Apprentissage profond & Intelligence Artificielle

Rapport de TD1

KPPV et réseaux de neurones pour la classification d'images

MOD 3.2

Groupe :

Niermaréchal Robin

Zouaoui Iliès

Enseignant :

Duboudin Thomas

09 Novembre 2021

Sommaire

1 Introduction

L'objectif de ce premier TD du MOD 3.2 est de mettre en application les notions que nous avons abordées lors des premiers cours en gardant en mémoire les précédents enseignements de TC d'INF afin de produire un code de classification basé sur les KPPV et un autre décrivant des réseaux de neurones.
Afin d'illustrer les objectifs clairement, nous allons dans un premier temps détailler la structure de notre code ainsi que les fonctions développées. Ensuite, nous évoquerons les tests réalisés ainsi que les résultats obtenus.

2 Structure globale du code et des fonctions développés

Afin de mettre en application les enseignements du MOD 3.2, nous devrons, au cours de ce TD, utilisé deux approches différentes pour prendre en main les notions de classifications mises en jeu. Ainsi, nous commencerons par développé un code de classification qui se basera sur les k-plus-proches-voisins puis nous utiliserons ensuite des réseaux de neurones avant de conclure sur les avantages et inconvénients de chacune de ces méthodes. Dans cette partie de notre rapport, nous détaillerons chacune des fonctions des codes ainsi obtenus ainsi que leur utilité dans notre code.

2.1 Code de classification à base des k-plus-proches-voisins

Le code de notre classificateur basé sur les k-plus-proches-voisins est contenu dans le fichier *"main_kppv.py"* de l'archive zip. Il est constitué des fonctions suivantes :

— **unpickle(file) :** permet d'importer les données des différents batch de la base cifar10 contenue dans l'archive zip afin de mener nos tests.

— **lecture_cifar(file) :** fonction qui appelle la fonction *unpickle(file)* afin d'importer les données de la base cifar10 et qui stocke les données dans un vecteur X et les codes de classes de données dans un vecteur Y. Les vecteurs X et Y contiennent les données de l'ensemble des batches.

— **decoupage_donnees(X,Y) :** découpe aléatoirement les données pour le test à hauteur de 80% et de 20% pour l'apprentissage à la fois pour X et pour Y.

— **kppv_distances(X_test,X_app) :** calcule la matrice de distances entre les images de test et celles d'apprentissage

```
def kppv_distances(X_test,X_app):
    X_test_dots=(X_test*X_test).sum(axis=1).reshape(X_test.shape[0],1)*np.ones
    (shape=(1,X_app.shape[0]))
    X_app_dots=(X_app*X_app).sum(axis=1)*np.ones(shape=(X_test.shape[0],1))
    Dist=X_test_dots+X_app_dots-2*X_test.dot(X_app.T)
    return Dist
```

— **kppv_predict(Dist,Y_app,K) :** estimation de la classe des images test, par rapport à l'ensemble d'apprentissage représenté par Y_{app}.

```
def kppv_predict(Dist,Y_app,K):
    A=np.argsort(Dist,axis=1)[:,:K] # les positions des K plus proches voisins de la
    ↪ matrice de distance
    B=np.take(Y_app,A) # les classes de ces K plus proches voisins
    C=np.apply_along_axis(np.bincount,1,B,minlength=10) # effectifs pour chaque
    ↪ classe dans les K voisins, triés par numéro de classe et non par distance
    Y_pred=np.array([])
    for i in range(np.shape(B)[0]):
        l_class=[]
        for j in B[i]:
            if j not in l_class:
                l_class.append(j)
        class_reco=np.column_stack((l_class,np.zeros(len(l_class),dtype=int))) # les
        ↪ classes reconnues, dans l'ordre de proximité, avec compteurs associés
        ↪ (nuls pour le moment)
        final_count=np.copy(class_reco) #une copie pour pouvoir modifier les
        ↪ compteurs par la suite
        for k,line in enumerate(class_reco):
            final_count[k,1]=C[i,line[0]] # calcul du nb d'apparition pour chaque
            ↪ classe reconnue
        Y_pred=np.append(Y_pred,final_count[np.argmax(final_count[:,1]),0]) # on
        ↪ retient la classe avec le + grand nb d'apparitions, et la plus proche si
        ↪ égalité
    return Y_pred
```

— **evaluation_classifieur(Y_test,Y_pred) :** permet d'évaluer la précision des classifications réalisées en calculant le pourcentage de bonnes prédictions dans l'échantillon.

Les expérimentations menées sur la méthode des k plus proches voisins sont disponibles dans le fichier *"kppv_experimentations.py"* de l'archive zip. Il est constitué des fonctions suivantes :

— **influence_param_k(path,k_max) :** fonction d'évaluation de l'influence du paramètre k (le nombre de voisins les plus proches), sans utiliser de descripteurs ni de validation croisée. Cette fonction a permis d'obtenir les courbes détaillées FIG. 1 montrant l'évolution de la performance du modèle en fonction du nombre de voisins retenu.

— **influence_param_k_LBP(path,k_max) :** de la même manière que la fonction précédente, cette fonction permet d'évaluer l'influence du paramètre k mais cette fois-ci avec l'utilisation d'un descripteur LBP. Les résultats seront présentés dans la FIG. 2 de la partie suivante.

— **influence_param_k_HOG(path,k_max) :** fonction permettant d'évaluer l'influence du paramètre k avec l'utilisation d'un descripteur HOG. Les résultats seront présentés dans la FIG.3 de la partie suivante.

— **influence_param_k_avec_validation_croisee(path,k_max,nb_batches) :** fonction permettant d'évaluer l'influence du paramètre k avec une validation croisée dont les résultats seront présentés dans la FIG.4 de la partie suivante.

2.2 Code de classification à base de réseaux de neurones

— **forward_2layers(X,W1,b1,W2,b2,Y) :** réalise une passe avant pour un réseau de neurones à deux couches, avec fonction d'activation type sigmoïde et fonction de perte type MSE.

```
def forward_2layers(X,W1,b1,W2,b2,Y):
    ######################################################
    # Passe avant : calcul de la sortie prédite Y_pred #
    ######################################################
    I1 = X.dot(W1) + b1 # Potentiel d'entrée de la couche cachée
    O1 = 1/(1+np.exp(-I1)) # Sortie de la couche cachée (fonction d'activation de
    ↪  type sigmoïde)
    I2 = O1.dot(W2) + b2 # Potentiel d'entrée de la couche de sortie
    O2 = 1/(1+np.exp(-I2)) # Sortie de la couche de sortie (fonction d'activation de
    ↪  type sigmoïde)
    Y_pred = O2 # Les valeurs prédites sont les sorties de la couche de sortie
    ######################################################
    # Calcul et affichage de la fonction perte de type MSE #
    ######################################################
    loss = np.square(Y_pred - Y).sum() / 2
    return O1,Y_pred,loss
```

— **back_propagate_2layers(X,Y,O1,O2,W1,W2,B1,B2,lr) :** réalise la rétropropagation et la mise à jour des poids du réseau.

La rétropropagation est réalisée en calculant le gradient de la fonction de perte L par rapport aux poids du modèle, c'est-à-dire W_2, W_1, B_2 et B_1. Nous allons détailler le calcul du gradient par rapport à W_2 et W_1, notés respectivement $\frac{\partial L}{\partial W_2}$ et $\frac{\partial L}{\partial W_1}$.
Tout d'abord, la décomposition du gradient donne (1).

$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial O_2}.\frac{\partial O_2}{\partial I_2}.\frac{\partial I_2}{\partial W_2} \tag{1}$$

Il est alors nécessaire de détailler les termes du produit matriciel présenté dans l'équation (1).
Tout d'abord, puisque $L(O_2, Y) = 1/2 \times (O_2 - Y)^2$

$$\frac{\partial L}{\partial O_2} = O_2 - Y \tag{2}$$

Puis, étant donné que $O_2 = \sigma(I_2)$ et au vu de l'expression de la dérivée de la fonction sigmoïde, on peut en déduire l'expression suivante :

$$\frac{\partial L}{\partial O_2}.\frac{\partial O_2}{\partial I_2} = \frac{\partial L}{\partial I_2} = (O_2 - Y) \odot ((1 - O_2) \odot O_2) \tag{3}$$

Enfin, étant donné que

$$\frac{\partial I_2}{\partial W_2} = \begin{pmatrix} O_1 & 0 & ... & 0 \\ 0 & O_1 & ... & ... \\ ... & ... & ... & 0 \\ 0 & ... & 0 & O_1 \end{pmatrix} \tag{4}$$

On obtient finalement (5)

$$\boxed{\frac{\partial L}{\partial W_2} = O_1^T.\frac{\partial L}{\partial I_2} = O_1^T.(O_2 - Y) \odot ((1 - O_2) \odot O_2)} \tag{5}$$

Puis de manière analogue, on décompose $\frac{\partial L}{\partial W_1}$:

$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial I_2}.\frac{\partial I_2}{\partial O_1}.\frac{\partial O_1}{\partial I_1}.\frac{\partial I_1}{\partial W_1} \tag{6}$$

Comme

$$\frac{\partial I_2}{\partial O_1} = W_2^T$$

On obtient

$$\frac{\partial L}{\partial I_2}.\frac{\partial I_2}{\partial O_1} = \frac{\partial L}{\partial O_1} = \frac{\partial L}{\partial I_2}.W_2^T \tag{7}$$

Puis, de manière analogue à l'équation (3), on obtient 8

$$\frac{\partial L}{\partial O_1}.\frac{\partial O_1}{\partial I_1} = (\frac{\partial L}{\partial I_2}.W_2^T) \odot (1 - O_1) \odot O_1 \tag{8}$$

Enfin, par analogie avec les équations (4) et (5),

$$\boxed{\frac{\partial L}{\partial W_1} = X^T.[(\frac{\partial L}{\partial I_2}.W_2^T) \odot (1 - O_1) \odot O_1]} \tag{9}$$

Enfin, concernant les gradients associés aux biais :

$$\frac{\partial L}{\partial B_2} = \frac{\partial L}{\partial I_2}.\frac{\partial I_2}{\partial B_2} \tag{10}$$

avec

$$\frac{\partial I_2}{\partial B_2} = I \tag{11}$$

Soit finalement :

$$\boxed{\frac{\partial L}{\partial B_2} = \frac{\partial L}{\partial I_2}} \tag{12}$$

Et par analogie

$$\boxed{\frac{\partial L}{\partial B_1} = \frac{\partial L}{\partial I_1}} \tag{13}$$

Enfin, on notera que la descente du gradient est effectuée de manière classique. Le code correspondant est donné ci-après :

```
def back_propagate_2layers(X,Y,O1,O2,W1,W2,B1,B2,lr):
    ## Gradient pour W2
    dL_dI2=(O2-Y)*((1-O2)*O2)
    dL_dw2=np.dot(np.transpose(O1),dL_dI2)

    ## Gradient pour W1
    dL_dO1=np.dot(dL_dI2,np.transpose(W2))
    dL_dI1=np.multiply(dL_dO1,(O1*(1-O1)))
    dL_dw1=np.dot(np.transpose(X),dL_dI1)

    ## Gradient pour B2
    dL_db2=dL_dI2

    ## Gradient pour B1
    dL_db1=dL_dI1

```

```
17      # Update des parametres du modele
18      w1=W1-dL_dw1*lr # pour w1
19      b1=B1-dL_db1*lr# pour b1
20      w2=W2-dL_dw2*lr # pour w2
21      b2=B2-dL_db2*lr # pour b2
22
23      return w1,b1,w2,b2
```

— **train_nn_2layers(path,D_h,lr,n_iter) :** réalise l'entraînement complet du réseau à 2 couches, pour un nombre de neurones de la couche cachée (D_h), un learning rate (lr) et un nombre d'itérations maximal (n_iter) donnés.

```
1  def train_nn_2layers(X,Y,D_h,lr,n_iter):
2      # Géométrie du problème
3      X=X[:1000,:]/255 # rescale pour faciliter la convergence
4      Y=Y[:1000,:]/9
5      N,D_in=X.shape
6      D_out=1
7
8      # Initialisation aléatoire des poids du réseau
9      W1 = 2 * np.random.random((D_in, D_h)) - 1
10     B1 = np.zeros((1,D_h))
11     W2 = 2 * np.random.random((D_h, D_out)) - 1
12     B2 = np.zeros((1,D_out))
13
14     ## Back propagation
15     loss_list=[]
16     accuracy_list=[]
17     for n in range(n_iter):
18         O1,O2,loss=forward_2layers(X, W1, B1, W2, B2, Y)
19         W1,B1,W2,B2=back_propagate_2layers(X, Y, O1, O2, W1, W2, B1, B2, lr)
20         loss_list.append(loss)
21         accuracy_list.append(evaluation_classifieur(Y, np.round(O2)))
22
23     return loss_list,accuracy_list
```

Les expérimentations menées sur le réseau de neurone sont disponibles dans le fichier *"neurones_experimentations.py"* de l'archive zip. Il est constitué des fonctions suivantes :

— **change_nb_neurons(path) :** tracé des vitesses de convergence pour différents nombre de neurones dans la couche cachées. Trace également la valeur de la fonction de perte obtenue en fin d'entraînement pour différents nombre de neurones dans la couche cachées

— **change_learning_rate(path) :** tracé des vitesses de convergence pour différentes valeur du learning rate ou taux d'apprentissage.

— **mini_batches(path) :** tracé des vitesses de convergence pour un réseau à 2 couches, en utilisant des mini-batches.

— **change_nb_layers(path) :** tracé des vitesses de convergence pour un réseau à 2 couches et un réseau à 3 couches.

— **descripteurs_3layers(path) :** tracé des vitesses de convergence en appliquant des descripteurs en amont d'un réseau à 3 couches.

3 Tests et résultats obtenus avec les fonctions mises en place

3.1 KPPV

3 études ont été menées sur le modèle réalisé des k plus proches voisins :

— La variation du nombre de voisins k, afin de déterminer le nombre de voisins qui procure la performance optimale. En effet, augmenter le nombre de voisins peut conduire au sur-entraînement du modèle, ce qui n'est évidemment pas souhaitable. De manière analogue, un nombre de voisins trop faible peu également conduire à un modèle trop peu performant.

— L'utilisation de descripteurs (*LBP* et *HOG*) en amont de la classification. Un descripteur choisit judicieusement peut faire ressortir des caractéristiques importantes de l'image en vue de la classification.

— La validation croisée à N répertoires. L'objectif de cette méthode de validation est de s'affranchir de certains biais introduits par le découpage en batch de test et d'apprentissage. En effet, le taux de classification peut varier sensiblement avec le choix des données d'apprentissage. En répétant la classification avec différents batch d'apprentissage, cette variabilité est diminuée.

Note importante : Afin de pouvoir comparer les études entre elles, nous travaillons dans chacun des cas sur la courbe décrivant l'évolution du taux de classification en fonction du nombre de voisins retenus. Ainsi, nous nous affranchissons des fluctuations liées au nombre de voisins, visibles sur la FIG. 1.

3.1.1 Variation du nombre de voisins

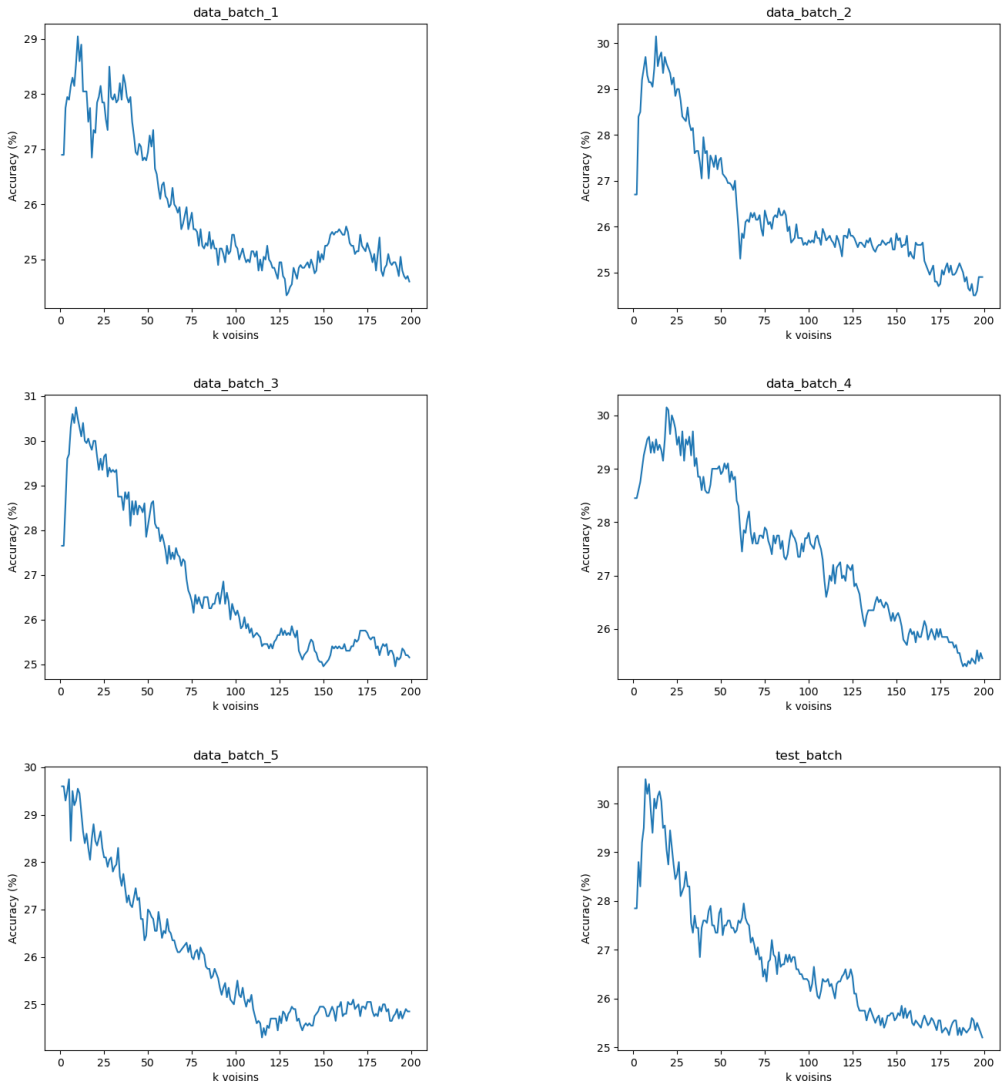

FIGURE 1 – Résultats des tests menés sur les différents batch de la base cifar10

Bien que la FIG. 1 montre que la sensibilité au paramètre k est différente selon les batches, nous remarquons une certaine similarité dans le comportement du taux de classification. En effet, un **optimum semble se distinguer entre 10 et 20 voisins**, associé à un taux de classification d'environ 30%.

3.1.2 Descripteur LBP

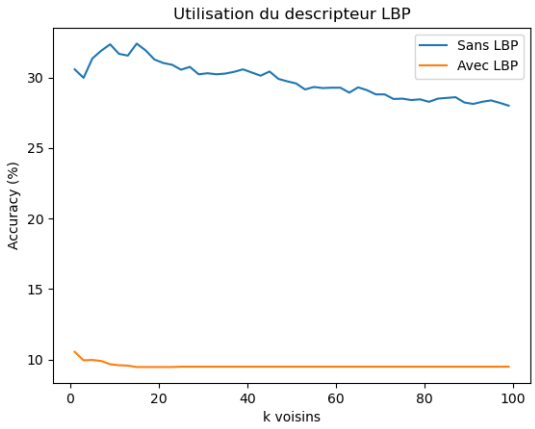

FIGURE 2 – Sensibilité au paramètre k après application du descripteur *LBP*. Test réalisé sur 20 000 images.

La FIG. 2 nous montre que l'utilisation du descripteur *LBP* n'est pas pertinente dans notre application de classification d'images, puisque le taux de classification est alors divisé par 3, et ce quelle que soit la valeur du paramètre k.

3.1.3 Descripteur HOG

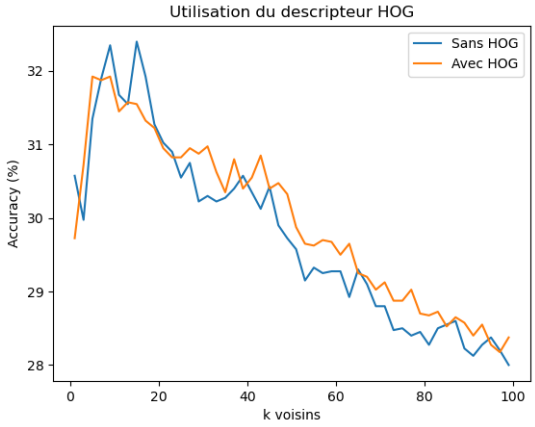

FIGURE 3 – Sensibilité au paramètre k après application du descripteur *HOG*. Test réalisé sur 20 000 images.

Contrairement au cas précédent, la FIG. 3 nous montre que l'utilisation du descripteur *HOG* peut s'avérer pertinente pour notre application de classification d'images, puisque le taux de classification est sensiblement le même par rapport à un modèle sans descripteur. Certes, nous observons une plus grande robustesse quand le paramètre k augmente, mais le modèle possède des niveaux de performance similaires au niveau de la valeur optimale de k. Utiliser le descripteur *HOG* ne détériore donc pas les résultats, mais n'apporte pas une amélioration significative. De plus, l'application du descripteur à chaque image rallonge de manière non-négligeable le temps de calcul. Dans notre cas d'utilisation, les bénéfices apportés ne semblent pas suffisamment élevés pour valider son utilisation.

3.1.4 Validation croisée

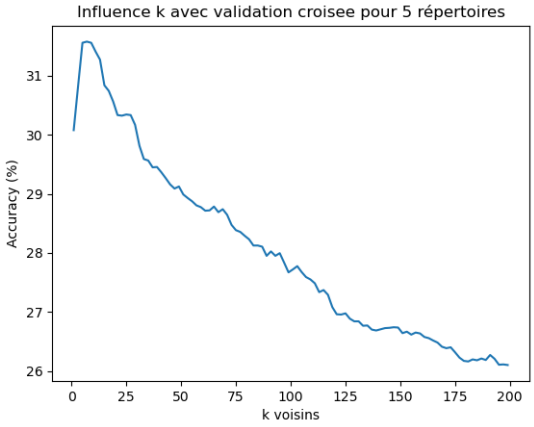

FIGURE 4 – Sensibilité au paramètre k après validation croisée, sans utiliser de descripteur. Test réalisé sur 20 000 images.

Comme illustré sur la FIG. 4, la validation croisée apporte les effets attendus : les fortes fluctuations du taux de classification lorsque k varie, observables sur la FIG. 1 sont fortement diminuées. La courbe d'évolution est comme "lissée" et il est bien plus facile de déterminer la valeur optimale de k. Ici, on peut estimer qu'elle se trouve entre 10 et 12 voisins.

3.2 Réseaux de neurones

3.2.1 Influence du nombre de neurones

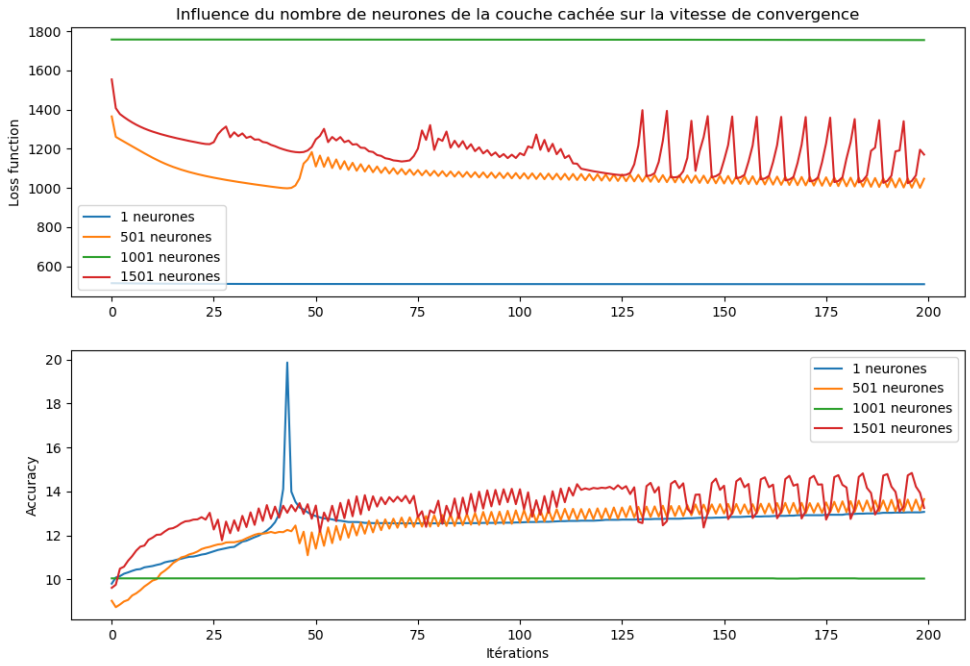

FIGURE 5 – Influence du nombre de neurones de la couche cachée sur la vitesse de convergence lors de l'entraînement. Test réalisé sur 10 000 images, avec un learning rate de 10^{-3}.

La FIG. 5 montre la difficulté de trouver un nombre de neurones assurant une convergence stable. En effet, lorsque le nombre de neurones de la couche cachée augmente fortement, la valeur de fonction de perte semble diminuer (en tendance) mais est soumise à de très fortes oscillations. De même, le taux de classification semble augmenter mais oscille lui aussi. Lorsque 1500 neurones sont utilisés (courbe rouge), une variation d'une dizaine d'itérations peut produire des résultats totalement différents. Ce comportement est probablement du à des valeurs de learning rate non appropriées qui provoquent une descente trop rapide du modèle au bout d'un certain nombre d'itérations et donc son oscillation autour du point de minimum global.
De plus, il est important de noter que pour certains nombres de neurones, le réseau ne semble pas apprendre : c'est la cas pour $n = 1$ et $n = 1001$ neurones.
De plus, bien que le réseau avec un neurone (en bleu) semble produire un modèle performant, on pourra se questionner quant à la pertinence de cette configuration au vu de la complexité du problème de classification. Ce résultat est peut-être dû à une erreur de code, mais nous n'avons pas réussi à la déterminer si c'est le cas.

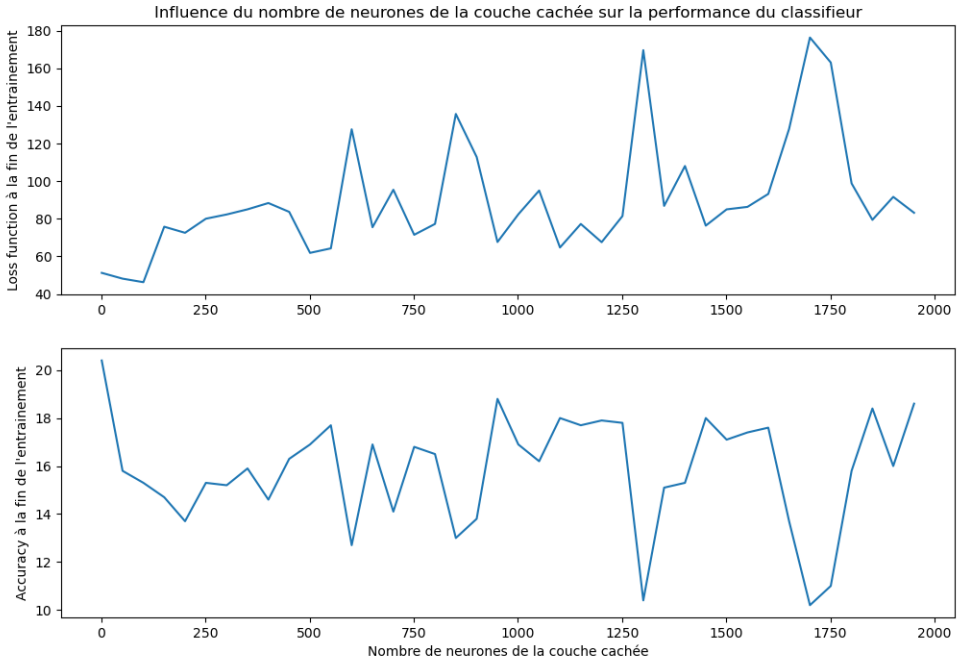

FIGURE 6 – Influence du nombre de neurones de la couche cachée sur le taux de classification obtenu en fin d'entraînement (200 itérations). Test réalisé sur 10 000 images, avec un learning rate de 10^{-3}.

Dès lors, nous avons tracé la FIG. 6 afin de trouver une valeur optimale pour le nombre de neurones de la couche cachée. Étant donné que les calculs sont longs, nous n'avons pas pu discrétiser très finement l'intervalle $[1; 2000]$ neurones. Contrairement à nos attentes, nous ne parvenons pas à déterminer un optimum, puisqu'il ne nous est pas possible de déterminer une quelconque tendance sur les évolutions du taux de classification et de la fonction de perte rapportées FIG. 6. De plus, les maximums observés au-delà de 1500 neurones semblent être liés à des learning rates non adaptés, comme décrit plus haut.
Pour conclure, il parait à ce stade difficile d'estimer le nombre de neurones optimal pour le réseau.

3.2.2 Influence du learning rate

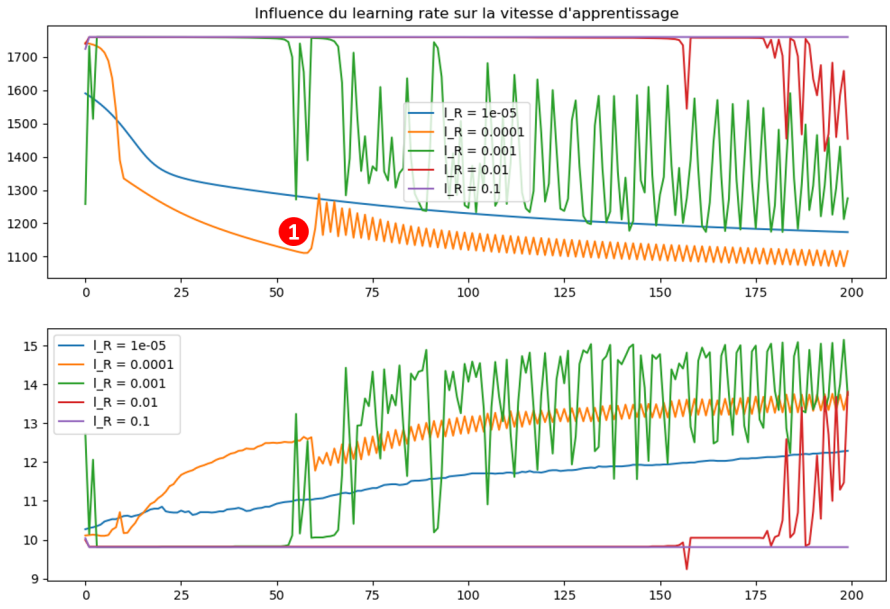

FIGURE 7 – Influence du learning rate sur la vitesse de convergence. Test réalisé sur 10 000 images, avec 500 neurones dans la couche cachée.

Le learning rate caractérise la vitesse de descente du gradient. La FIG. 7 montre son influence sur la vitesse de convergence lors de l'entraînement du modèle. Nous pouvons remarquer que le choix du learning rate doit être fait avec précaution, sous peine d'obtenir un modèle qui n'apprend pas (courbes rouges et violettes, correspondant à des learning rates supérieurs à 10^{-2}). La valeur de 10^{-4} procure une convergence plus rapide et un meilleur taux de classification au début de l'entraînement, jusqu'au point (1). Au niveau de ce point, le taux de classification décroît brutalement puis se met à osciller. Ce phénomène traduit la nécessité de changer le taux d'apprentissage à partir de cette étape, pour une valeur inférieure.
Changer le taux d'apprentissage au fur et à mesure des itérations pourrait permettre d'atteindre la convergence du modèle, et ce de manière plus rapide. En effet, on peut observer qu'un learning rate de 10^{-5} ne provoque pas de telles d'oscillations mais demeure bien plus lent. En effet, la performance au bout de 1000 itérations n'est pas meilleure avec $l_R = 10^{-5}$ puisque la descente du gradient est plus lente.

3.2.3 Implémentation d'une 3e couche

Une couche supplémentaire a été rajoutée au modèle. Plusieurs jeux de paramètres ont été testés, et nous avons conservé les réglages suivants pour notre étude :

- **Réseau à une couche cachée** (en bleu sur FIG. 8)
  - 750 neurones
  - learning rate de 10^{-4}
- **Réseau à 2 couches cachées** (en orange sur FIG. 8)
  - 750 neurones sur la couche 1
  - 1 000 neurones sur la couche 2
  - learning rate de 10^{-4}

Enfin, nous avons entraîné notre modèle sur uniquement 1 000 images afin de diminuer le temps de calcul. Il serait intéressant de pouvoir réaliser la même étude sur un échantillon bien plus grand, afin de confirmer les résultats obtenus.

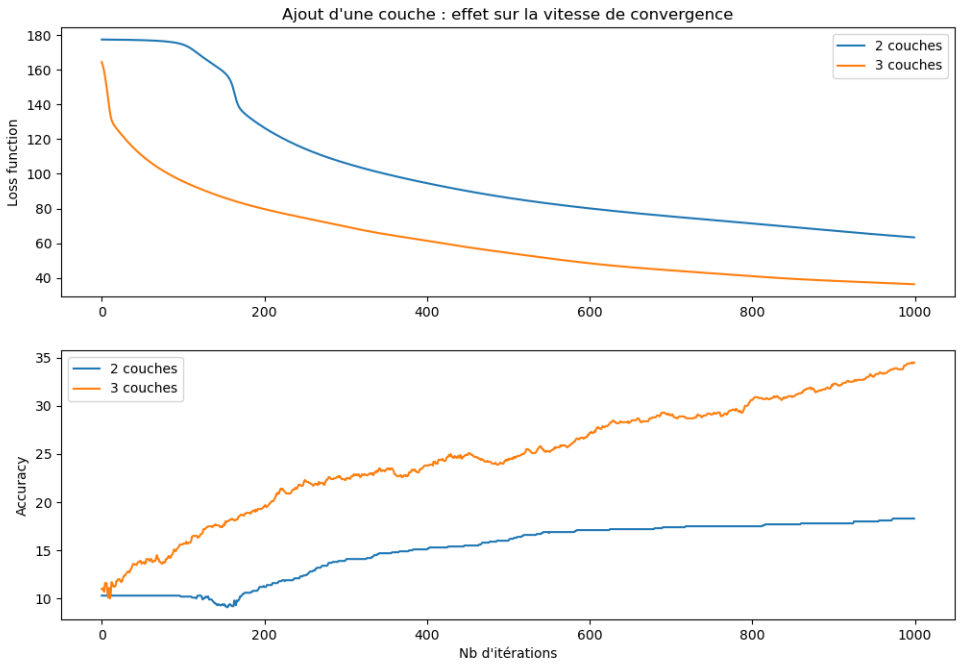

FIGURE 8 – Vitesses de convergence entre un réseau à 1 couche cachée (750 neurones) et un réseau à 2 couches cachées (750 et 1000 neurones). Test réalisé sur 1 000 images, avec un learning rate de 10^{-4}.

Avec les paramètres retenus, et en particulier avec le même nombre de neurones sur la première couche cachée, le modèle à 3 couches est plus rapide et plus performant. En effet, au bout de 1000 itérations, le taux de classification atteint 35% alors qu'avec un modèle à 2 couches, il ne dépasse pas les 20%.

3.2.4 Ajout de descripteurs sur le réseau à 3 couches

De manière analogue à ce qui a été réalisé avec le modèle *kppv*, nous appliquons les descripteurs *LBP* et *HOG* en amont du réseau à 3 couches (2 couches cachées) décrit dans la section précédente. En effet, ce réseau à 3 couches était le réseau procurant les meilleurs résultats jusqu'à présent, nous avons donc voulu tester les descripteurs sur ce réseau "optimal".

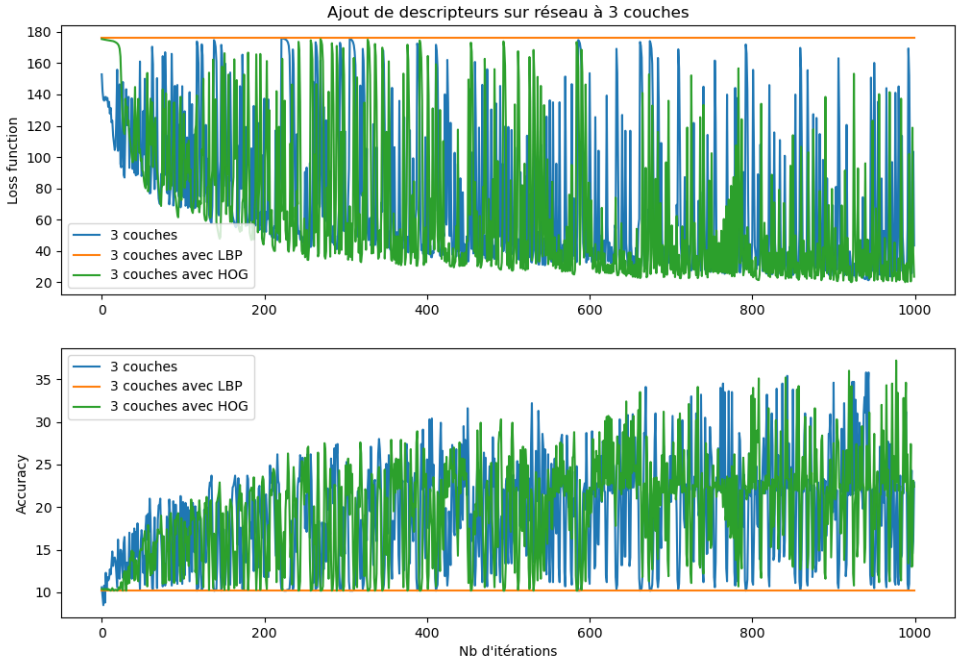

FIGURE 9 – Vitesses de convergence pour un réseau à 2 couches cachées (750 et 1000 neurones), avec différents descripteurs appliqués. Test réalisé sur 1 000 images, avec un learning rate de 10^{-3}.

Pour le descripteur *LBP*, le modèle ne semble pas apprendre, malgré nos tentatives de modification du nombre de neurones, du nombre d'images données pour l'apprentissage ainsi que du learning rate. Pour le descripteur *HOG*, nous avons tout d'abord obtenu un modèle qui n'apprenait pas avec un learning rate de 10^{-4}. En augmentant ce taux à une valeur de 10^{-3}, nous obtenons le comportement décrit FIG. 9, avec la courbe verte. Néanmoins, avec ce learning rate, le modèle oscille bien plus. De plus, la valeur de taux de classification obtenue en fin d'entraînement n'est pas meilleure que sans descripteur.

Dans notre cas, l'application des descripteurs n'aura pas permis d'améliorer la performance du réseau de neurones.

3.2.5 Utilisation de mini-batches

Enfin, comme illustré FIG.10, nous avons implémenté un apprentissage reposant sur la décomposition en 10 mini-batches, mais l'apprentissage n'est pas satisfaisant. En effet, nous observons de fortes oscillations, qui surviennent dès que l'on change de batch. Nous avons essayé de changer le learning rate, et le nombre de mini-batches utilisés mais le comportement est toujours similaire à celui présenté.

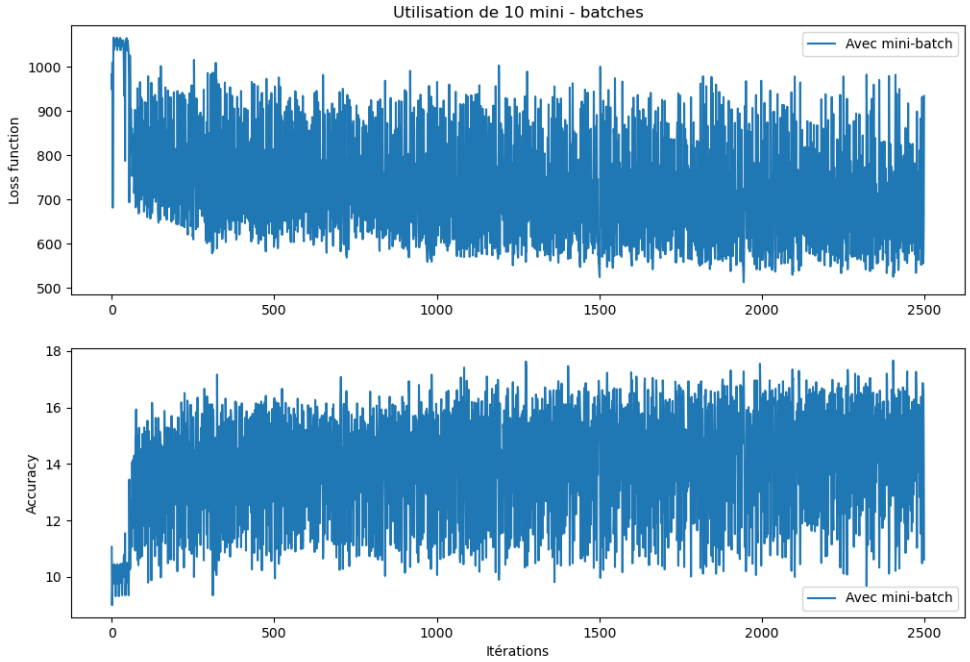

FIGURE 10 – Vitesses de convergence pour un réseau à une couche, en utilisant 10 mini batches. Test réalisé avec un learning rate de 10^{-3}.

3.2.6 Perspectives d'amélioration

Les différentes études réalisées sur le modèle utilisant un réseau de neurones nous ont montré qu'il est difficile de trouver un jeu de paramètres induisant des performances correctes de classification. En particulier, le modèle semble très sensible au learning rate retenu, paramètre qui peut bloquer ou faire osciller l'apprentissage du modèle. Ajouter des couches au modèle semble pertinent, à condition de déterminer les bons nombre de neurones pour chacune d'entre elles.
Enfin, de nombreuses stratégies pourraient être mises en place pour améliorer l'état actuel du modèle :

- Changer de fonction d'activation et opter pour une fonction de type RelU. L'objectif étant de s'affranchir de la saturation de la fonction sigmoïde.
- Utiliser un learning rate qui évolue au fil des itérations : grand au début du modèle, puis de plus en plus fin lorsque l'on se rapproche du point de minimum global.
- Faire du pre-traitement des images données au modèle. Par exemple, centrer l'échantillon d'apprentissage pourrait rendre le modèle plus robuste.
- Changer la méthode d'initialisation des poids du réseau, afin d'éviter un écrasement de la sortie de chaque neurones autour de 0 ou de 1.
- Opter pour une méthode différente concernant la descente du gradient, afin d'accélérer la descente du gradient.

Nous n'avons pas eu le temps d'implémenter toutes ces améliorations, mais elles mériteraient d'être étudiées en vue d'être intégrées au classifieur actuel.

4 Conclusion

En conclusion, ce premier TD de d'Apprentissage profond & Intelligence Artificielle nous a permis de mieux comprendre les applications pratiques de la programmation de réseaux de neurones.

Le code que nous avons pu développer est entièrement fonctionnel et représente une bonne base qui nous a permis de comprendre les problématiques du développement d'IA dans un contexte réaliste que nous pourrions rencontrer en entreprise.

5 Bibliographie

1. [Cours du MOD 3.2] (05 Octobre 2021)
   Consulté sur https://pedagogie3.ec-lyon.fr/course/view.php?id=1715&section=2

2. [Enoncé du TD1 - MOD 3.2] (05 Octobre 2020)
   Consulté sur https://pedagogie3.ec-lyon.fr/pluginfile.php/53257/mod_resource/content/3/TD1_classification_images.pdf