TP

Apprentissage automatique

PSI : Lycée Rabelais

1 Présentation du problème

On s'intéresse à la problématique de la reconnaissance de chiffres manuscrits. Cette application de l'intelligence artificielle est déjà assez ancienne - sa percée est intervenue en 1989 avec la réalisation d'une analyse syntaxique fiable et automatisée des codes postaux pour les courriers.

Depuis, de nombreuses institutions financières ont adopté la technique d'analyse automatique des numéros de compte sur les bordereaux de versement pour les virements électroniques ou les chèques bancaires.

Cette technique s'est maintenant généralisée dans d'autres domaines. Il s'agit, par exemple, du même type d'algorithme pour la reconnaissance faciale.

Une base de données contenant des données labellisées de chiffres manuscrits est disponible directement sur `Python` dans la bibliothèque `scikit learn`. On y retrouve :

- Des images de chiffres manuscrits (entrées de l'algorithme) ;
- L'entier associé au chiffre écrit de manière manuscrite (sorties de l'algorithme).

Question 1. De quel type de problème d'intelligence artificielle s'agit-il ?

Il s'agit d'un problème de classification supervisée.

Question 2. Comment sont obtenues de telles données ?

Il s'agit de chiffres manuscrits scannés ou photographiés puis labellisés par un humain !

On pourra donc commencer par écrire :

```
from sklearn import datasets ## module datasets contenant des bases de données
digits = datasets.load_digits() ## digits contient les entrées et les sorties

import matplotlib.pyplot as plt
plt.imshow(digits.images[25],cmap='binary')
            ## permet d'afficher l'image indicée 25
```

Question 3. Executer plusieurs fois ces premières lignes en changeant l'indice de l'image associée et observer le résultat.

On observe les images associées aux chiffres manuscrits.

Pour travailler sur les données, il est nécessaire d'obtenir, en entrée, des vecteurs (et non pas un tableau `numpy` à deux dimensions) contenant l'intensité associée à chaque pixel. Ici, l'image est en niveau de gris ce qui signifie qu'il n'y a donc qu'une seule couche "de couleur". Plus la valeur est grande, plus le pixel est foncé. Une valeur nulle est associée à un pixel blanc.

Pour pallier ce problème, on pourra écrire :

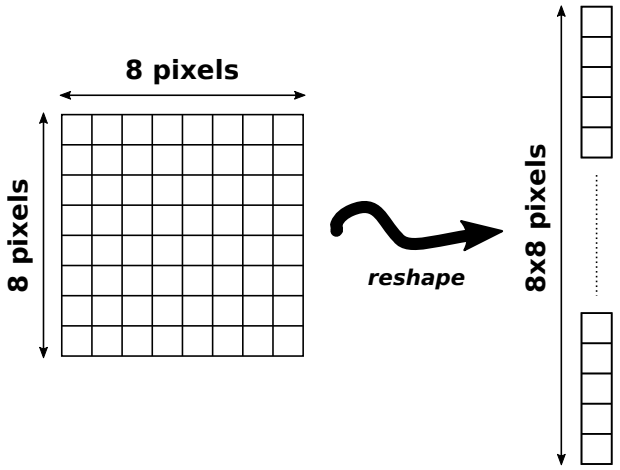
```
y = digits.target
x = digits.images.reshape((len(digits.images), -1))
```

Pour obtenir la taille d'un tableau `numpy`, appelé `tab`, on pourra écrire `tab.shape`.

Question 4. Décrire les éléments suivants et afficher leur taille en utilisant la fonction `shape` :

- `digits.images` est de taille `(1797, 8, 8)` (1797 images de 8 pixels de large et 8 pixels de haut).
- `digits.images[25]` est de taille `(8, 8)` (taille de l'image d'indice 25).
- `x` est de taille `(1797, 64)` (1797 images "réalignées" de 64 pixels de long).
- `x[25]` est de taille `(64)` (taille de l'image "réalignée" d'indice 25).
- `y` est de taille `(1797)` (vecteur de longueur 1797 contenant les labels (ici un entier entre 0 et 9) associé à chaque image).
- `y[25]` est de taille `()` (c'est un entier entre 0 et 9 (longueur "nulle")).

Question 5. Représenter "l'image" avant et après la fonction `reshape`.



2 Utilisation d'un réseau de neurones

On envisage d'abord d'utiliser un réseau de neurones pour la reconnaitre ces chiffres manuscrits.

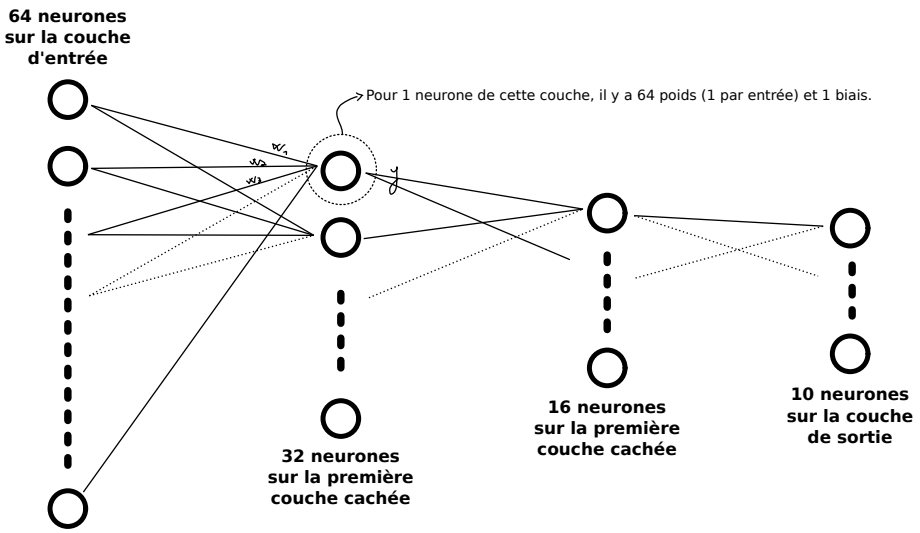
Question 6. Compte-tenu des images utilisées, combien faudra-t-il de neurones dans la couche d'entrée du réseau ?

Il faut autant de neurones sur la couche d'entrée que de caractéristiques dans les données donc ici 64 (correspondant au niveau de coloration de chaque pixel).

Question 7. Combien faudra-t-il de sorties ?

Il faut autant de neurones sur la couche de sortie que de groupes à créer à partir des données donc ici 10 (correspondant au 10 chiffres de 0 à 9).

Question 8. Combien de biais et de poids seront à optimiser ?



Au total, on recense :

$32 \times (64+1)$ paramètres pour la première couche cachée ;

$16 \times (32+1)$ paramètres pour la deuxième couche cachée ;

$10 \times (16+1)$ paramètres pour la couche de sortie.

Au total, on a donc **2778 paramètres**.

Question 9. Quelle est la méthode qui permet de trouver ces paramètres ?

Il s'agit de la méthode de descente de gradient.

Question 10. Quelle fonction d'activation peut-on utiliser pour résoudre ce problème ?

On peut utiliser la fonction sigmoïde (ou échelon).

Pour sélectionner les données d'apprentissage et les données de test, on utilise les lignes ci-dessous. Cela permet de préparer une portion des données pour la phase d'apprentissage (entrées `x_train` et sortie `y_train`) et une portion pour la phase de test (entrées `x_test` et sortie `y_test`) (ici 33% des données seront utilisées pour le test et 67% pour l'apprentissage). L'instruction `shuffle=True` permet de mélanger les données avant la séparation pour éviter d'utiliser une base de données déjà triée.

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y,shuffle=True, test_size=0.33)
```

Question 11. Le réseau choisi est-il adapté à la quantité de données ? Quel phénomène risque d'apparaitre ?

Il y a ici $1797 \times 0.67 \approx 1204$ **données pour l'apprentissage. On a donc plus de paramètres que de données ! Il y a risque de sur-apprentissage.**

Question 12. On fixe maintenant une seule couche cachée de N neurones. Exprimer le nombre de paramètres (poids et biais) en fonction de N.

Au total, on recense :

$N \times (64+1)$ paramètres pour la première couche cachée ;

$10 \times (N+1)$ paramètres pour la couche de sortie.

Au total, on a donc $75 \times N + 11$ **paramètres**.

Question 13. Quelle est la plus petite valeur de N à ne pas dépasser pour éviter le sous-apprentissage ?

Il faut au moins 10 neurones, sinon on mélange l'information pour distinguer dix grandeurs différentes.

On choisit $N = 10$ et on s'impose d'avoir deux fois plus de données d'apprentissage que de paramètres internes (poids et biais).

Question 14. Quels sont les pourcentages (`test_size`) que l'on peut choisir pour valider le fait d'avoir deux fois plus de données d'apprentissage que de paramètres internes.

On impose $75 \times N + 11 < \dfrac{N_{\text{app}}}{2}$ **avec** N_{app} **le nombre de données pour la phase d'apprentissage où** $N_{\text{app}} = t_{app} \times 0.67$ **avec** $t_{app} = 1 - \texttt{test\_size}$**.**

Il faut donc : $t_{app} > \dfrac{2}{1797}.(75 \times +11)$**. Ce qui donne** `test_size` $<$ **15 %.**

Pour créer le modèle, on écrira :

```
from sklearn.neural_network import MLPClassifier ## réseau de neurone pour la classification
mlp = MLPClassifier(hidden_layer_sizes=(10),activation='logistic')
    ## (10) signifie qu'il y a une couche cachée avec 10 neurones
```

```
    ## et 'logistic' signifie qu'on utilise une fonction sigmoide comme fonction d'activation
```

Pour entraîner le modèle, on écrira simplement :

```
mlp.fit(x_train,y_train)
 ### le modèle mlp est prêt à être utilisé !
```

Pour afficher les résultats vis-à-vis des données tests, on écrira enfin :

```
y_pred = mlp.predict(x_test)

from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, y_pred)
print('score = ',mlp.score(x_test, y_test)) ## Score obtenu
print('matrice de confusion =',cm) ## Affichage de la matrice de confusion
```

Question 15. Que contient `y_pred` ?

`y_pred` est le vecteur résultat qui contient les valeurs prédites (0,1,2,3 ... 9).

Question 16. Calculer la justesse de votre algorithme à partir de la matrice de confusion affichée.

On obtient :

```
score =  0.9124579124579124
matrice de confusion = [[59  0  0  0  3  0  1  0  0  0]
 [ 0 59  0  4  1  0  4  3  2  2]
 [ 0  0 61  0  0  0  0  0  0  0]
 [ 0  0  2 51  0  0  0  1  0  0]
 [ 0  0  0  0 45  0  0  0  0  3]
 [ 0  0  0  0  0 54  0  0  0  0]
 [ 0  0  0  0  0  0 59  0  0  0]
 [ 0  0  0  0  2  0  0 58  0  2]
 [ 2  4  0  0  1  1  1  0 45  9]
 [ 0  0  0  2  0  1  0  1  0 51]]
```

On définit le taux de bonnes prédictions d'images représentant le chiffre i :

$$t_i = \frac{\text{nb. de bonnes prédictions du chiffre } i}{\text{nb. d'images du chiffre } i} \quad \text{avec } i \in [0,9]$$

Question 17. Calculer les taux t_i puis les classer dans l'ordre décroissant.

Par exemple, on a ici $t_8 = \dfrac{45}{2+4+1+1+1+45+9}$.

On peut également évaluer l'intérêt du réglage du taux d'apprentissage (*learning rate*) en rajoutant dans le modèle :

```
mlp = MLPClassifier(hidden_layer_sizes=(10),activation='logistic',learning_rate_init=0.001)
```

(ici on impose un taux d'apprentissage de 0.001)

On peut également afficher le temps d'apprentissage des paramètres en modifiant dans le code précédent :

```
from time import time
start = time()
mlp.fit(x_train,y_train)
t_app = time()- start
```

Question 18. Faire quelques simulations (environ 5 pour chaque taux d'apprentissage) et remplir le tableau ci-dessous puis conclure.

Sans rentrer dans le détail, augmenter le *learning rate* réduit le temps d'apprentissage mais détériore la justesse et peut poser des problèmes de convergence.

3 Utilisation des algorithme des k-plus proches voisins

Question 19. Reprendre le problème en le traitant avec l'algorithme des k-plus proches voisins. Celui-ci est disponible dans la bibliothèque `scikit learn` en écrivant :

```
from sklearn.neighbors import KNeighborsClassifier
mlp = KNeighborsClassifier(n_neighbors=3)
 ## Création d'un modèle k-NN avec 3 voisins
```

Question 20. Faire quelques tests en variant le nombre de voisins. Discuter du résultat obtenu.

...

4 Il vous reste du temps ?

Vous pouvez traiter le problème de la classification des iris (vu en cours) en important les données avec l'instruction :

```
iris = datasets.load_iris()
```