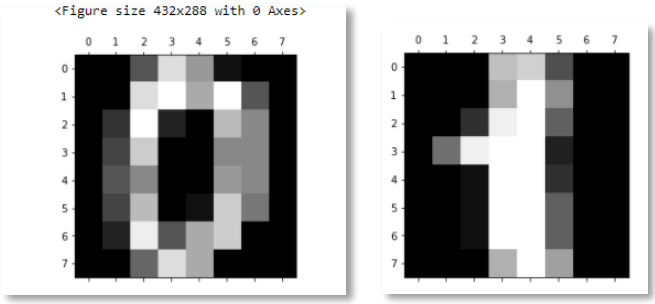
Réseaux de neurones : TP3

Loedel Pierric – Nadaud Edouard / 4AUFA 41

Nous commençons par importer le dataset et déterminer la dimension D des données et le nombre d'exemples par classe. Ci-après, quelques exemples en image :



b) Séparer une fois pour toutes la base initiale en deux : apprentissage (70%) et test (30%) (model_selection.train_test_split) :

Pour cela nous faisons la commande habituelle

2) Apprentissage

clf1 = MLPClassifier(hidden_layer_sizes=C, activation='tanh', solver='sgd', batch_size=1, alpha=0, learning_rate='adaptive', verbose=1)

Le maximum de neurones dans la couche cachée est de 24 pour éviter le sur apprentissage (Nbr de paramètre libre <NB_d'exemple_dans_apprentissage*N)(nombre de sortie).

*Nous avons 264 classes et 1797 images dans notre base d'apprentissage soit NB_d'exemple_dan s_apprentissage*N(nombre de sortie)=17970.*

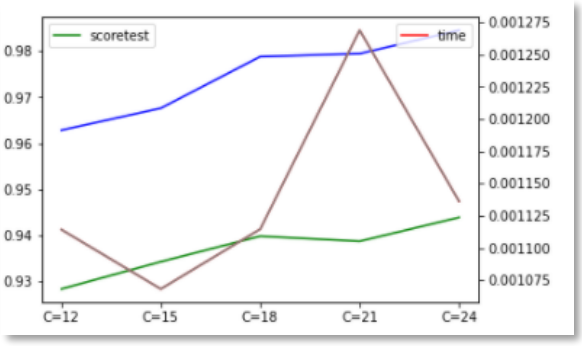
Nbr de paramètre libre=C*10+C*64=C*74 soit C< 17970/74 =24

« Le nombre que j'ai trouvé me semble petit néanmoins j'utiliserai cette valeur car mon ordinateur a une faible puissance de calcul pour aller plus haut. Je ne vois pas mon erreur dans mon calcul (si cela est possible, j'aimerais bien un retour pour progresser). »

De plus comme lors du tp précédent j'enregistrerai toutes les données dans des Excel pour faciliter les calculs en local plus tard.

La reconnaissance d'image étant aujourd'hui très performante, j'attends un taux de reconnaissance haut pour mon programme, étant donné que mon ordinateur est lent à faire les calculs, j'estimerai que mon modèle est correct avec 95 pourcents de réussite en moyenne sur chaque classe. Sachant qu'un neurone ne serait pas suffisant au vu du tp précédent, je commencerai à 12 neurones. Je fais la moyenne sur 10 entrainements avec un saut de 3 neurones dans ma hiden layer. Je fais le choix de 10 entrainements pour avoir une moyenne et donc eviter un taux d'erreur trop haut au premier essai par « chance »

Nous enregistrons (save) puis récupérons nos datas, comme vous pourrez le voir dans le notebook.



Les résultats obtenus vous sont montrés et expliqués ci-après :

Nous choisissons C = 15 au vu de nos résultats.
En effet, avec ce choix, nous minimisons le temps et la variance (résilience de notre modèle) entre test et train. Néanmoins, notre taux d'erreur reste important pour un développement à grande échelle sachant qu'aujourd'hui, le taux de reconnaissance est proche de 100%.

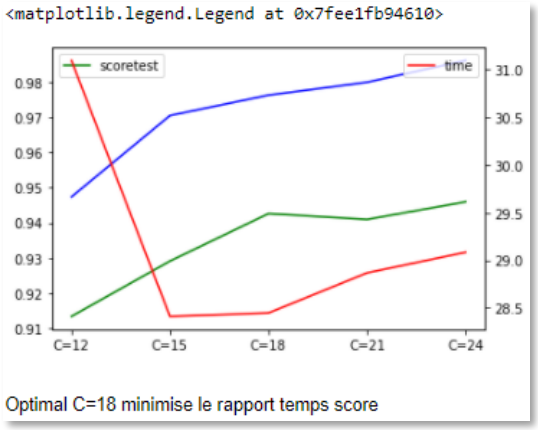
Néanmoins comme expliqué je me satisfais avec un taux de 95 pourcent en train.
En effet, la moyenne du score entre 15 et 24 est négligeable, comparée au temps de calcul.

En bleu nous avons le score en train, en vert en test et en rouge le temps de calcul moyen.
L'ordonnée de droite représente l'accuracy et l'ordonnée de droite le temps de calcul en heure.

3. Cross-validation

Je décide d'aller de 3 en 3 pour les neurones comme précedement. Nous remarquons que les resultats en cross validation sont meilleur en train comme en test néanmoins cela reste minoritaire.
Cepandant le temps d'exécution est lui beaucoup plus long.
Les résultats obtenus sont tels que :

```
<matplotlib.legend.Legend at 0x7fee1fb94610>
```



Optimal C=18 minimise le rapport temps score

```
print(f"le score en train est de : {model1.score(X_train,y_train)}")
print(f"le score en test est de : {model1.score(X_test,y_test)}")
```

```
le score en train est de : 0.9817024661893397
le score en test est de : 0.9537037037037037
```

Dans cette matrice de confusion,
Nous pouvons remarquer que le plus grand nombre d'erreurs se trouve pour les valeurs 1 et 8.
De plus, nous pouvons observer que le 1 est régulièrement confondu avec un 5 (3 fois) et le 8 avec un neuf (4 fois). L

```
metrics.confusion_matrix(y_test,y_pred_test)
```

```
array([[53,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0, 48,  2,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0, 45,  2,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  0, 51,  0,  1,  0,  0,  2,  0],
       [ 0,  0,  0,  0, 59,  0,  1,  0,  0,  0],
       [ 0,  3,  0,  0,  0, 60,  1,  0,  1,  1],
       [ 1,  0,  0,  0,  0,  0, 52,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0, 54,  0,  1],
       [ 0,  1,  0,  0,  0,  0,  0,  1, 41,  0],
       [ 0,  0,  1,  1,  0,  1,  0,  0,  4, 52]])
```

```
erreurarraytrain
```

```
[0, 5, 0, 0, 3, 2, 0, 2, 6, 5]
```

```
metrics.confusion_matrix(y_train,y_pred_train)
```

```
array([[125,   0,   0,   0,   0,   0,   0,   0,   0,   0],
       [  0, 129,   0,   0,   2,   0,   0,   0,   0,   1],
       [  0,   0, 128,   0,   0,   0,   0,   0,   2,   0],
       [  0,   0,   0, 125,   0,   0,   0,   0,   3,   1],
       [  0,   0,   0,   0, 121,   0,   0,   0,   0,   0],
       [  0,   0,   0,   0,   0, 114,   0,   0,   0,   2],
       [  0,   0,   0,   0,   0,   0, 128,   0,   0,   0],
       [  0,   0,   0,   0,   1,   0,   0, 123,   0,   0],
       [  0,   5,   0,   0,   0,   0,   0,   1, 124,   1],
       [  0,   0,   0,   0,   0,   2,   0,   1,   1, 117]])
```

Dans cette matrice de confusion, nous pouvons remarquer que le plus grand nombre d'erreur se trouve pour les valeurs 1 et 9. Toujours de plus, nous pouvons observer que le 1 est régulièrement confondu avec un 9 (5 fois), et le 8 avec un 3 (3 fois).

```
erreurarraytest
```

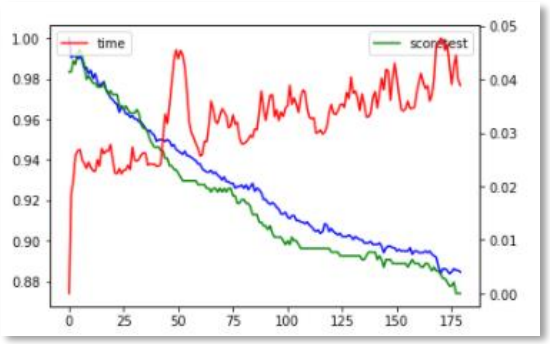
```
[1, 4, 3, 3, 0, 2, 2, 1, 7, 2]
```

KNN :

```
X.shape
```

```
(1797, 64)
```

Notre K va varier de 0 a 1797/10 soit Kmax=180 apres on sera en surentrainement

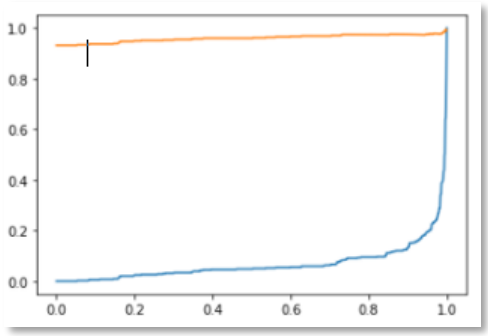


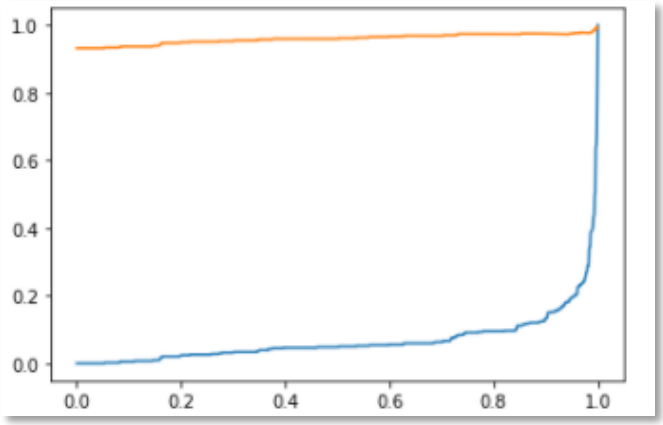
Ici, notre K optimal se situe aux environs de 15. En effet, plus le K augmente, plus le temps de calcul augmente et la précision baisse en test et en train. Notre modèle a trop de voisins, et donc de distance où il se trompe.

4. Rejet

Plus le taux de rejet est important plus le taux de reconnaissance augmente. Néanmoins, cela n'est pas immense mais peut permettre d'améliorer le modèle. En effet, l'avantage d'un réseau de neurones MLP se situe lorsque nous avons des data set immenses (pour notre dataset un KNN serait plus efficace) mais si ce modèle devait rentrer en production, alors il serait bien de maximiser le rejet (créer une data base de données rejetées (donc beaucoup plus petite)) que la base initial que l'on reclassifierait grâce à un KNN, un algorithme plus performant mais plus long en exécution sur des tres grand dataset car il se doit de recalculer toute les distances pour chaque data. Cette association augmenterait significativement le nombre de données bien classées en minimisant le temps :

Puis nous obtenons finalement le même résultat :





5. Cascade de classifieurs :

Pour cela, nous commençons par exécuter le réseau de neurones, puis nous entraînons le KNN :

```
#on execute notre reseau de neuronne:
model5= MLPClassifier(hidden_layer_sizes=18, activation='tanh',solver='sgd', batch_size=1, alpha=0, learning_rate='adaptive', ver
model5.fit(X_train,y_train)
```

```python
#entrainement du KNN
modelKNN=KNeighborsClassifier(n_neighbors=15, algorithm='brute'
modelKNN.fit(X_train,y_train)
```

```python
compteur=0
erreurarraytrain=[]
for i in range(10):
    erreur=0
    for compteur in range(10):
        if (compteur != i):
            erreur=erreur + matriceconfusionpremieretage[compteur][i]
            compteur=compteur+1
        else:
            compteur=compteur+1
    erreurarraytrain.append(erreur)
    compteur=0

print(f"nous avons {sum(erreurarraytrain)} donnée mal classé ")
```

```python
#realisation des prediction uniquement grace au model MLP entrainé
y_pred_test=model5.predict(X_test)
#on save la matrice de confusion
matriceconfusionpremieretage=metrics.confusion_matrix(y_test,y_pred_test)
```

```python
matriceconfusionpremieretage
```

```
array([[51,  0,  0,  0,  1,  0,  1,  0,  0,  0],
       [ 0, 46,  4,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  2, 41,  3,  0,  0,  0,  0,  1,  0],
       [ 0,  0,  4, 48,  0,  1,  0,  0,  1,  0],
       [ 0,  1,  0,  0, 58,  0,  1,  0,  0,  0],
       [ 0,  0,  0,  3,  1, 61,  0,  0,  1,  0],
       [ 1,  1,  0,  0,  0,  0, 51,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0, 54,  0,  1],
       [ 0,  2,  0,  1,  0,  1,  0,  0, 39,  0],
       [ 0,  1,  0,  1,  0,  1,  0,  1,  3, 52]])
```

```python
#rejet
y_pred_test1=model5.predict_proba(X_test)
rejet=argmax_reject_threshold(y_pred_test1,0.9)
matriceconfusionrejet=metrics.confusion_matrix(y_test,rejet)
```

```python
print(f"nous avons {sum(rejet==-1)} exemples rejeté")
```

Nous avons 39 données mal classées.
Sans compter les données rejetées.

Nous avons 46 exemples rejetés. J'ai mis un taux de rejet très élevé car j'ai un petit dataset je peux donc me le permettre car meme a un taux élevé la vitesse de calcul reste élevé:

Nous pouvons voir que : 2-->0, 18-->1, 4-->2 1-->3, 2-->4, 3-->5, 3-->6, 3-->7, 5-->8, 5-->9 La probabilité n'a pas été suffisante pour décider de leur classification. KNN ayant un meilleur taux de reconnaissance mais un temps plus long (si nous avions un dataset de centaines de milliers de données), nous ne l'utilisons qu'en deuxième recours pour décider sur 46 données (8.5% du dataset (le threshold mis étant très haut pour minimiser l'erreur)).

```python
matriceconfusionrejet
```

```
array([[ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 2, 50,  0,  0,  0,  1,  0,  0,  0,  0,  0],
       [18,  0, 30,  2,  0,  0,  0,  0,  0,  0,  0],
       [ 4,  0,  0, 40,  2,  0,  0,  0,  0,  1,  0],
       [ 1,  0,  0,  4, 47,  0,  1,  0,  0,  1,  0],
       [ 2,  0,  1,  0,  0, 56,  0,  1,  0,  0,  0],
       [ 3,  0,  0,  0,  0,  1, 61,  0,  0,  1,  0],
       [ 3,  1,  0,  0,  0,  0,  0, 49,  0,  0,  0],
       [ 3,  0,  0,  0,  0,  0,  0,  0, 51,  0,  1],
       [ 5,  0,  0,  0,  0,  0,  1,  0,  0, 37,  0],
       [ 5,  0,  1,  0,  0,  0,  0,  0,  0,  2, 51]])
```

Nous faisons un predict avec notre KNN et notre KOPI = 20 :

```python
indexrejete=(np.where(rejet==-1))
#ON FAIT UN PREDICT AVEC NOTRE KNN ET NOTRE KOPI=20
predictKNN=modelKNN.predict(X_test[indexrejete])
```

Ensuite, nous décidons de remplacer les predict du KNN dans le predict avec rejet pour faire la dernière matrice de confusion :

Nous obtenons 0 exemple rejeté.
Ceci est normal, puisque nous les avons tous reclassés grâce au KNN.

```python
for compteur in range(len(indexrejete[0])):
    rejet[indexrejete[0][compteur]]=predictKNN[compteur]
```

```python
rejetinitial=sum(rejet==-1)
print(f"nous avons {sum(rejet==-1)} exemples rejeté")
```

Enfin, la matrice de confusion finale nous donne :

```
matriceconfusionfinal=metrics.confusion_matrix(y_test,rejet)
```

```
matriceconfusionfinal
```

```
array([[52,  0,  0,  0,  1,  0,  0,  0,  0,  0],
       [ 0, 48,  2,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0, 44,  2,  0,  0,  0,  0,  1,  0],
       [ 0,  0,  4, 48,  0,  1,  0,  0,  1,  0],
       [ 0,  1,  0,  0, 58,  0,  1,  0,  0,  0],
       [ 0,  0,  0,  0,  1, 64,  0,  0,  1,  0],
       [ 1,  0,  0,  0,  0,  0, 52,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0, 54,  0,  1],
       [ 0,  1,  0,  0,  0,  1,  0,  0, 41,  0],
       [ 0,  1,  0,  0,  0,  1,  0,  2,  2, 53]])
```

```
compteur=0
erreurarraytrain=[]
for i in range(10):
    erreur=0
    for compteur in range(10):
        if (compteur != i):
            erreur=erreur + matriceconfusionfinal[compteur][i]
            compteur=compteur+1
        else:
            compteur=compteur+1
    erreurarraytrain.append(erreur)
    compteur=0
erreur=sum(erreurarraytrain)/100
print(f"nous avons {sum(erreurarraytrain)} donnée mal classé ")
print(f"notre taux d'erreur est de : {erreur*100/540}")
print(f"notre taux d'erreur est passé de :{46*100/540} a : {erreur*100/540}")
```

Nous avons maintenant 26 données mal classées.
Notre taux d'erreur est de : 0.04814814814814815.
Notre taux d'erreur est passé de 8.518518518518519 à 0.04814814814814815.