Yannis VAULRY
Kevin SERI

Travail sur le jeu de données MNIST

Dépôt GitHub : https://github.com/Keevin350/TP_MNIST

Introduction :

MNIST a été développé par les précurseurs du deep learning, Y. LeCun et Y. Bengio, en 1998. Il contient des données d'écriture manuelle des chiffres de 0 à 9. Il mène généralement à un problème de classification multi-classes à 10 classes. Dans sa forme initiale, l'échantillon d'apprentissage comporte 60000 exemples, et 10000 en test.

Un exemple en entrée est une image de taille fixe 28x28, chaque pixel étant blanc (0) ou noir (1). Le chiffre est centré dans l'image (fig. 2).

Figure 2 – Quelques exemples de chiffres manuscrits de MNIST. Un exemple est donc un vecteur de 28*28=784 composantes correspondant à un niveau de gris pour chacun des 784 pixels.

Objectif

Le but de ce TP est d'utiliser le dataset MNIST disponible dans sklearn pour réaliser un apprentissage robuste qui sera capable de prédire les chiffres écrits dans les différentes images.

Partie 1 : Import et visualisation des données

Pour importer le dataset MNIST, nous allons utiliser la libraire sklearn et plus précisément la fonction fetch_openml. Nous importons, aussi de nombreuses librairies pour la suite du projet.

Nous pouvons, visualiser les données du dataset avec l'attribut data. On observe que le dataset est bien composé de 784 features et dispose de 70000 lignes.

Entrée [1]:
```
import warnings
from sklearn.datasets import fetch_openml
from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import train_test_split, GridSearchCV
import pandas as pd
import numpy as pd
from sklearn.preprocessing import StandardScaler

warnings.simplefilter("ignore")
mnist = fetch_openml("mnist_784")
```

Entrée [2]:
```
mnist.data
```

Out[2]:

	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	pixel10	...	pixel775	pixel776	pixel777	pixel778
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
69995	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
69996	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
69997	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
69998	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0
69999	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0

Afin d'être sûr, nous avons affiché les différentes shape du dataset MNIST (target et data).

Entrée [3]:
```
print(mnist.target.shape)
print(mnist.data.shape)
```

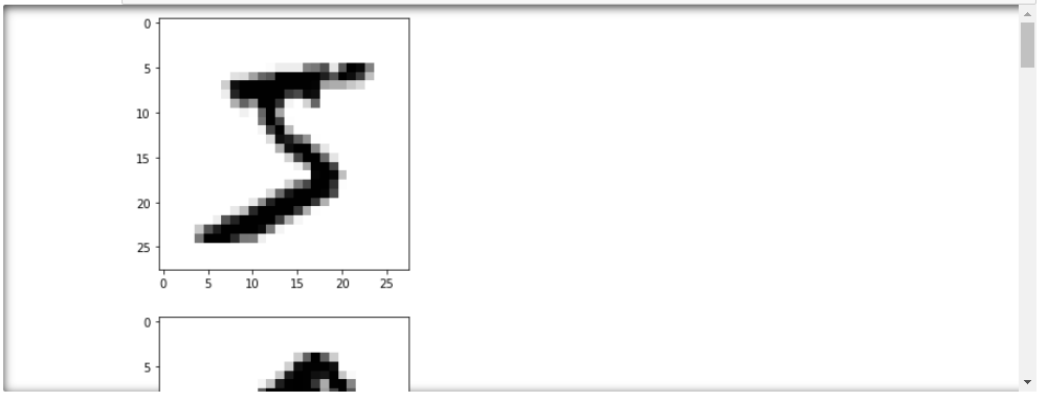
```
(70000,)
(70000, 784)
```

Il est possible de représenter graphiquement les différentes données du dataset, grâce à la fonction data.to_numpy(). Nous avons donc affiché les dix premières images du dataset.

Entrée [4]:
```
import matplotlib.pyplot as plt

lotImage = mnist.data.to_numpy()

for i in range(10):
    plt.imshow((lotImage[i].reshape(28,28)), cmap=plt.cm.gray_r)
    plt.show()
```



Partie 2 : Création du modèle d'apprentissage

Afin de créer un modèle d'apprentissage, nous avons besoin de récupérer les 60000 premiers enregistrements pour notre de base d'apprentissage et les 10000 données restantes seront utilisées pour la base de test.
Dans la partie du code ci-dessus, nous importons la librairie time et la fonction accuarcy_score() pour évaluer nos différents modèles créés. On vérifie par ailleurs que la taille des bases de test et d'apprentissage contiennent le bon nombre d'éléments.

Entrée [5]:
```
import time
from sklearn.metrics import accuracy_score

X_train = mnist.data[:60000]
X_test = mnist.data[60000:]
y_train = mnist.target[:60000]
y_test = mnist.target[60000:]

print("X_train : ", len(X_train), " y_train : ", len(y_train))
print("X_test : ", len(X_test), " y_test : ", len(y_test))
```

```
X_train :  60000  y_train :  60000
X_test :  10000  y_test :  10000
```

Nous décidons de créer un modèle MLPClassifier sans renseigner d'hyperparamètres à l'aide ci-dessous. Cela nous permet d'avoir approximativement le temps d'exécution ainsi qu'un score qui peut être potentiellement amélioré avec des hyperparamètres optimaux pour le MLPClassfier.

Entrée [6]:
```
start_time = time.time()
mlp = MLPClassifier()
mlp.fit(X_train, y_train)
prediction = mlp.predict(X_test)
print(accuracy_score(y_test, prediction))
print("--- %s seconds ---" % (time.time() - start_time))
```

```
0.9656
--- 161.75504803657532 seconds ---
```

Le score obtenu précédemment peux être amélioré en standardisant les données. Nous avons donc utilisé la fonction standardScaler afin de normaliser les données de test et d'apprentissage. Nous remarquons que le score est un peu supérieur au score obtenu précédemment, cependant le temps d'exécution est moins long.

Entrée [30]:
```
scaler = StandardScaler()
X_train_scaler = scaler.fit_transform(X_train)
X_test_scaler = scaler.fit_transform(X_test)

start_time = time.time()
mlp = MLPClassifier()
mlp.fit(X_train_scaler, y_train)
prediction = mlp.predict(X_test_scaler)
print(accuracy_score(y_test, prediction))
print("--- %s seconds ---" % (time.time() - start_time))
```

```
0.975
--- 49.234519720077515 seconds ---
```

Une fonction existe sous sklearn qui nous permet de récupérer d'obtenir les hypermètres optimaux pour les algorithmes de machine learning. Cette fonction se nomme GridSearchCV(). Nous avons créé une variable qui contient les différentes valeurs des hyperparamètres à tester.

Le problème avec cette fonction est que nous obtenons un temps d'exécution long, car nous essayons toutes les valeurs des hyperparamètres. Afin de tenter de raccourcir le temps d'exécution, nous avons donné au paramètre cv (cross-validation) la valeur 3 et nous avons donné à la valeur n_jobs la valeur 4 afin d'utiliser les 4 cœurs du processeur.

Après plus de 9 heures d'exécution, nous avons obtenu les hyperparamètres optimaux suivants :

Entrée [34]:
```
parametres = {'hidden_layer_sizes':[(50,),(50,50),(50,50,50)],
              'activation':["identity","logistic","tanh","relu"],
              'solver':["lbfgs","sgd","adam"],
              'alpha':[0.0000001,0.000001,0.00001,0.0001,0.001,0.01,0.1],
              'learning_rate':["constant","invscaling","adaptive"]}

mlp_clf = MLPClassifier()
gridS_mlp = GridSearchCV(mlp_clf, parametres, cv=3, n_jobs=4)
gridS_mlp.fit(X_train_scaler, y_train)
```

Entrée [18]:
```
print(gridS_mlp.best_params_)
```
```
{'activation': 'relu', 'alpha': 0.1, 'hidden_layer_sizes': (50, 50), 'learning_rate': 'adaptive', 'so
lver': 'adam'}
```

Nous allons maintenant tester les hyperparamètres optimaux sur un MLPClassifier. Nous précisons que dans variable paramètres nous avons donné des valeurs arbitraires pour hidden_layer_sizes. D'après notre gridsearch, la valeur optimale pour hidden_layer_sizes serait un réseau de neurones de 2 couches cachées de 50 neurones.

Entrée [22]:
```
start_time = time.time()
mlp = MLPClassifier(hidden_layer_sizes = (50, 50), activation="relu", solver="adam",alpha =0.1 ,learni
mlp.fit(X_train_scaler, y_train)
prediction = mlp.predict(X_test_scaler)
print(accuracy_score(y_test, prediction))
print("--- %s seconds ---" % (time.time() - start_time))
```
```
0.975
--- 80.16264581680298 seconds ---
```

Le temps d'exécution est de 80 secondes et nous obtenons un accuracy_score de 0.975. Nous avons décidé de tester plusieurs tirages afin de voir si notre MLPClassier pouvait obtenir un score plus grand. Ces tirages ont été réalisé avec une boucle for de 20 tours.

```
Entrée [28]: MaximumScore = 0

for i in range(0,20):

    start_time = time.time()
    mlp = MLPClassifier(hidden_layer_sizes = (50, 50), activation="relu", solver="adam",alpha =0.1 ,lea
    mlp.fit(X_train_scaler, y_train)
    prediction = mlp.predict(X_test_scaler)

    if MaximumScore < accuracy_score(y_test, prediction):
        MaximumScore = accuracy_score(y_test, prediction)
        tempsExec = time.time() - start_time
        print(accuracy_score(y_test, prediction))
        print("--- %s seconds ---" % (time.time() - start_time))
```

```
0.9733
--- 66.92190551757812 seconds ---
0.9736
--- 54.65928030014038 seconds ---
0.977
--- 83.78791618347168 seconds ---
```

Nous remarquons que le score s'est un peu amélioré et le temps d'exécution reste globalement le même.

Nous avons décidé de faire varier le nombre de neurone disponible dans les 2 couches cachées de notre réseau de neurones afin de voir si le nombre de neurones par couche impactait la qualité de notre modèle d'apprentissage et augmentait notre accuracy_score (prédiction).

```
Entrée [32]: MaximumScore = 0

for i in range(0,10):

    start_time = time.time()
    mlp = MLPClassifier(hidden_layer_sizes = (100, 100), activation="relu", solver="adam",alpha =0.1 ,
    mlp.fit(X_train_scaler, y_train)
    prediction = mlp.predict(X_test_scaler)

    if MaximumScore < accuracy_score(y_test, prediction):
        MaximumScore = accuracy_score(y_test, prediction)
        tempsExec = time.time() - start_time
        print(accuracy_score(y_test, prediction))
        print("--- %s seconds ---" % (time.time() - start_time))
```

```
0.9764
--- 118.79019117355347 seconds ---
0.9779
--- 109.41379451751709 seconds ---
0.9788
--- 94.53426766395569 seconds ---
0.9795
--- 137.07823491096497 seconds ---
```

Nous avons score à 0.9795, ce score est le plus élevé. Nous remarquons que le temps d'exécution est plus long avec 100 neurones par couches cachée qu'avec 50 neurones par couche cachée.

Nous avons décidé de tester avec 200 neurones par couche cachée, mais le score obtenu est moins élevé que précédemment.

```
Entrée [45]: MaximumScore = 0

for i in range(0,10):

    print(i)
    start_time = time.time()
    mlp = MLPClassifier(hidden_layer_sizes = (200, 200), activation="relu", solver="adam",alpha =0.1 ,
    mlp.fit(X_train_scaler, y_train)
    prediction = mlp.predict(X_test_scaler)

    if MaximumScore < accuracy_score(y_test, prediction):
        MaximumScore = accuracy_score(y_test, prediction)
        print(accuracy_score(y_test, prediction))
        print("--- %s seconds ---" % (time.time() - start_time))
```

```
0
0.975
--- 114.3752863407135 seconds ---
1
2
0.979
--- 193.7587184906006 seconds ---
3
4
5
6
7
8
9
```

En essayant de trouver une solution pour obtenir un modèle d'apprentissage et un score plus élevé. Nous avons décidé de réaliser à nouveau un nouveau gridSearch sur hidden_layer_sizes, alpha et learning_rate.

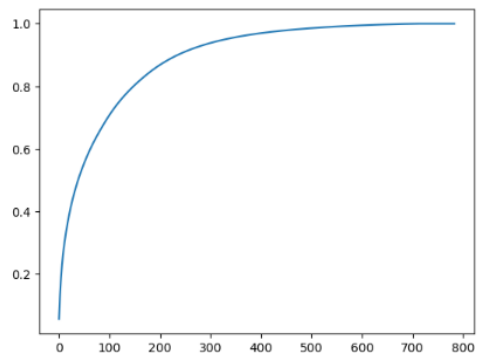
Notre dataset étant plutôt grand, nous avons décidé de voir s'il était possible de réduire de dimension tout en gardant une variance correcte aux alentours d'au moins 95%. C'est pour cela que nous avons utiliser un PCA afin de réduire les dimensions de X_train (base d'apprentissage).

D'après des valeurs cumulées explained_variance_ratio_, pour garder 95 % de la variance nous avons décidé de faire un PCA avec n_components=500.

```
Entrée [8]: from sklearn.decomposition import PCA
            import numpy as np

            pca = PCA(n_components=784)
            X_train_reduction_scaler = pca.fit_transform(X_train_scaler)
            plt.plot(np.cumsum(pca.explained_variance_ratio_))

Out[8]: [<matplotlib.lines.Line2D at 0x2aa1c7e02c8>]
```



En réalisant ce nouveau gridSearch, nous obtenons ces hyperparamètres optimaux :
{'alpha': 0.1, 'hidden_layer_sizes': (50,), 'learning_rate': 'invscaling'}

```
Entrée [9]:
pca = PCA(n_components=500)
X_train_reduction_scaler = pca.fit_transform(X_train_scaler)

parametres = {'hidden_layer_sizes':[(50,),(50,50),(50,50,50)],
              'alpha':[0.0000001,0.000001,0.00001,0.0001,0.001,0.01,0.1],
              'learning_rate':["constant","invscaling","adaptive"]}

mlp_clf = MLPClassifier()
gridS_mlp = GridSearchCV(mlp_clf, parametres, cv=2, n_jobs=-1)
gridS_mlp.fit(X_train_reduction_scaler, y_train)

Out[9]:
GridSearchCV(cv=2, estimator=MLPClassifier(), n_jobs=-1,
             param_grid={'alpha': [1e-07, 1e-06, 1e-05, 0.0001, 0.001, 0.01,
                                   0.1],
                         'hidden_layer_sizes': [(50,), (50, 50), (50, 50, 50)],
                         'learning_rate': ['constant', 'invscaling',
                                           'adaptive']})

Entrée [10]:
print(gridS_mlp.best_params_)

{'alpha': 0.1, 'hidden_layer_sizes': (50,), 'learning_rate': 'invscaling'}
```

Nous avons ensuite essayé de réaliser un MLPClassifier avec ces hypermètres optimaux. Nous obtenons un score d'accuracy de 0.9761 avec un temps d'exécution de 214 secondes.

```
Entrée [23]:
start_time = time.time()
mlp = MLPClassifier(hidden_layer_sizes = (50,), alpha =0.1 ,learning_rate = "invscaling")
mlp.fit(X_train_scaler, y_train)
prediction = mlp.predict(X_test_scaler)
print(accuracy_score(y_test, prediction))
print("--- %s seconds ---" % (time.time() - start_time))

0.9761
--- 214.3872811794281 seconds ---
```

Nous avons réalisé une boucle de 5 tirages dans laquelle nous avons augmenté le nombre de neurones dans la couche cachée afin de voir si comme dans notre test réalisé plus haut, si le nombre de neurones influence notre temps d'exécution et notre score final.

```
Entrée [26]:
MaximumScore = 0

for i in range(0,5):
    print(i)
    start_time = time.time()
    mlp = MLPClassifier(hidden_layer_sizes = (500,), alpha =0.1 ,learning_rate = "invscaling")
    mlp.fit(X_train_scaler, y_train)
    prediction = mlp.predict(X_test_scaler)

    if MaximumScore < accuracy_score(y_test, prediction):
        MaximumScore = accuracy_score(y_test, prediction)
        print(accuracy_score(y_test, prediction))
        print("--- %s seconds ---" % (time.time() - start_time))

0
0.9792
--- 305.7928705215454 seconds ---
1
2
3
0.9795
--- 625.4932098388672 seconds ---
4
0.9796
--- 529.3072807788849 seconds ---
```

Nous obtenons un meilleur score que le score le plus élevé que nous avions (pour rappel : 0.9795), cependant nous avons multiplié le temps d'exécution du modèle d'apprentissage par 3.5.

Nous remarquons que le gain est très négligeable par rapport au temps d'exécution. Si nous travaillons sur un dataset avec plus de lignes le temps sera encore plus long avec ce modèle d'apprentissage.

D'après la documentation, le MLPClassifier dispose d'un hyperparamètre nommé random_state, ce paramètre est un peu embêtant dans notre cas car il nous empêche d'avoir un score fixe sur un modèle d'apprentissage, ce qui signifie qu'à chaque relance du modèle, il est possible d'avoir un nouveau score. Nous avons donc placé un random_state = 1 afin de toujours retomber sur le même modèle et le même score (nous n'avons plus besoin de réaliser plusieurs tirages). Nous remarquons qu'en effectuant cela nous arrivons à obtenir un score = 0.9799.

Nous avons donc un meilleur score comparé au score précédent (0.9796).

Afin d'obtenir encore un autre meilleur score il faudrait analyser le nombre de neurones à placer dans la couche cachée.

```
Entrée [7]: start_time = time.time()
            mlp=MLPClassifier(random_state=1, max_iter=200, hidden_layer_sizes=(500,),learning_rate = "invscaling")
            mlp.fit(X_train_scaler, y_train)
            prediction = mlp.predict(X_test_scaler)
            print(accuracy_score(y_test, prediction))
            print("--- %s seconds ---" % (time.time() - start_time))

            0.9799
            --- 180.79276251792908 seconds ---
```

Par la suite, nous avons testé un modèle d'apprentissage MLPClassifier avec random_state = 1, nous avons diminué le nombre d'itérations de 200 à 100. Et nous avons récupéré la valeur alpha par défaut (qui est dans la documentation MLPClassfier). Nous pouvons voir que nous obtenons un score à 0.9802, cependant le temps d'exécution est plus long que le temps d'exécution du score qui est 0.9799.

```
Entrée [7]: start_time = time.time()
            mlp=MLPClassifier(solver='adam',random_state=1, max_iter=100, hidden_layer_sizes=(500,),alpha=1e-6)
            mlp.fit(X_train_scaler, y_train)
            prediction = mlp.predict(X_test_scaler)
            print(accuracy_score(y_test, prediction))
            print("--- %s seconds ---" % (time.time() - start_time))

            0.9802
            --- 335.47805285453796 seconds ---
```

Conclusion :

Pour répondre à la question du TP. Nous avons réussi à battre tous les scores du tableau de l'énoncé du TP sauf les deux dernières lignes qui sont avec une approche MLP 1-HL-300 et une approche MLP 2-HL-500-300 cross-ent, softmax.

Approche	Preprocessing	ACC	Ref
perceptron	aucun	12.0	LeCun 1998
perceptron	oui	8.4	LeCun 1998
MLP 1-HL-300	aucun	3.6	LeCun 1998
MLP 1-HL-300	oui	1.6	LeCun 1998
MLP 2-HL-500-300 cross-ent, softmax	aucun	1.53	Hinton 2005

Nous rappelons que le score le plus élevé que nous avons obtenu est de 0.9802. Ce score a été obtenu en standardisant les données pour ensuite effectuer un gridsearch sur trois hyperparamètres :

- Learning_rate ;
- Hidden_layer_sizes ;
- Alpha.

Nous avons ensuite rajouter un random_state = 1 et max_iter divisé par 2 soit 100.