

Akilhoussen ONALY

S9 MOD Apprentissage profond & Intelligence Artificielle : une introduction

2021-2022

TD1 – Kppv et réseaux de neurones pour la classification d'images

Compte rendu

Table des matières

I- Introduction.....	1
II- Structure du code	1
III- Système de classification à base de l'algorithme des k plus proches voisins	2
1) Développement du classifieur.....	2
2) Expérimentations.....	6
IV- Système de classification à base de réseaux de neurones.....	9
1) Développement du classifieur.....	9
2) Expérimentations.....	13
V- Conclusion	20

I- Introduction

L'objectif de ce TD est de réaliser une classification d'images en utilisant deux algorithmes différents : en premier, l'algorithme des k plus proches voisins et en second, des réseaux de neurones classiques. Les données proviennent du dataset CIFAR-10 accessible à l'adresse <https://www.cs.toronto.edu/~kriz/cifar.html> et qui contient 60000 images en couleur de taille 32x32 réparties en 10 classes (avion, voiture, oiseau, chat, ...).

II- Structure du code

Le code est structuré de la façon suivante (des mentions seront faites à certains fichiers dans la suite du compte rendu) :

```
|— data <- Contient les données (les mettre dans ce dossier pour tester le code)
|   |— data_batch_1
|   |— data_batch_2
|   |— data_batch_3
```

```
|   ├── data_batch_4
|   ├── data_batch_5
|   └── test_batch
|── src          <- Contient le code source du projet
|   ├── __init__.py  <- Pour que src soit un module
|   ├── utils.py    <- Contient les fonctions utilitaires (lecture, découpage, ...)
|   ├── kppv.py     <- Contient les fonctions propres à l'algorithme kppv
|   ├── kppv_script.py <- Contient le script pour tester l'algorithme kppv
|   ├── nn_functions.py <- Contient les fonctions propres aux réseaux de neurones
|   └── nn_script.py <- Contient le script pour tester les réseaux de neurones
```

Remarque : ce projet a été développé avec Python 3.9

III- Système de classification à base de l'algorithme des k plus proches voisins

Dans cette partie, nous allons développer l'algorithme des k plus proches voisins et l'évaluer sur notre jeu de données. Le principe de cet algorithme de classification est de prédire la classe d'un nouvel élément en prenant la classe la plus fréquente parmi les classes des k éléments les plus proches.

Nous commençons tout d'abord par implémenter les différentes fonctions utiles.

1) Développement du classifieur

Fonction *lecture_cifar* (cf. *utils.py*) :

Afin de créer cette fonction, nous créons d'abord une fonction *unpickle* qui va lire un fichier contenant un dictionnaire et retourne ce dictionnaire (cette fonction est donnée aussi sur la page web du dataset CIFAR-10) :

```
1. def unpickle(file):
2.     """ Fonction qui dé-séréalise un fichier sauvegardé localement
3.     et retourne un dictionnaire
4.
5.     Entrée(s) :
6.     - file (str) : chemin du fichier
7.
8.     Sortie(s):
9.     - dict (dict) : dictionnaire stocké dans le fichier """
10.
11.     with open(file, 'rb') as fo:
12.         dict = pickle.load(fo, encoding='bytes')
13.     return dict
```

Ensuite, nous pouvons implémenter la fonction *lecture_cifar*. Pour cela, nous lisons les différents fichiers à l'aide de la fonction *unpickle* et ensuite nous récupérons les tableaux contenant les données d'entrée ainsi que les labels. Notons que le jeu de données est divisé en 5 batches et un batch supplémentaire de test mais étant donné que nous allons réaliser nous-mêmes la division en sous-ensemble d'apprentissage et test, nous concaténons tous les tableaux pour avoir un jeu de données unique. Les données d'entrée sont converties en type *float32* et les labels en type *int* car ces

derniers représentent la classe pour chaque échantillon (entre 0 et 9). Par ailleurs, on normalise les données d'entrée pour que celles-ci se trouvent entre 0 et 1 plutôt qu'entre 0 et 255.

Voici alors la fonction correspondante :

```
1. def lecture_cifar(path):
2.     """ Fonction qui lit les données du dataset CIFAR-10 et retourne les tableaux X et Y
3.     X est normalisé (divisé par 255)
4.
5.     Entrée(s) :
6.     - path (str) : chemin du dossier contenant les données
7.
8.     Sortie(s) :
9.     - X,Y (np.array,np.array) : tableaux Numpy 2D contenant les données et les labels
10.    """
11.    data_list, labels_list = [], []
12.    for k in range(1,6): #lecture des batches 1 à 5
13.        path_data_batch_k = f"{path}/data_batch_{k}"
14.        dict_k = unpickle(path_data_batch_k)
15.        data_k = np.array(dict_k[b'data']).astype('float32')
16.        # Le reshape sert à ajouter une dimension supplémentaire :
17.        labels_k = np.array(dict_k[b'labels']).astype('int').reshape(-1,1)
18.        data_list.append(data_k)
19.        labels_list.append(labels_k)
20.    # Lecture du batch test :
21.    last_path = f"{path}/test_batch"
22.    last_dict = unpickle(last_path)
23.    last_data = np.array(last_dict[b'data']).astype('float32')
24.    last_labels = np.array(last_dict[b'labels']).astype('int').reshape(-1,1)
25.    data_list.append(last_data)
26.    labels_list.append(last_labels)
27.
28.    # Concaténation et normalisation :
29.    X = np.concatenate(data_list,axis = 0)
30.    X = X / 255
31.    Y = np.concatenate(labels_list,axis = 0)
32.
33.    return X, Y
```

Fonction *decoupage_donnees* (cf. utils.py) :

Cette fonction sert à découper aléatoirement les tableaux retournés par la fonction *lecture_cifar* en sous-ensembles d'apprentissage et de test. Pour cela, nous créons des indices allant jusqu'à la taille du jeu de données entier et ensuite nous les mélangeons. Nous prenons ensuite les premiers 80 % de ces indices pour constituer le jeu d'apprentissage et les 20 % restants pour constituer le jeu de test.

Voici alors la fonction correspondante :

```
1. def decoupage_donnees(X,Y):
2.
3.     """ Fonction qui découpe les tableaux retournés par la fonction lecture_cifar
4.     en jeux d'apprentissage et de test.
5.
6.     Entrée(s):
7.     - X, Y (np.array, np.array) : données d'entrée et labels
8.
9.     Sortie(s):
10.    - Xapp, Yapp, Xtest, Ytest (tous np.array): jeux d'apprentissage et de test découpés aléatoirement
11.    """
12.
13.    train_percentage = 0.8 # pourcentage du découpage
14.
15.    np.random.seed(0) # pour rendre le découpage déterministe
16.    n = X.shape[0] # taille du jeu de données original
17.    idxs = np.arange(n) # génération d'indices
18.    np.random.shuffle(idxs) # mélange aléatoire des indices
19.    n_train = int(round(train_percentage * n)) # taille du jeu de données d'apprentissage
20.    idx_train = idxs[:n_train] # on récupère les n_train premiers indices
```

```

21.     idx_test = idxs[n_train:] # les indices restants servent pour le test
22.     Xapp, Yapp = X[idx_train:], Y[idx_train:]
23.     Xtest, Ytest = X[idx_test:], Y[idx_test:]
24.
25.     return Xapp, Yapp, Xtest, Ytest

```

Fonction *kppv_distances* (cf. *utils.py*) :

Afin de créer la fonction qui va calculer la distance euclidienne entre chaque élément du jeu de test et chaque élément du jeu d'apprentissage, nous allons utiliser les propriétés de *broadcasting* de Numpy. Cela nous permettra de faire le calcul sans faire de boucle et sans devoir redimensionner les tableaux car si (n1,m) sont les dimensions de Xtest et (n2,m) les dimensions de Xapp alors nous savons que le résultat doit être de dimension (n1,n2) où chaque ligne représente un élément de Xtest et chaque valeur de cette ligne est la distance de cet élément avec un élément de Xapp. La formule suivante¹ permet alors d'obtenir le résultat :

$$\text{dist}(X_{\text{test}}, X_{\text{app}}) = \sqrt{\|X_{\text{test}}\|^2 + (\|X_{\text{app}}\|^2)^T - 2 \cdot X_{\text{test}} \cdot X_{\text{app}}^T}$$

En effet, le terme en rouge sera de dimension (n1,1) et le terme en vert sera de dimension (1,n2) et leur somme sera de dimension (n1,n2) grâce au *broadcasting* de Numpy. Le terme en bleu qui est un produit matriciel d'une matrice (n1,m) par une matrice (m,n2) sera de dimension (n1,n2) également donc le résultat final est bien de dimension (n1,n2) comme attendu.

Voici alors l'implémentation correspondante :

```

1.  def kppv_distances(Xapp,Xtest):
2.
3.      """ Fonction qui permet de calculer la distance euclidienne entre Xapp et Xtest
4.
5.      Entrée(s):
6.      - Xapp, Xtest (np.array, np.array) : jeux d'apprentissage et test
7.
8.      Sortie(s):
9.      - dist (np.array) : matrice de distance
10.     """
11.
12.     el1 = np.power(Xtest,2).sum(axis = 1,keepdims = True) #premier terme
13.     el2 = np.power(Xapp,2).sum(axis = 1,keepdims = True).T #deuxième terme
14.     el3 = 2*np.dot(Xtest,Xapp.T) #troisième terme
15.
16.     dist = np.sqrt(el1 + el2 - el3) #résultat
17.
18.     return dist

```

Fonction *kppv_predict* (cf. *kppv.py*) :

Afin de prédire les classes pour Xtest, nous trions tout d'abord la matrice de distance selon la deuxième dimension pour récupérer les indices des éléments de Xapp dans l'ordre croissant de distance. Cela peut être réalisé avec la fonction *argsort* de numpy. Cependant, il s'avère que comme souhaite récupérer uniquement les indices des k plus proches éléments pour ensuite trouver la classe la plus fréquente, nous n'avons pas besoin de l'ordre entre ces k éléments, c'est-à-dire que les k plus proches éléments peuvent être dans n'importe quel ordre et pas dans l'ordre qui permet de trier la matrice de distance. La fonction *argpartition* qui est

¹ La formule peut ne pas être valide mathématiquement mais il s'agit d'une traduction du calcul qui sera réalisé sur Python

plus rapide que *argsort* permet de faire cela. Cependant, cette implémentation n'est pas satisfaisante car si la fonction *argpartition* peut être plus rapide pour un *k* donné, si nous souhaitons la prédiction pour plusieurs valeurs de *k*, le calcul peut être redondant dans la mesure où si la matrice de distance est triée une fois pour toute, nous n'avons pas besoin de refaire le calcul. Nous fournissons donc à la fonction un argument optionnel qui permet de fournir les indices de tri de la matrice de distance. Une fois que nous avons les indices, nous pouvons ensuite trouver la classe la plus fréquente.

Voici l'implémentation correspondante, des détails sont fournis dans les commentaires :

```
1. def kppv_predict(dist,Yapp,k,argsort = np.array([])):
2.
3.     """ Fonction qui permet de retourner les classes prédites pour les éléments de Xtest
4.
5.     Entrée(s):
6.     - dist (np.array) : matrice de distance
7.     - Yapp (np.array) : labels d'apprentissage
8.     - k (int) : nombre de voisins
9.     - argsort (np.array) (Optionnel) : indices qui permettent de trier la matrice de distance
10.
11.     Sortie(s):
12.     - Ypred (np.array) : classes prédites pour Xtest
13.     """
14.
15.     if argsort.size != 0: # si les indices de tri sont fournis
16.         idx = argsort[:, :k] # indices des k plus proches éléments dans Yapp
17.     else: # si les indices de tri ne sont pas fournis
18.         idx = np.argpartition(dist, k, axis=1)[:, :k] # indices des k plus proches éléments dans Yapp
19.
20.     labels = Yapp[idx, :] # labels des k plus proches éléments de Yapp
21.
22.     axis = 1
23.
24.     # La ligne suivante permet de calculer les classes uniques dans "labels"...
25.     # et les indices qui permettent de reconstruire "labels" sachant que le résultat...
26.     # est aplati (flattened) :
27.     u, indices = np.unique(labels, return_inverse=True)
28.
29.     # La ligne suivante permet de compter selon la deuxième dimension l'occurrence ...
30.     # de chaque classe avec la fonction bincount de numpy à laquelle ...
31.     # on spécifie le nombre de bins (nombre de classes) ...
32.     # ensuite, on applique cette opération selon la 2ème dimension ...
33.     # avec la fonction apply_along_axis de numpy ...
34.     # à la matrice des indices calculée précédemment ...
35.     # qu'on redimensionne pour qu'elle ait les mêmes dimensions que "labels" ...
36.     # Cela permet d'avoir pour chaque ligne de Xtest, les différentes classes (leurs indices plus
    précisément)...
37.     # et leur nombre d'occurrences, on récupère ensuite l'indice de la classe la plus fréquente ...
38.     # avec la fonction argmax et on récupère la classe correspondante avec u :
39.
40.     nb_bins = np.max(indices) + 1
41.     Ypred = u[np.argmax(np.apply_along_axis(np.bincount, axis, indices.reshape(labels.shape),
42.                                         minlength = nb_bins), axis=axis)]
43.
44.     return Ypred
```

Fonction *evaluation_classifieur* (cf. *utils.py*) :

Voici le code de la fonction correspondante qui est suffisamment explicite :

```
1. def evaluation_classifieur(Ytest,Ypred):
2.
3.     """ Fonction qui permet d'avoir la précision de la prédiction
4.     (accuracy)
5.
6.     Entrée(s):
7.     - Ytest,Ypred (np.array,np.array)
8.
9.     Sortie(s):
10.    - accuracy (float) : la précision
11.    """
12.
13.    error = Ytest - Ypred #différence entre Ytest et Ypred
```

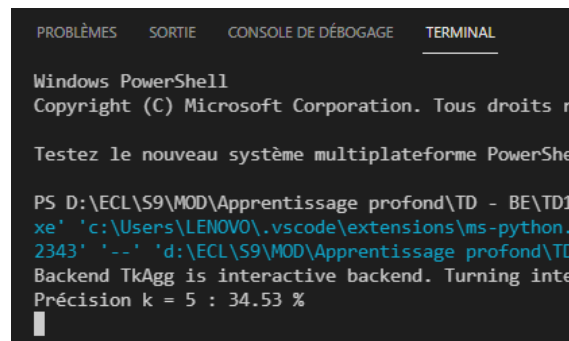
```
14.     binary = (error != 0).astype('int') #1 si la différence est non nulle, 0 sinon
15.     sum = binary.sum(axis = 0).sum() # somme du résultat précédent
16.     # On calcule ensuite le taux d'erreur en divisant par la longueur de Ytest...
17.     # et la précision est juste 1 - le taux d'erreur et on multiplie par 100 ...
18.     # pour avoir un pourcentage :
19.     accuracy = 100 * (1 - sum / len(Ytest))
20.
21.
22.     return accuracy
```

2) Expérimentations

Évaluation pour une valeur de k :

Une fois que toutes les fonctions sont implémentées, nous pouvons réaliser des expérimentations avec notre jeu de données. Le script commenté se trouve dans *src/kppv_script.py*.

Pour $k = 5$, nous obtenons une précision de 34.53 % comme visible sur la capture suivante :



```
PROBLÈMES  SORTIE  CONSOLE DE DÉBOGAGE  TERMINAL
Windows PowerShell
Copyright (C) Microsoft Corporation. Tous droits réservés.

Testez le nouveau système multiplateforme PowerShell.

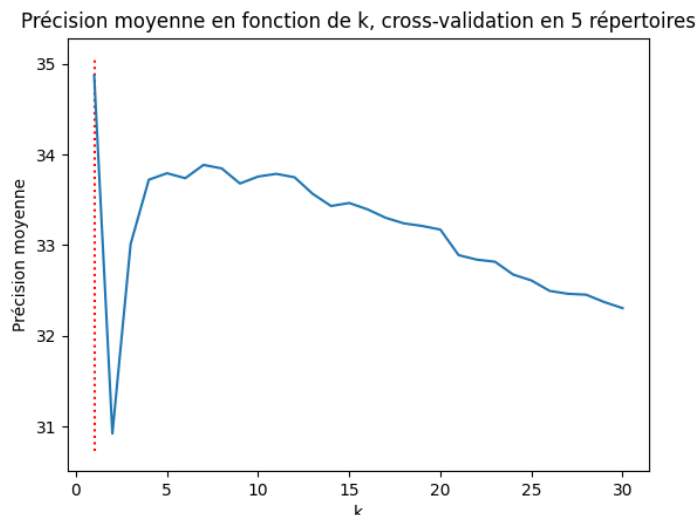
PS D:\ECL\S9\MOD\Apprentissage profond\TD - BE\TD1...
xe' 'c:\Users\LENOVO\.vscode\extensions\ms-python.p
2343' '--' 'd:\ECL\S9\MOD\Apprentissage profond\TD
Backend TkAgg is interactive backend. Turning inter
Précision k = 5 : 34.53 %
```

Choix de la meilleure valeur de k par cross-validation :

À présent, nous allons réaliser une **cross-validation** en **5 répertoires** pour trouver le meilleur k à choisir et étudier l'influence de sa valeur sur la performance. Nous aurions pu le faire directement sur le découpage initial mais comme il s'agit d'un découpage parmi d'autres, il se peut qu'il y ait une dépendance à ce découpage donné. La cross-validation permet alors d'éliminer en cette dépendance pour que le modèle ait la meilleure capacité de généralisation.

L'opération consiste à diviser le jeu de données en 5 parties égales et ensuite à chaque fois, on utilise 1 partie pour le test et les 4 restantes pour l'apprentissage. La métrique utilisée sera la précision moyenne obtenue sur l'ensemble des 5 itérations. Notons que cette opération est coûteuse en temps et en mémoire (cela prend plus d'une dizaine de minutes) pour notre jeu de données qui est de grande taille et donc nous avons dû ajouter une suppression explicite de variables dans les boucles pour libérer la mémoire.

Voici la figure obtenue :



Nous constatons d'après la figure précédente que la meilleure valeur de k est k = 1 où on atteint quasiment 35 %, ce qui signifie en quelque sorte que rajouter des images supplémentaires ne fait que rajouter du bruit à la comparaison. Par ailleurs, vu l'allure de la courbe, nous pouvons faire l'hypothèse que la précision va continuer à décroître au-delà de k = 30.

Représentation des images par les descripteurs HOG :

À présent, essayons d'utiliser des descripteurs au lieu d'utiliser les images directement. Pour cela, nous allons utiliser les descripteurs basés sur les histogrammes de gradients orientés (HOG) à l'aide la fonction *hog* de *scikit-image*. On crée donc une fonction *hog_features* dans *utils.py* qui va calculer ces descripteurs en prenant en entrée le tableau X. Voici le code correspondant avec des commentaires (dans *utils.py*) :

```

1. def hog_features(X):
2.
3.     """ Fonction qui calcule les descripteurs HOG des images contenues dans X
4.
5.     Entrée(s):
6.     - X (np.array) : tableau des images
7.
8.     Sortie(s):
9.     - hog_features (np.array) : tableau des descripteurs pour les images de X
10.    """
11.
12.    m = X.shape[0]
13.    list_im = []
14.    for k in tqdm(range(m)):
15.        R = X[k,:1024].reshape(32,32,1) # canal rouge de l'image
16.        G = X[k,1024:1024*2].reshape(32,32,1) # canal vert de l'image
17.        B = X[k,1024*2:].reshape(32,32,1) # canal bleu de l'image
18.        full_im = np.concatenate((R,G,B),axis = -1)
19.        fd = hog(full_im, orientations=10, pixels_per_cell=(16, 16),
20.                cells_per_block=(1, 1), visualize=False, multichannel = True)
21.
22.        list_im.append(fd)
23.        del R,G,B,full_im,fd #suppression des variables pour libérer de la mémoire
24.
25.    hog_features = np.array(list_im).astype('float32')
26.
27.    return hog_features

```

Notons que les paramètres de la fonction *hog* ont été spécifiés directement dans la fonction mais pour choisir ces paramètres, il conviendrait de les spécifier en entrée de la fonction. Ces paramètres pourraient alors être choisis par cross-validation comme cela a été fait pour k.

Pour $k = 5$, nous obtenons une précision de 29.62 % comme visible sur la capture suivante :

```
PROBLÈMES  SORTIE  CONSOLE DE DÉBOGAGE  TERMINAL

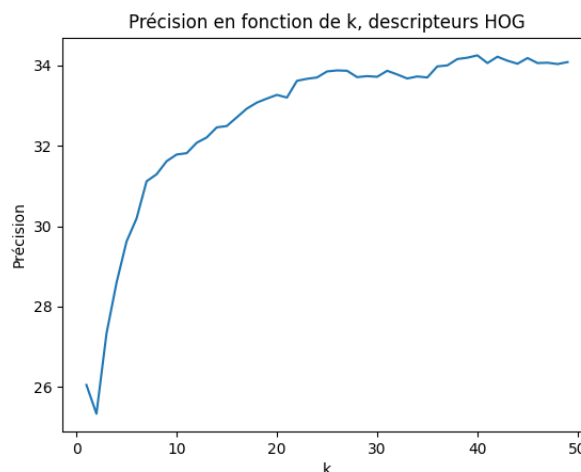
\MOD\Apprentissage profond\TD - BE\TD1\src\kppv_
100%|
Backend TkAgg is interactive backend. Turning in

D:\ECL\S9\MOD\Apprentissage profond\TD - BE\TD1>
1" && cmd /C "C:\Users\LENOVO\AppData\Local\Prog
xtensions\ms-python.python-2021.10.1365161279\py
\MOD\Apprentissage profond\TD - BE\TD1\src\kppv_
100%|
Backend TkAgg is interactive backend. Turning in
Précision HOG k = 5 : 29.62 %

D:\ECL\S9\MOD\Apprentissage profond\TD - BE\TD1>
```

On constate donc que la représentation des images par des descripteurs n'améliore pas la performance (du moins pour $k = 5$). Cela est compréhensible car les images sont de basse résolution (32 x 32) et donc il est vraisemblable que les descripteurs ne soient pas aussi précis qu'on le voudrait. Afin de s'assurer que nous ne sommes pas dans un cas particulier, on peut tester la performance pour plusieurs valeurs de k .

Voici la courbe que nous obtenons :



On constate donc que les performances s'améliorent quand k augmente même si on est dans le même ordre de grandeur que précédemment. On pourrait donc préférer cette méthode qui est moins couteuse en temps de calcul car on passe de 3072 à variables à 40 pour chaque image. La meilleure valeur de k serait alors choisie par cross-validation comme précédemment.

Représentation des images par des descripteurs LBP :

Enfin, on représente les images par des descripteurs LBP (Local Binary Patterns). On crée pour cela une fonction `lbp_features` (dans `utils.py`) similaire à `hog_features`, la seule différence est que pour calculer les descripteurs basés sur les LBP, on doit donner en argument une image en niveaux de gris et le résultat est au même format que l'entrée donc on doit l'aplatir :

```
1. def lbp_features(X):
2.
3.     """ Fonction qui calcule les descripteurs LBP des images contenues dans X
4.
```



```

5.     Entrée(s):
6.     - X (np.array) : tableau des images
7.
8.     Sortie(s):
9.     - lbp_features (np.array) : tableau des descripteurs LBP pour les images de X
10.    """
11.    m = X.shape[0]
12.    list_im = []
13.    for k in tqdm(range(m)):
14.        R = X[k,:1024].reshape(32,32,1) # canal rouge de l'image
15.        G = X[k,1024:1024*2].reshape(32,32,1) # canal vert de l'image
16.        B = X[k,1024*2:].reshape(32,32,1) # canal bleu de l'image
17.        full_im = np.concatenate((R,G,B),axis = -1)
18.        grayscale_im = rgb2gray(full_im)
19.        fd = local_binary_pattern(grayscale_im,10,0.1).flatten()
20.        list_im.append(fd)
21.        del R,G,B,full_im,grayscale_im,fd #suppression des variables pour libérer de la mémoire
22.
23.    lbp_features = np.array(list_im).astype('float32')
24.
25.    return lbp_features

```

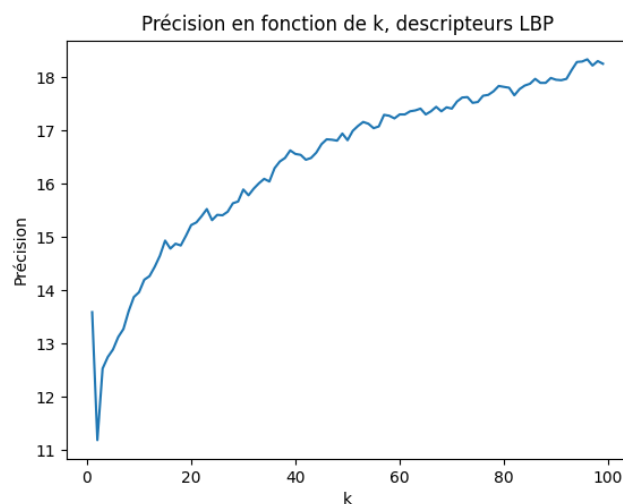
Pour $k = 5$, nous obtenons une précision de 12.89 % comme visible sur la capture suivante :

```

D:\ECL\S9\MOD\Apprentissage profond\TD - BE\TD1
/ECL/S9/MOD/Apprentissage profond/TD - BE/TD1/s
100%|
Précision LBP k = 5 : 12.89 %

```

Testons à présent pour d'autres valeurs de k et voyons comment évolue la précision, voici la courbe obtenue :



Nous observons sur la figure précédente que les performances s'améliorent quand k augmente mais nous sommes loin des performances précédentes. Il semble donc que nous ayons perdu beaucoup d'information en extrayant ces descripteurs LBP.

IV- Système de classification à base de réseaux de neurones

Dans cette partie, nous allons implémenter un réseau de neurones pour réaliser la classification des images.

1) Développement du classifieur

Afin de développer le classifieur, nous allons tout d'abord développer plusieurs fonctions qui serviront de base. En premier arrivent les fonctions mathématiques qui interviennent dans le réseau telles que les fonctions d'activation, les fonctions de perte et éventuellement leurs dérivées. Voici le code commenté de toutes ces fonctions (dans *nn_functions.py*) :

```
1. import numpy as np
2.
3. def sigmoide(x):
4.     """ Fonction sigmoide """
5.     return 1/(1+np.exp(-x))
6.
7. def relu(x):
8.     """ Fonction relu """
9.     return np.maximum(x,0)
10.
11. def sortie_lineaire(X,W,b):
12.     """ Fonction qui calcule la partie linéaire d'un neurone """
13.     #On vérifie les dimensions :
14.     assert (X.shape[1] == W.shape[0] and b.shape == (1,W.shape[1])), "Les dimensions ne correspondent pas"
15.
16.     return X.dot(W) + b
17.
18. def derivee_sigmoide(x):
19.     """ Dérivée sigmoide """
20.     return sigmoide(x) * (1 - sigmoide(x))
21.
22. def derivee_relu(x):
23.     """ Dérivée relu """
24.     return np.where(x <= 0, 0, 1)
25.
26. def softmax(x):
27.     """ Fonction softmax """
28.     return np.exp(x) / np.exp(x).sum(axis = 1,keepdims = True)
29.
30. def sortie_activee(X,W,b,activation):
31.     """ Fonction qui calcule la sortie d'un neurone en spécifiant l'activation choisie """
32.     s = sortie_lineaire(X,W,b)
33.
34.     if activation == 'relu':
35.         return relu(s), s
36.     elif activation == 'softmax':
37.         return softmax(s), s
38.     elif activation == 'sigmoide':
39.         return sigmoide(s), s
40.
41. def mse(Y_pred,Y):
42.     """ Fonction de perte mse """
43.     res = np.square(Y_pred - Y).sum() / 2
44.     return res
45.
46. def grad_mse(Y_pred,Y):
47.     """ Gradient de la fonction de perte mse par rapport à Y_pred """
48.     res = Y_pred - Y
49.     return res
50.
51. def cross_entropy(Y_pred,Y):
52.     """ Fonction de perte cross_entropy """
53.     Ypred = Y_pred.clip(min=1e-8,max=None) #pour éviter de prendre log(0)
54.     res = (-Y*np.log(Ypred)).sum()
55.     return res
56.
```

Nous développons ensuite une fonction qui sert à initialiser les paramètres. Cette fonction prend en entrée la liste des dimensions des couches du réseau (entrée et sortie incluses) et retourne un dictionnaire où l'on pourra accéder aux paramètres avec des clés de type 'W1', 'W2', ... pour les poids et 'b1', 'b2', ... pour les biais. Par ailleurs, nous avons fait le choix de faire une initialisation qui dépend de l'activation pour les poids. En effet, dans nos expériences préliminaires, l'initialisation uniforme entre -1 et 1 fonctionne mieux quand la fonction d'activation est sigmoide et l'initialisation avec une loi normale centrée en 0 et de faible variance fonctionne mieux pour l'activation RELU. Pour les poids, nous les initialisons tous par 0. Nous vérifions également que les dimensions correspondent bien avec le format attendu.

Voici le code commenté de la fonction correspondante (dans *nn_functions.py*) :

```
1. def initialiser_parametres(liste_dims,activation):
2.     """ Fonction pour l'initialisation des paramètres
3.
4.     Entrée(s):
5.     - liste_dims : liste des dimensions des différentes couches du réseau
6.
7.     Sortie(s):
8.     - parametres (dict) : dictionnaire contenant les paramètres
9.     """
10.    np.random.seed(1) # pour rendre la génération aléatoire déterministe à chaque exécution
11.    parametres = {}
12.    L = len(liste_dims)
13.
14.    for l in range(1, L):
15.        if activation == 'relu': # si la fonction d'activation est relu
16.            parametres['W'+str(l)] = (1e-2) * np.random.randn(liste_dims[l-1], liste_dims[l]) # loi normale
17.            de faible variance et centrée en 0
18.        elif activation == 'sigmoide': # si la fonction d'activation est sigmoide
19.            parametres['W'+str(l)] = 2 * np.random.random((liste_dims[l-1], liste_dims[l])) - 1 # loi
20.            uniforme entre -1 et 1
21.            parametres['b'+str(l)] = np.zeros((1,liste_dims[l])) #initialisation par 0
22.            assert(parametres['W' + str(l)].shape == (liste_dims[l-1], liste_dims[l])) #vérification des
23.            dimensions
24.            assert(parametres['b' + str(l)].shape == (1,liste_dims[l]))
25.
26.    return parametres
```

Enfin, nous pouvons implémenter les fonctions d'apprentissage et de prédiction de réseau. La fonction d'apprentissage consistera à entraîner le réseau dont les hyperparamètres et l'architecture seront spécifiés et cette fonction retournera en sortie les poids et biais appris. À l'intérieur de cette fonction, certains choix ont été faits pour adapter le réseau à notre problème qui est une classification multi-classes :

- La fonction d'activation finale est choisie en fonction de la perte. En effet, si la fonction de perte est l'erreur quadratique alors une fonction sigmoide suffit car le réseau va apprendre à minimiser l'écart entre les prédictions et les labels donc il suffit que les prédictions soient entre 0 et 1. En revanche, si la fonction de perte est l'entropie croisée, alors nous sommes obligés de choisir la fonction *softmax* pour l'activation finale car l'entropie croisée est en général réservée aux problèmes de classification multi-classes or là une fonction sigmoide se contente de borner les sorties dans [0,1] et correspond davantage à une classification multi-label, la fonction *softmax* permet de s'assurer que les prédictions pour chaque classe soient de somme 1 pour un échantillon donné, ce qui correspond donc à des probabilités.
- La régularisation choisie est la régularisation L2. Il n'y a pas de raison particulière d'utiliser cette régularisation pour la classification multi-classes mais ce choix a été fait pour simplifier l'expérimentation.

Voici alors le code commenté² de cette fonction (dans *nn_functions.py*) :

```
1. def apprentissage_reseau(
2.     X,
3.     Y,
4.     activation,
5.     taille_batch,
6.     alpha,
```

² Il est supposé dans ce document que le lecteur connaît les formules de la rétropropagation du gradient, elles ne sont pas détaillées ici pour faciliter la lecture

```

7.     liste_dims,
8.     nb_iterations,
9.     coeff_l2,
10.    perte
11. ):
12.
13.     """ Fonction qui fait apprendre un réseau de neurones avec des paramètres spécifiés en entrée
14.     et retourne les poids et biais appris
15.
16.     Entrée(s):
17.     - X (np.array) : tableau 2D des données d'entrée
18.     - Y (np.array) : tableau 2D des données de sortie (labels)
19.     - activation (str) : fonction d'activation pour les couches cachées
20.     - taille_batch (int) : taille du mini-batch pour la descente du gradient
21.     - alpha (float) : taux d'apprentissage
22.     - liste_dims (list) : liste des dimensions des couches du réseau (entrée et sortie incluses)
23.     - nb_iterations (int) : nombre d'itérations pour la descente du gradient (epochs)
24.     - coeff_l2 (float) : coefficient de régularisation L2
25.     - perte (str) : fonction de perte à utiliser
26.     """
27.
28.     params = initialiser_parametres(liste_dims,activation) # dictionnaire des paramètres initialisés
29.     nb_batches = int(X.shape[0]/taille_batch) # nombre de batches pour la descente du gradient
30.     nb_couches = len(liste_dims) - 1 # nombre de couches du réseau (entrée exclue et sortie incluse)
31.     idxs = list(range(X.shape[0])) #indices de X
32.     np.random.seed(1) # Pour rendre le calcul déterministe
33.     np.random.shuffle(idxs) # Mélange aléatoire des indices (pour choisir les batches aléatoirement)
34.     # Choix de l'activation finale en fonction de la perte choisie :
35.     if perte == 'mse':
36.         activation_sortie = 'sigmoide'
37.     elif perte == 'cross_entropy':
38.         activation_sortie = 'softmax'
39.     for k in range(1,nb_iterations + 1):
40.         loss = 0
41.         for i in range(nb_batches): # itération sur les batches
42.             vars = {} # dictionnaire pour stocker les variables
43.             idx_batch = idxs[i*taille_batch:(i+1)*taille_batch] # indice des batches
44.             vars['a0'] = X[idx_batch,:] # calcul du batch d'entrée
45.             Yi = Y[idx_batch,:] # calcul du batch de sortie
46.             # Phase de propagation en avant :
47.             for j in range(nb_couches-1):
48.                 vars[f'a{j+1}'],vars[f's{j+1}'] = sortie_activee(
49.                     vars[f'a{j}'],
50.                     params[f'W{j+1}'],
51.                     params[f'b{j+1}'],
52.                     activation
53.                 )
54.             vars[f'a{nb_couches}'],vars[f's{nb_couches}'] = sortie_activee(
55.                 vars[f'a{nb_couches-1}'],
56.                 params[f'W{nb_couches}'],
57.                 params[f'b{nb_couches}'],
58.                 activation_sortie
59.             )
60.             Y_pred = vars[f'a{nb_couches}'] # Les valeurs prédites sont les sorties de la couche de sortie
61.
62.             loss += globals()[perte](Y_pred,Yi) # calcul (mise à jour) de la perte
63.
64.             # Phase de rétropropagation du gradient :
65.             if perte == 'mse':
66.                 grad_Y_pred = grad_mse(Y_pred,Yi)
67.                 vars[f'grad_s{nb_couches}'] = grad_Y_pred * derivee_sigmoide(vars[f's{nb_couches}'])
68.             elif perte == 'cross_entropy':
69.                 vars[f'grad_s{nb_couches}'] = (Y_pred - Yi)
70.             vars[f'grad_W{nb_couches}'] = vars[f'a{nb_couches-1}'].T.dot(vars[f'grad_s{nb_couches}']) + 2 *
coeff_l2 * params[f'W{nb_couches}']
71.             vars[f'grad_b{nb_couches}'] = np.sum(vars[f'grad_s{nb_couches}'],axis = 0,keepdims = True)
72.             for j in range(nb_couches-1,0,-1): # itération en arrière sur les couches
73.                 vars[f'grad_a{j}'] = vars[f'grad_s{j+1}'].dot(params[f'W{j+1}']).T
74.                 vars[f'grad_s{j}'] = vars[f'grad_a{j}'] * globals()[f'derivee_{activation}'](vars[f's{j}'])
75.                 vars[f'grad_W{j}'] = vars[f'a{j-1}'].T.dot(vars[f'grad_s{j}']) + 2 * coeff_l2 *
params[f'W{j}']
76.                 vars[f'grad_b{j}'] = np.sum(vars[f'grad_s{j}'],axis = 0,keepdims = True)
77.
78.             # Mise à jour des poids et des biais
79.             for j in range(nb_couches):
80.                 params[f'W{j+1}'] -= alpha * vars[f'grad_W{j+1}']
81.                 params[f'b{j+1}'] -= alpha * vars[f'grad_b{j+1}']
82.             for j in range(nb_couches): #Ajout du terme de régularisation à la loss
83.                 loss += coeff_l2 * np.power(params[f'W{j+1}'],2).sum()
84.             print(f'Itération {k} : {loss:.5f}') # Affichage de la loss à chaque epoch
85.
86.     return params

```

La fonction de prédiction, quant à elle, prendra en argument les poids et biais appris et retournés par la fonction précédente (en plus d'autres hyperparamètres) afin de prédire les classes pour les données d'entrée.

Voici le code commenté de cette fonction (dans *nn_functions.py*) :

```
1. def prediction_reseau(
2.     X,
3.     liste_dims,
4.     activation,
5.     params
6. ):
7.
8.     """ Fonction qui calcule la sortie d'un réseau de neurones
9.
10.    Entrée(s):
11.    - X (np.array) : tableau 2D d'entrée du réseau
12.    - liste_dims (list) : liste des dimensions des couches du réseau (entrée et sortie incluses)
13.    - activation (str) : fonction d'activation des couches cachées
14.    - params (dict) : dictionnaire des paramètres obtenus après entraînement
15.
16.    Sortie(s):
17.    - Y_pred (np.array) : sortie du réseau
18.    """
19.
20.    vars = {}
21.    vars['a0'] = X
22.    nb_couches = len(liste_dims) - 1
23.    # Phase de propagation en avant :
24.    for j in range(nb_couches-1):
25.        vars[f'a{j+1}'], vars[f's{j+1}'] = sortie_activee(
26.            vars[f'a{j}'],
27.            params[f'w{j+1}'],
28.            params[f'b{j+1}'],
29.            activation
30.        )
31.    vars[f'a{nb_couches}'], vars[f's{nb_couches}'] = sortie_activee(
32.        vars[f'a{nb_couches-1}'],
33.        params[f'w{nb_couches}'],
34.        params[f'b{nb_couches}'],
35.        'sigmoide'
36.    )
37.    Y_pred = vars[f'a{nb_couches}'] # Les valeurs prédites sont les sorties de la couche de sortie
38.
39.    return Y_pred
```

Une fois ces fonctions implémentées, il nous reste 2 dernières fonctions à coder. Ces fonctions consisteront à réaliser l'encodage one-hot des labels (et à retourner à l'encodage initial) car le réseau de neurones va apprendre à prédire l'équivalent de probabilités.

Voici le code des fonctions correspondantes (dans *utils.py*) :

```
1. def one_hot(Y):
2.     """ Fonction qui permet de réaliser l'encodage one-hot du tableau des labels """
3.     res = np.eye(10)[Y].reshape((Y.shape[0],10))
4.     return res
5.
6. def reverse_one_hot(Y):
7.     """ Fonction qui permet de retourner à l'encodage labels à partir de l'encodage one-hot"""
8.     res = np.argmax(Y,axis = 1).astype('int')
9.     return res
```

Notons que la fonction *reverse_one_hot* fonctionne également avec des probabilités donc nous pourrons l'utiliser pour transformer les prédictions des réseaux de neurones en décisions binaires pour chaque classe (avec exclusion mutuelle entre les classes pour un échantillon donné).

2) Expérimentations

Le script des différentes expérimentations se trouve dans *nn_script.py*. L'idée ici sera de partir d'un réseau quelconque avec des paramètres choisis arbitrairement et ensuite d'étudier l'influence de chaque hyperparamètre. Il convient de remarquer que même si nous tenterons d'étudier l'influence de chaque hyperparamètre sur la performance, ces hyperparamètres ne sont pas indépendants pour autant et donc il est préférable d'anticiper le changement que pourrait apporter chaque hyperparamètre à la valeur prise par un autre hyperparamètre. C'est avec cette approche globale que nous pourrions alors obtenir à la fin un réseau avec des performances bien meilleures en partant d'un réseau initial puis en modifiant certains hyperparamètres bien choisis à chaque fois. Par ailleurs, le but ici n'est pas de trouver la valeur optimale de chaque hyperparamètre mais d'en trouver l'ordre de grandeur approprié (pour les hyperparamètres quantitatifs, ex. nombre de neurones, nombre de couches, etc.) ou le bon réglage (pour les hyperparamètres qualitatifs, ex. fonction d'activation, de perte, etc.). Notons aussi que dans ces expérimentations, nous n'entraînerons pas le réseau jusqu'à ce que la perte arrête de baisser mais seulement pour un nombre défini (le plus petit possible) d'itérations car le but est de tester l'influence de chacun des hyperparamètres et en général elle est visible assez rapidement.

Réseau initial :

Nous partons d'un réseau de neurones à 1 couche cachée avec 20 neurones. La fonction d'activation pour toutes les couches est la fonction sigmoïde et la perte utilisée est la perte quadratique. Nous choisissons une taille de mini-batch de 512 pour la descente du gradient et le taux d'apprentissage est de 10^{-2} . Enfin, nous allons entraîner le réseau avec 10 itérations.

Voici le résultat obtenu :

```
/ECL/S9/MOD/Apprentissage profond/  
Itération 1 : 22412.33802  
Itération 2 : 20699.46384  
Itération 3 : 20185.85567  
Itération 4 : 19921.76972  
Itération 5 : 19728.72573  
Itération 6 : 19575.21127  
Itération 7 : 19443.15912  
Itération 8 : 19334.53646  
Itération 9 : 19243.81439  
Itération 10 : 19162.94774  
Précision 1er réseau : 31.54 %
```

Avec ce réseau simple, nous arrivons déjà à une performance similaire à l'algorithme kppv.

Ajout d'une couche cachée :

Ajoutons une couche cachée au réseau précédent. Notons que l'ajout d'une couche augmente le nombre de paramètres à apprendre et donc l'espace d'optimisation est plus large, le nombre d'itérations nécessaire pour trouver l'optimum dans cet espace peut donc augmenter. Nous allons donc entraîner ce réseau pour quelques itérations supplémentaires pour pouvoir comparer l'effet sur les performances. Voici le résultat obtenu :

```
/ECL/S9/MOD/Apprentissage profond
Itération 1 : 22319.46962
Itération 2 : 21273.71406
Itération 3 : 21023.90174
Itération 4 : 20558.08862
Itération 5 : 20195.25294
Itération 6 : 19945.86025
Itération 7 : 19748.77117
Itération 8 : 19600.65772
Itération 9 : 19482.35734
Itération 10 : 19380.60355
Itération 11 : 19290.66782
Itération 12 : 19210.14742
Itération 13 : 19134.92866
Itération 14 : 19064.60417
Itération 15 : 19003.51192
Précision 2ème réseau : 32.23 %
```

Les performances semblent être légèrement meilleures. Nous pouvons donc conserver un réseau à 2 couches dans la suite. Voyons dans la suite si l'augmentation du nombre de neurones améliore le résultat.

Augmentation du nombre de neurones par couche et variation du taux d'apprentissage :

Étant donné que nous avons que les variables d'entrée sont de grande dimension, il semble logique d'augmenter le nombre de neurones de la 1^{ère} couche et pour la 2^{ème} couche nous pouvons choisir un nombre de neurones intermédiaire entre celui de la 1^{ère} couche et celui de la couche de sortie. Nous allons donc choisir un réseau avec 100 neurones dans la 1^{ère} couche puis 50 neurones dans la 2^{ème} couche. Nous augmentons significativement le nombre d'itérations pour pouvoir comparer l'effet sur la performance car le réseau est plus complexe que précédemment. Par ailleurs, nous allons comparer 2 taux d'apprentissage différents sur ce réseau. Voici le résultat obtenu :

$\alpha = 10^{-2}$	$\alpha = 10^{-3}$
Itération 47 : 19015.47150	Itération 47 : 18643.28366
Itération 48 : 18984.60691	Itération 48 : 18624.13010
Itération 49 : 18951.51414	Itération 49 : 18605.36255
Itération 50 : 18915.83781	Itération 50 : 18586.95261
Précision 3ème réseau : 29.83 %	Précision 3ème réseau : 34.12 %
D:\ECL\S9\MOD\AppData\profond\TD - BE\	D:\ECL\S9\MOD\AppData\profond\TD - BE\TD1>

On constate que la performance est nettement meilleure pour le taux d'apprentissage de 10^{-3} et d'ailleurs on fait mieux que les réseaux précédents. Nous allons donc garder ces nombres de neurones sur les 2 couches ainsi que le taux d'apprentissage de 10^{-3} .

Changement de fonction d'activation des couches cachées, RELU au lieu de sigmoïde :

À présent, changeons de fonction d'activation : utilisons RELU au lieu de la fonction sigmoïde. Nous obtenons alors le résultat suivant :

```

Itération 48 : 15738.26649
Itération 49 : 15738.26649
Itération 50 : 15784.06952
Précision 4ème réseau : 45.17 %

```

Cela représente une nette amélioration par rapport au réseau précédent et cela dépasse également l'ensemble des performances observées jusqu'à présent. La fonction d'activation RELU semble donc être plus adaptée que Sigmoïde pour notre problème et notre jeu de données. Cela provient probablement du fait que la fonction RELU réduit les problèmes d'explosion/disparition du gradient. Nous conserverons donc cette fonction dans la suite.

Changement de fonction de perte, entropie croisée au lieu de perte quadratique :

Changeons maintenant également la fonction de perte en choisissant l'entropie croisée au lieu de la perte quadratique tout en gardant le taux d'apprentissage à 10^{-3} . Nous obtenons le résultat suivant :

```

Itération 48 : 76902.36339
Itération 49 : 77758.01587
Itération 50 : 76832.20569
Précision 5ème réseau : 40.77 %

```

La performance diminue d'environ 5 % par rapport au réseau précédent. Nous pouvons cependant prolonger l'apprentissage pour voir si cela améliore les résultats. Pour 100 itérations, nous obtenons le résultat suivant :

```

Itération 97 : 68844.99703
Itération 98 : 69835.21603
Itération 99 : 68388.38846
Itération 100 : 69030.45497
Précision 5ème réseau : 44.43 %
D:\ECL\S9\MOD\Apprentissage profond\TD - BE\TD1>

```

On constate qu'on s'approche du réseau précédent (avec RELU et une perte quadratique). Nous pouvons donc garder ce réseau étant donné que la perte d'entropie croisée (associée à la fonction d'activation *softmax*) est plus adaptée à notre problème et ce réseau peut être encore amélioré même si l'apprentissage semble être plus lent.

Diminution de la taille du batch :

À présent, diminuons la taille du mini-batch pour la descente du gradient en la diminuant par puissance de 2 :

256 :		
	<pre> Itération 48 : 59323.33481 Itération 49 : 58731.89816 Itération 50 : 58382.68556 Précision 6ème réseau taille_batch 256 : 48.32 % </pre>	
128 :		
	<pre> Itération 49 : 52692.56952 Itération 50 : 52556.60540 Précision 6ème réseau taille_batch 128 : 49.59 % </pre>	

64 :		
	Itération 48 : 49896.78355 Itération 49 : 49734.75886 Itération 50 : 49235.98019 Précision 6ème réseau taille_batch 64 : 49.31 %	
32 :		
	Itération 48 : 48488.19258 Itération 49 : 48329.51890 Itération 50 : 48025.98426 Précision 6ème réseau taille_batch 32 : 50.13 %	

La diminution de la taille du mini-batch semble améliorer les performances et la taille de 32 semble être la meilleure. Nous conservons donc cette taille dans la suite.

Prise en compte de la régularisation L2 :

À présent, nous allons prendre en compte la régularisation L2 en prenant un coefficient de régularisation non nul et en testant plusieurs valeurs par puissance de 10 :

$\lambda = 0.1$		
	Itération 49 : 71174.35122 Itération 50 : 71116.37110 Précision 7ème réseau coeff_l2 0.1 : 42.93 %	
$\lambda = 0.01$		
	Itération 48 : 53600.84277 Itération 49 : 53348.93299 Itération 50 : 53168.46550 Précision 7ème réseau coeff_l2 0.01 : 50.32 %	
$\lambda = 0.001$		
	Itération 48 : 49094.85665 Itération 49 : 48762.82210 Itération 50 : 48595.18565 Précision 7ème réseau coeff_l2 0.001 : 49.32 %	

On constate que les résultats s'améliorent légèrement pour $\lambda = 0.01$, ce qui signifie qu'il y avait potentiellement un léger surapprentissage dans le réseau précédent (avec la taille de batch valant 32) et donc la régularisation a eu pour effet de diminuer le surapprentissage pour que le modèle puisse mieux généraliser sur de nouvelles données.

Utilisation de descripteurs HOG en entrée :

Jusqu'à maintenant notre approche était centrée sur le modèle et comment on pouvait l'améliorer en changeant certains hyperparamètres, nous avons vu que nous avons gagné presque 20 % en précision par rapport au réseau initial ce qui est une amélioration significative et illustre la puissance des réseaux de neurones pour la classification d'images.

Tentons à présent une approche centrée sur les données elles même pour voir si cela améliore la performance. Pour cela, nous allons donner au réseau en entrée des descripteurs HOG au lieu des images brutes et utiliser le même réseau (sans la régularisation).

Nous obtenons une performance de 40 % comme visible sur la capture suivante :

```
Itération 97 : 78679.45561  
Itération 98 : 78613.22449  
Itération 99 : 78555.61437  
Itération 100 : 78488.80043  
Précision 8ème réseau : 40.36 %
```

On constate donc que le réseau fait moins bien que le réseau sans les descripteurs, ce qui signifie que les descripteurs choisis font soit perdre de l'information soit ils rajoutent du bruit aux données.

Utilisation des descripteurs LBP en entrée :

Essayons maintenant les descripteurs LBP en entrée à la place des descripteurs HOG. Avec le même réseau (sans la régularisation), nous obtenons la performance suivante :

```
Itération 48 : 35514.83259  
Itération 49 : 33798.91251  
Itération 50 : 33420.98215  
Précision 9ème réseau : 24.00 %
```

Nous avons le même constat que le réseau avec les descripteurs HOG. Par ailleurs, il convient de rappeler que les descripteurs LBP avaient donné quasiment le même résultat avec l'algorithme kppv. Compte tenu de la valeur de la loss qui est étonnamment faible, voyons si une régularisation avec un coefficient de 0.1 améliore les résultats :

```
Itération 49 : 83523.76505  
Itération 50 : 83366.06436  
Précision 9ème réseau : 29.29 %
```

La régularisation a un effet positif mais la performance reste toujours médiocre.

Conclusion provisoire sur l'utilisation des descripteurs :

Nous constatons que l'approche basée sur les descripteurs ne donne pas de bons résultats (du moins avec les hyperparamètres que nous avons choisi). Cela est plutôt logique car la qualité de l'extraction des descripteurs a une répercussion sur la performance, si les descripteurs sont de mauvaise qualité et ne synthétisent pas correctement l'information contenue dans les images, le réseau de neurones pourra difficilement apprendre. C'est pourquoi il est préférable de laisser le réseau extraire ses propres descripteurs, ce qui est fait en général par les premières couches d'un réseau de neurones profond. C'est aussi la raison pour laquelle le réseau sans les descripteurs à 2 couches avec 100 neurones sur la 1^{ère} couche, puis 50 neurones sur la 2^{ème} couche est celui qui fonctionne le mieux dans les expériences décrites dans ce compte rendu et c'est celui que nous choisirons pour la cross-validation.

Cross-validation :

Évaluons à présent le meilleur modèle par validation croisée en 5 répertoires. Nous obtenons une précision moyenne de 50.13 % comme visible sur la capture suivante :

```
Itération 48 : 56467.15489
Itération 49 : 56172.48245
Itération 50 : 56045.34562
100%|
Précision moyenne : 50.15208333333334
```

Le modèle à 2 couches est donc suffisamment stable et on peut considérer qu'il n'y a pas de surapprentissage.

Afin de confirmer que ce réseau est bien meilleur que le réseau initial, entraînons le réseau initial avec le même nombre d'itérations (50) car nous ne l'avions entraîné que pour 10 itérations au départ. Nous obtenons la performance suivante :

```
Itération 47 : 18002.41780
Itération 48 : 17979.05358
Itération 49 : 17951.28720
Itération 50 : 17949.51023
Précision 1er réseau : 36.92 %
```

Les résultats sont donc a priori confirmés.

Conclusion des expérimentations :

Ces expérimentations nous ont permis de constater qu'un réseau de neurones utilisant :

- 2 couches cachées avec 100 neurones pour la 1^{ère} et 50 neurones pour la 2^{ème}
- une fonction RELU comme activation
- l'entropie croisée comme perte
- la régularisation L2 avec $\lambda=0.01$
- une petite taille de batch (32)
- un taux d'apprentissage de 10^{-3}

permettait d'obtenir de meilleures performances qu'un réseau avec :

- 1 seule couche cachée avec 20 neurones
- la fonction sigmoïde comme activation
- l'erreur quadratique comme perte
- une grande taille de batch (512)
- un taux d'apprentissage de 10^{-2}
- l'absence de régularisation

Parmi l'ensemble des hyperparamètres, l'effet amélioratif de certains sur la performance peut être discutée car il est difficile de modéliser l'interaction avec les autres hyperparamètres. Cependant, nous pouvons considérer que certains d'entre eux semblent améliorer le résultat de manière quasi-indépendante des autres, cela semble être le cas par exemple de la fonction d'activation RELU utilisée à la place de sigmoïde.

V- Conclusion

Ce TD nous a permis de mettre en œuvre la classification d'images en utilisant dans un premier temps l'algorithme des k plus proches voisins puis dans un deuxième temps, un réseau de neurones. Nous avons vu que l'algorithme kppv apportait une précision avoisinant les 35 % et nous laissait peu de marge de manœuvre pour améliorer la performance. Les réseaux de neurones nous ont permis d'améliorer ce résultat en augmentant la précision de 15 %, ce qui est significatif. Par ailleurs, pour arriver à ce résultat, nous avons modifié « par tâtonnement » certains hyperparamètres sans chercher à optimiser de manière systématique alors que nous aurions pu le faire. Ce réseau nous offrait encore beaucoup de marge de manœuvre mais il n'est pas certain que nous ayons gagné encore beaucoup en performance quand on sait que les réseaux de neurones profonds classiques ne sont pas vraiment utilisés pour la classification d'images. En effet, ces réseaux de neurones ne prennent pas en compte la corrélation spatiale qu'il y a dans une image, c'est pourquoi les réseaux convolutifs ont été inventés et sont aujourd'hui la méthode la plus utilisée dans la vision par ordinateur utilisant l'intelligence artificielle.