sur les poids des couches. Cela permet aussi de limiter l'overfitting sur réseau, le rendant ainsi plus généralisable.

— N = 9000
— mini_batch_size = 9000
— N_epoch = 2000
— D_layers = [512,256]
— learning_rate = 1e-2
— loss = 'cross entropy'
— activation='leaky_relu'
— taux_regul = 0.01

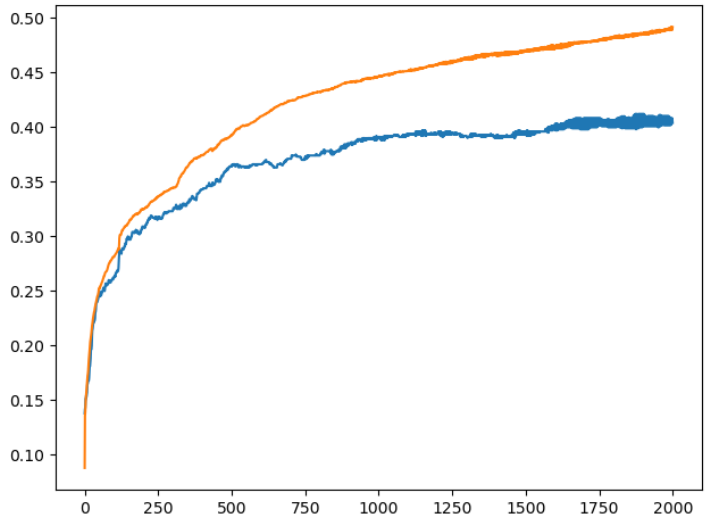

FIGURE 17 – Évolutions des précisions de prédiction par rapport aux données d'entraînement (en orange) et par rapport au données de test (en bleu)

On constate que la courbe est lissée et que l'écart entre les données d'apprentissage et les données d'entraînement est moins important. En réalisant le test sur plusieurs valeurs de `taux_regul` on trouve que 0.01 fonctionne bien. On pourrait cependant combiner cette méthode avec un taux d'apprentissage adaptatif pour permettre un ajustement plus précis pour des poids proches de 0.

2.4.2 Descente de gradient par mini batch

Une autre façon de procéder pour réduire l'overfitting et accèlerer l'apprentissage est l'utilisation de mini-batch pour la descente de gradient. On parcours alors l'ensemble des données (epochs) plusieurs fois en ne prenant à chaque fois que des sous-ensemble relativement petits des données d'apprentissage.

En modifiant les paramètres précédents pour prendre en compte plusieurs mini-batch on obtient la figure **??** :

- N = 45000
- mini_batch_size = 45
- N_epoch = 100
- D_layers = [512,256]
- learning_rate = 1e-2
- loss = 'cross entropy'
- activation='leaky_relu'
- taux_regul = 0.01

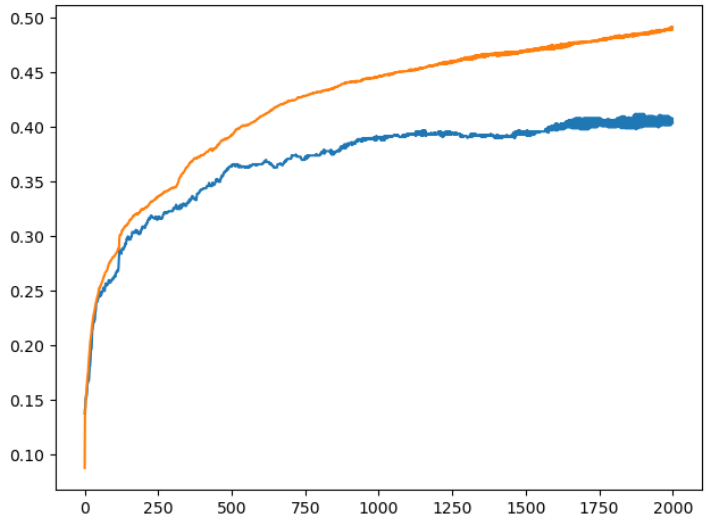

FIGURE 18 – Évolutions des précisions de prédiction par rapport aux données d'entraînement (moyenne sur un epoch en orange) et par rapport au données de test (en bleu)

On constate que l'utilisation de mini-batch accélère grandement l'apprentissage, en effet après avoir vu une seule fois les données on atteint déjà des précisions de presque 20%. En effet cela permet de réaliser bien plus d'itération de descente de gradient avec le même nombre de données, cependant cette descente s'avère bien plus aléatoire étant donné qu'on ne prend en compte qu'une partie de l'ensemble des données à la fois (d'ou le terme de gradient stochastique)

2.5 Évaluation du réseau obtenu

On réalise enfin l'expérience avec l'ensemble des données d'apprentissage et les paramètres suivants :

- N = 45000
- mini_batch_size = 45
- N_epoch = 100
- D_layers = [512,256]
- learning_rate = 1e-2
- loss = 'cross entropy'
- activation='leaky_relu'
- taux_regul = 0.01

Après environ 3h d'apprentissage on obtient les résultats représentés sur la figure 20. Avec toutes ces données on observe une allure de la courbe d'évolution de la précision similaire à précédemment mais les valeurs atteinte sont bien plus grandes : jusqu'à 52% sur les données de test en seulement 200 epochs (Figure 18)

On remarque aussi que malgré les trois heures d'apprentissage aucun pallier n'est encore réellement atteint. Il serait alors possible (avec la diminution du taux d'apprentissage d'atteindre des valeurs de 55% à 60% de précisions.

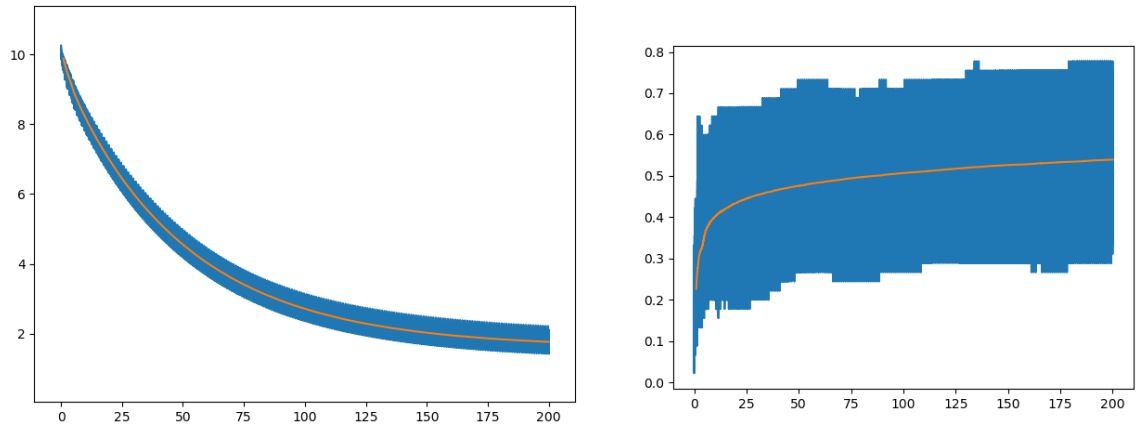

FIGURE 19 – Évolutions de la loss (à gauche) et de la précision (à droite). Les moyenne sur chaque epoch sont en orange.

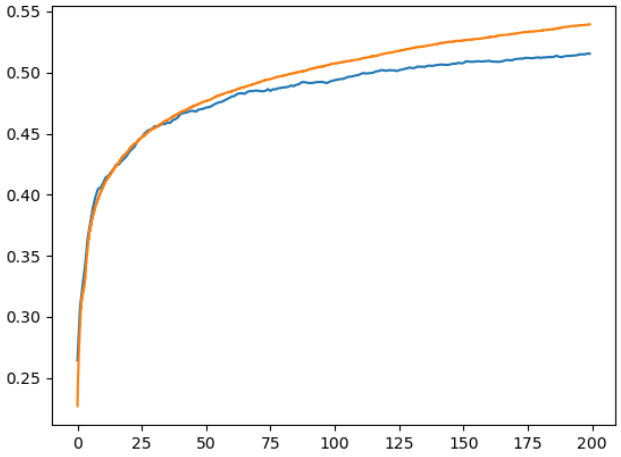

FIGURE 20 – Évolutions des précisions de prédiction par rapport aux données d'entraînement (en orange) et par rapport au données de test (en bleu)

3 Conclusion

Les deux méthodes de classification vues au cours de ce BE présentent des points intéressants. La première classification par les k plus proche voisins présente l'intérêt d'être facile à mettre en place, cependant les prédictions restent peu fiables avec seulement 35% de prédiction juste dans le meilleur des cas. Aussi cette méthode ne nécessite aucun temps d'apprentissage en revanche le temps de prédiction est très important (au moins proportionnel au nombre de données d'apprentissage)

D'autre part la méthode de classification par réseau de neurones nécessite un peu plus de travail pour la mettre en place mais permet d'obtenir des résultats bien meilleurs (jusqu'à 52%). Aussi si le temps d'apprentissage peut ici être très long, la prédiction se fait cette fois en temps constant une fois le réseau entraîné.

Annexes

Annexe A prepareData.py

A.1 Lecture cifar

```
def lecture_cifar(path,isTest=False):
    """
    Parameters
    ----------
    path : String
        Chemin vers le dossier des données cifar.
    isTest : Boolean, optional
        S'il s'agit d'un test on ne lit qu'un batch. The default is False.

    Returns
    -------
    X : Array(N,3072)
        Ensemble des vecteurs images (N = 10000 si isTest ,N = 50000 sinon)
    """
    nBatch = 5
    if isTest:
        nBatch = 1

    batchSize = 10000

    N = batchSize * nBatch
    D = 32 * 32 * 3

    X = np.zeros((N,D),dtype=('float32'))
    Y = np.zeros((N,1),dtype=('int'))

    import pickle
    for k in range(nBatch):
        with open(os.path.join(path,"data_batch_"+str(k+1)), 'rb') as fo:
            dict = pickle.load(fo, encoding='bytes')
            X[k*batchSize:(k+1)*batchSize,:] = dict[b'data']
            Y[k*batchSize:(k+1)*batchSize,0] =
            np.array(dict[b'labels'],dtype=('int'))
    X = X/256

    return X,Y
```

A.2 Découpage données

```
def decoupage_donnees(X,Y,taux=0.2):
    """
    Parameters
    ----------
    X : Array(NxD)
        Matrice des données (N nombre de données, D dimmension des données)

    Y : Array(Nx1)
        Vecteur des labels (N nombre de données).
```

```
    taux : float, optional
        Part des données etant . The default is 0.2.

    Returns
    -------
    Xapp,Yapp : Array(Napp,D), Array(Napp,1) , Napp = N*(1-taux)
        Données et labeld d'apprentissage

    Xtest,Ytest : Array(Ntest,D), Array(Ntest,1) , Napp = N*taux
        Données et labeld de test

    """

    mask = np.ones(len(Y), dtype=bool)
    mask[:int(len(Y)*taux)] = False
    np.random.shuffle(mask)

    Xapp = X[mask,:]
    Yapp = Y[mask]
    Xtest = X[mask==False,:]
    Ytest = Y[mask==False]
    return Xapp,Yapp,Xtest,Ytest
```

A.3 Conversion en LBP

```
def rgb2LBP(X,n_points = 16, radius = 2):
    """
    Parameters
    ----------
    X : Array(NxD)
        Matrice des données à convertir (N nombre de données, D dimension
        des données)
    n_points : int, optional
        Nombre de point à prendre en compte pour le calcul des motifs
        binaires locaux. The default is 16.
    radius : float, optional
        Rayon du cercle à prendre en compte pour le calcul. The default is
        2.

    Returns
    -------
    newX : Array(NxD2)
        Matrice des données converties (N nombre de données, D2 dimension
        des données converties)

    """
    reshaped = np.reshape(X, (len(X),32,32,3),'F')
    newX = np.zeros((len(X),18))
    for i,x in enumerate(reshaped):
        img = rgb2gray(x)
        lbp = local_binary_pattern(img, n_points, radius, 'uniform')
        n_bins = int(lbp.max() + 1)
        hist, _ = np.histogram(lbp, density=True, bins=n_bins, range=(0, n_bins))
        newX[i,:] = hist
    return newX
```

A.4 conversion en HOG

```
def rgb2HOG(X,orientations=8,cellSize=4,cellPerBlock=1):
    """
    Parameters
    ----------
    X : Array(NxD)
        Matrice des données à convertir (N nombre de données, D dimension
        des données)
    orientations : int, optional
        Nombre d'orientations du gradient. The default is 8.
    cellSize : int, optional
        Taille des cellules dans lesquels on calcul le gradient. The
        default is 4.
    cellPerBlock : int, optional
        Nombre de cellules par blocs. The default is 1.
    Returns
    -------
    newX : Array(NxD2)
        Matrice des données converties (N nombre de données, D2 dimension
        des données converties)

    """
    reshaped = np.reshape(X, (len(X),32,32,3),'F')

    hogTest = hog(reshaped[0],
        orientations=orientations,
        pixels_per_cell=(cellSize, cellSize),
        cells_per_block=(cellPerBlock, cellPerBlock),
        block_norm='L2-Hys',
        feature_vector=True,
        multichannel=True)
    newX = np.zeros((len(X),len(hogTest)))

    for i,x in enumerate(reshaped):
        hogImg = hog(x,
            orientations=orientations,
            pixels_per_cell=(cellSize, cellSize),
            cells_per_block=(cellPerBlock, cellPerBlock),
            block_norm='L2-Hys',
            feature_vector=True,
            multichannel=True)
        newX[i,:] = hogImg
    return newX
```

Annexe B kppv.py

B.1 Distances kppv

```
def kppv_distances(Xtest,Xapp,batchSize = 20):
    """

    Parameters
    ----------
    Xtest : Array(NtestxD)
        Matrice des données de test (Ntest nombre de données
```

```
        d'apprentissage', D dimmension des données)

    Xapp : Array(NappxD)
        Matrice des données d'apprentiassage(N nombre de données de test,
        D dimmension des données)

    Returns
    -------
    Dist : Array(Ntest,Napp)
        Matrice des distances l2 entre toutes les données de l'ensemble de
        test par rapport à toutes les données de l'ensemble
        d'apprentissage.

    """
    Dist = np.zeros((len(Xtest),len(Xapp)),dtype=('float32'))

    nBatchI = len(Xapp)//batchSize + 1
    nBatchJ = len(Xtest)//batchSize + 1
    for j in range(nBatchJ):
        batchStartJ = j*batchSize
        batchEndJ = (j+1)*batchSize
        #reshape de chaque batch pour pouvoir utiliser le broadcast de numpy
        testBatch = Xtest[batchStartJ:batchEndJ,None,:]

        print(str(j) + "/" + str(nBatchJ))
        for i in range(nBatchI):
            batchStartI = i*batchSize
            batchEndI = (i+1)*batchSize

            #reshape de chaque batch pour pouvoir utiliser le broadcast de
            numpy
            appBatch = Xapp[None,batchStartI:batchEndI,:]

            #squaredDiff est un tenseur où squaredDiff[j1,j2,i] =
                testBatch[j1,i] - appBatch[j2,i]
            squaredDiff = np.square(testBatch - appBatch)
            Dist[batchStartJ:batchEndJ,batchStartI:batchEndI] =
                np.sqrt(np.sum(squaredDiff,axis=2))
    return Dist
```

B.2 Prédiction kppv

```
def kppv_predict(Dist,Yapp,k):
    """
    Parameters
    ----------
    Dist : Array(Ntest,Napp)
        Matrice des distances l2 entre toutes les données de l'ensemble de
        test par rapport à toutes les données de l'ensemble
        d'apprentissage.
    Yapp : Array
        Vecteur des labels correspondant aux données d'apprentissage
        utilisées pour calculer Dist
    k : int
        Nombre de plus proche voisins a prendre encompte
```

```
        Returns
        -------
        Ypred : Array
            le vecteur des classes prédites pour les éléments de Xtest

        """
        minIndicies = np.argpartition(Dist,k,axis=1)[:,:k]
        #nearestDistances = np.take_along_axis(Dist,minIndicies,1)
        nearestLabels = np.reshape(Yapp[minIndicies],(len(Dist),k))

        # Calcul de label le plus present parmis les k plus proches voisins
        # (Problème en cas d'égalité, possibilité de choisir les voisins les
            plus proches dans ce cas)
        # On reste avec une approche simple renvoyant le label d'indice le plus
        faible dans ce cas
        axis=1
        u, indices = np.unique(nearestLabels, return_inverse=True)
        Ypred = u[np.argmax(np.apply_along_axis(np.bincount, axis,
            indices.reshape(nearestLabels.shape),
            None, np.max(indices) + 1), axis=axis)]

        return Ypred
```

Annexe C reseau.py

C.1 Classe Reseau

```
class Reseau:

    def __init__(self, D_in, D_layers, D_out, activation='sigmoid',
    taux_regul = 0):
        """
        Parameters
        ----------
        D_in : int
            Dimension des données d'entrée'
        D_layers : int[]
            Nombre de neurone des couches cachées.
        D_out : int
            Dimension de sortie (nombre de neurones de la couche de sortie)
        activation : String ('sigmoid', 'relu'), optional
            Fonction d'activation à utiliser. The default is 'sigmoid'.

        Returns
        -------
        None.

        """
        self.m_D_in = D_in
        self.m_D_layers = D_layers
        self.m_D_Dout = D_out

        #Initialisation des couches
        self.m_layers = []
        self.m_layers.append(Layer(D_in, D_layers[0], activation,
```

```
        taux_regul))
        for k in range(len(D_layers)-1):
            self.m_layers.append(Layer(D_layers[k], D_layers[k+1],
            activation, taux_regul))
        self.m_layers.append(Layer(D_layers[-1], D_out, activation,
        taux_regul))

    def forward(self, X):
        """
        Calcul Ypred en fonction de X en appelant successivement la
        fonction forward correspondant à la fonction d'activation souhaitée
        de chaque couche du reseau.
        Chaque couche enregistre le resultat intermédiaire pour le calcul
        du gradient.

        Parameters
        ----------
        X : Array(N,D_in)
            Matrice d'entrée correspondant à l'enssemble des données à
            classer

        Returns
        -------
        Ypred : Array(N,D_out)
            Matrice de prediction des labels pour chaque entrée

        """
        Xi = X
        for l in self.m_layers:
            Oi = l.m_forward(l,Xi)

            #L'entrée de la couche suivante est la sortie de la couche actuelle
            Xi = Oi.copy()
        Ypred = Oi
        return Ypred

    def backward(self, grad_Y_predf):
        """
        Calcul l'ensemeble des gradient utiles en fonction de gras_Y_predf
        et des données caluclées lors de l'appel à forward.
        Chaque couche enregistre les resultats correspondants au gradients
        des poids nécéssaire pour la mise à jour de ces derniers
        Parameters
        ----------
        grad_Y_predf : Array(N,D_in)
            Gradient de Y_pred par rapport à la loss

        Returns
        -------
        None

        """
        grad_Oif = grad_Y_predf
        for l in reversed(self.m_layers):
            grad_Xif = l.m_backward(l,grad_Oif)
            #le gradient de
            grad_Oif = grad_Xif.copy()
```

```
    def update_weights(self,learning_rate):
        """
        Mets à jours les poids en fonction des gradients calculés à l'appel
        de backward.

        Parameters
        ----------
        learning_rate :
            taux d'apprentissage'

        Returns
        -------
        None

        """
        for l in self.m_layers:
            l.update_weights(learning_rate)

    def get_regularisation(self):
        """
        Returns
        -------
        float :
            Somme de l'ensemble du carré des poids du reseau.

        """
        regul = 0
        for l in self.m_layers:
            regul += l.get_regularisation()
        return regul
```

C.2 Classe Layer

```
class Layer:

    def __init__(self, D_Xi, D_Oi , activation='sigmoid', taux_regul = 0):
        """
        Parameters
        ----------
        D_Xi : int
            Dimension de l'entrée de la couche.
        D_Oi : int
            Dimension de l'a sortie de la couche.
        activation : String ('sigmoid', 'relu')
            Fonction d'activation à utiliser (default = 'sigmoid'')

        Returns
        -------
        None.

        """
        self.m_D_Xi = D_Xi
        self.m_D_Oi = D_Oi
        self.m_activation = activation
        self.m_taux_regul = taux_regul
```

```
        if(activation == 'relu'):
            self.m_forward = Layer.relu_forward
            self.m_backward = Layer.relu_backward
            self.relu_init(D_Xi,D_Oi)
        elif(activation == 'leaky_relu'):
            self.m_forward = Layer.leaky_relu_forward
            self.m_backward = Layer.leaky_relu_backward
            self.relu_init(D_Xi,D_Oi)
        else: ## activation = 'sigmoid' ?
            self.m_forward = Layer.sigmoid_forward
            self.m_backward = Layer.sigmoid_backward
            self.sigmoid_init(D_Xi,D_Oi)


    def update_weights(self,learning_rate):
        """
        Modifie les poids de la couche en fonction des gradients caluclés
        et du taux d'apprentissage'

        Parameters
        ----------
        learning_rate : float
            Taux d'apprentissage'

        Returns
        -------
        None
        """
        self.m_Wi = self.m_Wi - self.m_grad_Wif * learning_rate
        self.m_bi = self.m_bi - self.m_grad_bif * learning_rate
        return 0


    def sigmoid_forward(self, Xi):
        """
        Applique la fonction sigmoïd sur le potentiel d'entrée de la couche
        cachée Ii

        Parameters
        ----------
        Xi : Array
            Matrice d'entrée de la couche'

        Returns
        -------
        Array
            Matrice de sortie Oi pour la fonction d'activation "sigmoïd"
        """
        #Commun
        self.m_Xi = Xi
        self.m_Ii = Xi.dot(self.m_Wi) + self.m_bi

        # Activation
        self.m_Oi = 1/(1+np.exp(-self.m_Ii))
        return self.m_Oi
```

```
    def sigmoid_backward(self, grad_Oif):
        """
        Calcul les gradients des poids (Wi et bi) et de l'entrée (Xi) à
        partir du gradient de la sortie(Oi) pour la fonction d'activation
        "sigmoid".
        Le gradient des poids est modifié pour prendre en compte la
        régularisation si besoin.

        Parameters
        ----------
        grad_Oif : Array
            Matrice du gradient de la sortie Oi par rapport à la loss.

        Returns
        -------
        Array
            Matrice du gradient de la sortie Oi par rapport à la loss pour
            la fonction d'activation "sigmoid".
        """
        # Activation
        self.m_grad_Iif = (1-self.m_Oi)*self.m_Oi * grad_Oif

        #Commun
        self.m_grad_bif =
        np.ones((len(self.m_Xi),1)).T.dot(self.m_grad_Iif)
        self.m_grad_Wif = self.m_Xi.T.dot(self.m_grad_Iif)
        #Regularisation
        if(self.m_taux_regul):
            self.m_grad_bif += self.m_bi * self.m_taux_regul
            self.m_grad_Wif += self.m_Wi * self.m_taux_regul

        self.m_grad_Xif = self.m_grad_Iif.dot(self.m_Wi.T)
        return self.m_grad_Xif

    def sigmoid_init(self,D_Xi,D_Oi):
        """
        Initialisation des poids ajustée pour la fonction d'activation
        "sigmoïd"'

        Parameters
        ----------
        D_Xi : int
            Dimension du vecteur d'entrée de la couche.
        D_Oi : int
            Dimension du vecteur de sortie de la couche.

        Returns
        -------
        None.
        """
        # W est la matrice des poids de chaque sinaps de la couche
        # b est la matrice des bias en entrée
        self.m_Wi = np.random.randn(D_Xi, D_Oi) / np.sqrt(D_Xi)
        self.m_bi = np.zeros((1,D_Oi))


    def relu_forward(self, Xi):
```

```
        """
        Applique la fonction relu sur le potentiel d'entrée de la couche
        cachée Ii

        Parameters
        ----------
        Xi : Array
            Matrice d'entrée de la couche'

        Returns
        -------
        Array
            Matrice de sortie Oi pour la fonction d'activation "relu""
        """
        #Commun
        self.m_Xi = Xi
        self.m_Ii = Xi.dot(self.m_Wi) + self.m_bi

        # Activation
        self.m_Oi = np.maximum(self.m_Ii,0)
        return self.m_Oi

    def relu_backward(self, grad_Oif):
        """
        Calcul les gradients des poids (Wi et bi) et de l'entrée (Xi) à
        partir du gradient de la sortie(Oi) pour la fonction d'activation
        "relu".
        Le gradient des poids est modifié pour prendre en compte la
        régularisation si besoin.

        Parameters
        ----------
        grad_Oif : Array
            Matrice du gradient de la sortie Oi par rapport à la loss.

        Returns
        -------
        Array
            Matrice du gradient de la sortie Oi par rapport à la loss pour
            la fonction d'activation "relu".
        """
        #Activation
        self.m_grad_Iif = (self.m_Oi>0) * grad_Oif

        #Commun
        self.m_grad_bif =
        np.ones((len(self.m_Xi),1)).T.dot(self.m_grad_Iif)
        self.m_grad_Wif = self.m_Xi.T.dot(self.m_grad_Iif)
        #Regularisation
        if(self.m_taux_regul):
            self.m_grad_bif += self.m_bi * self.m_taux_regul
            self.m_grad_Wif += self.m_Wi * self.m_taux_regul

        self.m_grad_Xif = self.m_grad_Iif.dot(self.m_Wi.T)
        return self.m_grad_Xif

    def relu_init(self,D_Xi,D_Oi):
```

```
        """
        Initialisation des poids ajustée pour la fonction d'activation
        "relu" et "leaky-relu"'

        Parameters
        ----------
        D_Xi : int
            Dimension du vecteur d'entrée de la couche.
        D_Oi : int
            Dimension du vecteur de sortie de la couche.

        Returns
        -------
        None.
        """
        # W est la matrice des poids de chaque sinaps de la couche
        # b est la matrice des bias en entrée
        self.m_Wi = np.random.randn(D_Xi, D_Oi) * np.sqrt(2/D_Xi)
        self.m_bi = np.zeros((1,D_Oi)) + 0.001

    def leaky_relu_forward(self, Xi):
        """
        Applique la fonction leaky relu sur le potentiel d'entrée de la
        couche cachée Ii

        Parameters
        ----------
        Xi : Array
            Matrice d'entrée de la couche'

        Returns
        -------
        Array
            Matrice de sortie Oi pour la fonction d'activation
            "leaky-relu""
        """
        #Commun
        self.m_Xi = Xi
        self.m_Ii = Xi.dot(self.m_Wi) + self.m_bi

        # Activation
        self.m_Oi = np.maximum(self.m_Ii,0) + np.minimum(self.m_Ii*0.01,0)
        return self.m_Oi

    def leaky_relu_backward(self, grad_Oif):
        """
        Calcul les gradients des poids (Wi et bi) et de l'entrée (Xi) à
        partir du gradient de la sortie(Oi) pour la fonction d'activation
        "leaky-relu".
        Le gradient des poids est modifié pour prendre en compte la
        régularisation si besoin.

        Parameters
        ----------
        grad_Oif : Array
            Matrice du gradient de la sortie Oi par rapport à la loss.
```

```
        Returns
        -------
        Array
            Matrice du gradient de la sortie Oi par rapport à la loss pour
            la fonction d'activation "leaky-relu".
        """
        #Activation
        self.m_grad_Iif = ((self.m_Oi>0) + (self.m_Oi<0)*0.01 ) * grad_Oif

        #Commun
        self.m_grad_bif =
        np.ones((len(self.m_Xi),1)).T.dot(self.m_grad_Iif)
        self.m_grad_Wif = self.m_Xi.T.dot(self.m_grad_Iif)
        #Regularisation
        if(self.m_taux_regul):
            self.m_grad_bif += self.m_bi * self.m_taux_regul
            self.m_grad_Wif += self.m_Wi * self.m_taux_regul

        self.m_grad_Xif = self.m_grad_Iif.dot(self.m_Wi.T)
        return self.m_grad_Xif

    def get_regularisation(self):
        """
        Returns
        -------
        float
            Somme des poids au carré de la couche.

        """
        if(self.m_taux_regul):
            return np.square(self.m_Wi).sum()*self.m_taux_regul/2 +
            np.square(self.m_bi).sum()*self.m_taux_regul/2
        else:
            return 0
```

Annexe D main.py

D.1 Imports et définitions

```
import sys,os

dirPath = os.path.dirname(os.path.abspath(__file__))
sys.path.append(dirPath)
import numpy as np
import matplotlib.pyplot as plt

from prepareData import lecture_cifar, decoupage_donnees, rgb2HOG, rgb2LBP
from kppv import kppv_distances, kppv_predict
from reseau import Reseau

def evaluation_classifieur(Ytest,Ypred):
    """
    renvoyantle taux de classification ()
    """
```

```
    Ncorrect = np.count_nonzero(Ytest.flatten()==Ypred.flatten())
    return Ncorrect/len(Ytest)
```

D.2 Lecture et découpage de données

```
X,Y = lecture_cifar(os.path.join(dirPath,"cifar-10-batches-py"), isTest = True)

### Utilisation de descripteurs ?

X = rgb2HOG(X,cellSize=8,cellPerBlock=1)
#X = rgb2LBP(X)

### Découpage des données

Xapp,Yapp,Xtest,Ytest = decoupage_donnees(X, Y)
```

D.3 K plus proches voisins

```
#%%% Calcul de la matrice des distances (Xapp par rapport à Xtest)
Dist = kppv_distances(Xtest, Xapp)

#%%% Influence de k sur l'efficacité du classifieur


liste_k = []
liste_accuracy = []
for k in range(1,101):
    Ypred = kppv_predict(Dist, Yapp, k)
    accu = evaluation_classifieur(Ytest,Ypred)
    liste_accuracy.append(accu)
    liste_k.append(k)

plt.plot(liste_k,liste_accuracy)

#%%% Validation croisée à N répertoires

N = 5

#Division de Xapp en N sous ensemble
foldSize = int(len(Xapp)/N)

liste_k = []
liste_accuracy = []
for n in range(5):
    foldXtest = Xapp[n*foldSize:(n+1)*foldSize]
    foldYtest = Yapp[n*foldSize:(n+1)*foldSize]
    foldXapp = np.concatenate((Xapp[:n*foldSize],Xapp[(n+1)*foldSize:]))
    foldYapp = np.concatenate((Yapp[:n*foldSize],Yapp[(n+1)*foldSize:]))

    foldDist = kppv_distances(foldXtest, foldXapp)

    liste_k.append([])
    liste_accuracy.append([])
    for k in range(1,101):
        foldYpred = kppv_predict(foldDist, foldYapp, k)
        accu = evaluation_classifieur(foldYtest,foldYpred)
```

```
        liste_accuracy[n].append(accu)
        liste_k[n].append(k)
```

D.4 Réseau de neurone

```
loss_list = []
accu_list = []
accu_test_list = []

N = 45000
mini_batch_size = 45
N_mini_batch_per_epoch = int(N/mini_batch_size)
N_epoch = 100

Xnn = Xapp[:N]
Ynn = np.zeros((N,10))
Ynn[np.arange(N),Yapp[:N].flatten()]=1

N_test = 5000
Xnn_test = Xtest[:N_test]
Ynn_test = np.zeros((N_test,10))
Ynn_test[np.arange(N_test),Ytest[:N_test].flatten()]=1

Xnn_minibatches = [Xnn[k*mini_batch_size:(k+1)*mini_batch_size] for k in
range(N_mini_batch_per_epoch)]
Ynn_minibatches = [Ynn[k*mini_batch_size:(k+1)*mini_batch_size] for k in
range(N_mini_batch_per_epoch)]

D_in, D_out = Xnn.shape[1], 10
D_layers = [512,256]

learning_rate = 1e-3

reseau = Reseau(D_in, D_layers, D_out, activation='leaky_relu', taux_regul = 0.01)

for ne in range(10):
    for nmb in range(N_mini_batch_per_epoch):
        Xnn = Xnn_minibatches[nmb]
        Ynn = Ynn_minibatches[nmb]

        ######################################################
        # Passe avant : calcul de la sortie prédite Y_pred #
        ######################################################
        Y_pred = reseau.forward(Xnn)
        ###############################################
        # Calcul et affichage de la fonction perte #
        ###############################################

        #MSE loss
        #loss = np.square(Y_pred - Ynn).sum() / (2*mini_batch_size)
        #grad_Y_pred = (Y_pred - Ynn) / mini_batch_size

        #Cross entropy loss
        Y_pred_exp = np.exp(Y_pred)
        Y_pred = (Y_pred_exp.T/np.sum(Y_pred_exp,axis=1)).T
        loss = - np.sum( Ynn*np.log(Y_pred) ) / mini_batch_size
```

```
        grad_Y_pred = (Ynn * (Y_pred - 1) + (1-Ynn) * Y_pred) / mini_batch_size

        #regularissation
        if(True):
            loss += reseau.get_regularisation()

        loss_list.append(loss)
        accu=evaluation_classifieur(np.argmax(Ynn,axis=1),np.argmax(Y_pred,axis=1))
        accu_list.append(accu)
        if(nmb%10==0):
            print(str(ne)+"\t"+str(loss)+"\t"+str(accu))
        ######################################################
        # Passe arrière : calcul des gradients des Wi et bi #
        ######################################################
        reseau.backward(grad_Y_pred)
        ##########################
        # Mise à jour de poids #
        ##########################
        reseau.update_weights(learning_rate)


    Y_pred_test = reseau.forward(Xnn_test)
    accu_test=evaluation_classifieur(np.argmax(Ynn_test,axis=1),np.argmax(Y_pred_test,axis=1))
    accu_test_list.append(accu_test)
    if(ne%1==0):
        print(accu_test)
```