

Reprise du projet 5 avec le jeux MNIST original

Imports et fonctions

```
Entrée [1]: import numpy as np

import random

from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report
from sklearn.preprocessing import StandardScaler, MinMaxScaler, RobustScaler
from sklearn.model_selection import learning_curve
from sklearn.pipeline import make_pipeline, Pipeline
from sklearn.neighbors import KNeighborsClassifier

import h5py

import time

import matplotlib.pyplot as plt

from joblib import dump, load

from PIL import Image

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D
```

2022-03-28 18:08:26.370698: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libcudart.so.11.0'; dLError: libcudart.so.11.0: cannot open shared object file: No such file or directory
2022-03-28 18:08:26.370725: I tensorflow/stream_executor/cuda/cudart_stub.cc:29] Ignore above cudart dLError if you do not have a GPU set up on your machine.

```
Entrée [2]: def erreur_prediction(valeurs_predites= None, valeurs_reelle= None):
    """
    Vérifie que les valeurs prédites sont les mêmes que les valeurs réelles.

    paramètres en entrée:
    valeurs_predites
    valeurs_reelle

    Paramètres de sortie:
    erreur: nombre d'erreur de prédiction
    precision: 100*(1-erreur/(longueur des tableaux))
    liste_erreur: Liste de toutes les erreurs détecté sous forme de dictionnaire
                  clé valeur réelle: valeur prédite
    """

    assert len(valeurs_predites) == len(valeurs_reelle), "Erreur de dimension: la dimension des tableaux "+\
        f"doit être identique. {len(valeurs_predites)} # de {len(valeurs_reelle)}"

    erreur= 0
    d= {}
    longueur= len(valeurs_reelle)
    for i in range(longueur):
        if valeurs_reelle[i]!= valeurs_predites[i]:
            if valeurs_reelle[i] not in d.keys():
                d[valeurs_reelle[i]]= [valeurs_predites[i]]
            else:
                d[valeurs_reelle[i]].append(valeurs_predites[i])
            erreur+= 1

    return erreur, 100*(1-erreur/longueur), d
```

Reprise partielle du brief

```
Entrée [3]: # Importation des données depuis scikit learn datasets
digits = load_digits()

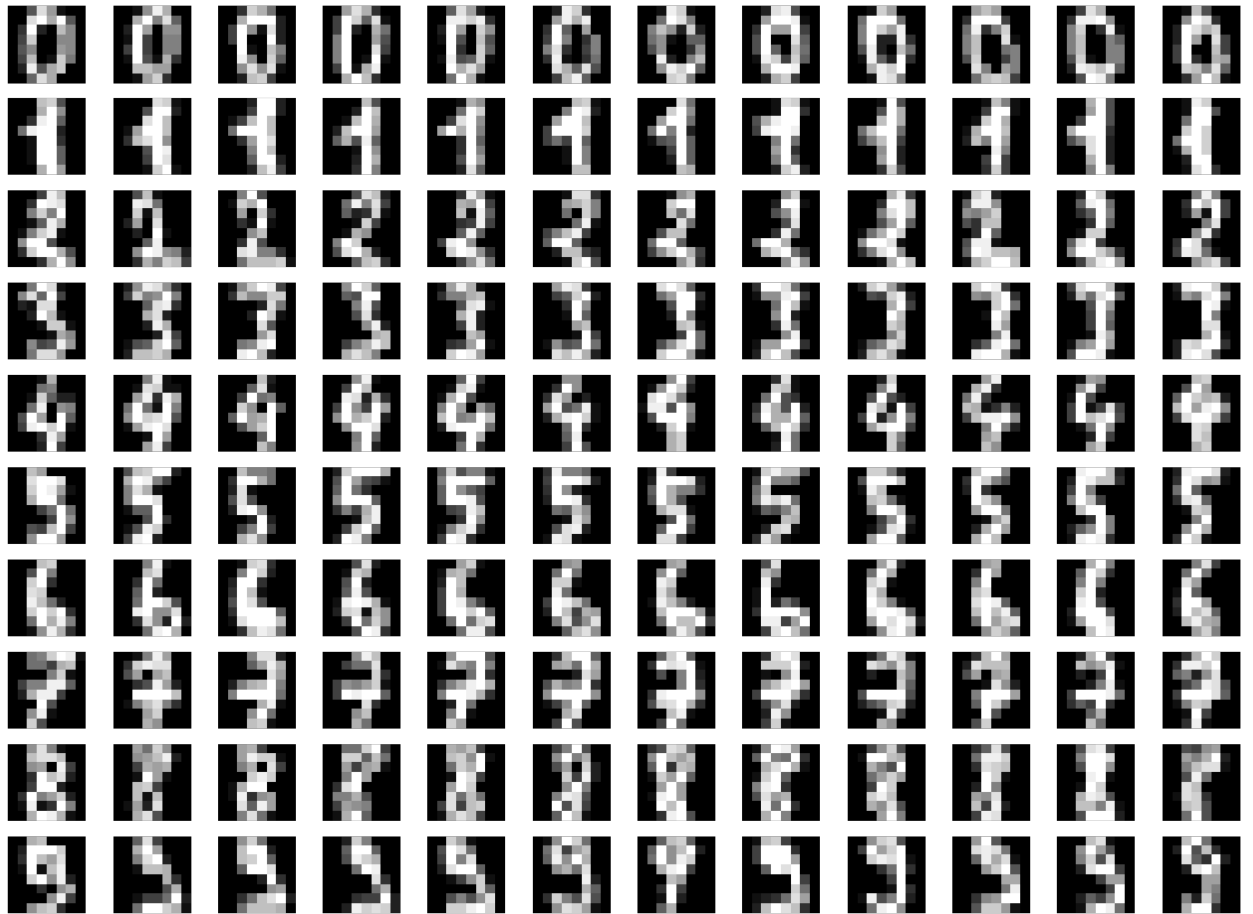
# Les images sont organisées en matrice de 8x8
X = digits.images
# Labels des images (chiffre de 0 à 9)
Y = digits.target
# Les images sont "aplaties" et transformées en un tableau de 64 éléments
Xdata= digits.data

print(f"X Shape: {X.shape}, Xdata Shape: {Xdata.shape}, Y shape: {Y.shape}, "+\
      f"digits.target_name: {digits.target_names}\n")
```

X Shape: (1797, 8, 8), Xdata Shape: (1797, 64), Y shape: (1797,), digits.target_name: [0 1 2 3 4 5 6 7 8 9]

```
Entrée [4]: # Préparation de l'affichage
n_digits = np.unique(Y) # = 0,1,2,...9
M =12
```

```
fig, axs = plt.subplots(len(n_digits), M, figsize=(20, 15))
# Afficher M exemples de tout les digits (de 0 à 9)
for i, d in enumerate(n_digits):
    x= X[Y == d]
    for j in range(M):
        num= random.randint(0,x.shape[0]-1)
        axs[i,j].imshow(X[Y == d][j], cmap="gray")
        axs[i,j].axis('off')
```



```
Entrée [5]: # On split le jeux de données
X_train, X_test, Y_train, Y_test = train_test_split(Xdata, Y, test_size=0.2, random_state= 511)

# On construit et on 'fit' le modèle
log_regr = LogisticRegression(solver='liblinear')
log_regr.fit(X_train, Y_train)

# on teste sur les données de.... test :)
prediction = log_regr.predict(X_test)

err, pre, d= erreur_prediction(valeurs_predites= prediction, valeurs_reelle= Y_test)
print(f"{err} erreurs sur {len(X_test)} images soit une précision de {pre:.2f} %")
print(f"Dictionnaire des erreurs (clé valeur réelle: valeur prédite):\n{d}")
```

19 erreurs sur 360 images soit une précision de 94.72 %
Dictionnaire des erreurs (clé valeur réelle: valeur prédite):
{8: [1, 1, 1, 3, 1], 3: [5, 8], 9: [8, 3, 4, 8], 5: [9, 9, 9, 9], 4: [7, 6], 1: [6, 8]}

```
Entrée [6]: target_names= ["détection de 0", "détection de 1", "détection de 2", "détection de 3", "détection de 4",
                           "détection de 5", "détection de 6", "détection de 7", "détection de 8", "détection de 9"]
print("\n",classification_report(Y_test,prediction, target_names=target_names))
```

	precision	recall	f1-score	support
détection de 0	1.00	1.00	1.00	31
détection de 1	0.89	0.94	0.91	33
détection de 2	1.00	1.00	1.00	35
détection de 3	0.94	0.94	0.94	36
détection de 4	0.97	0.95	0.96	41
détection de 5	0.98	0.91	0.94	45
détection de 6	0.95	1.00	0.97	35
détection de 7	0.97	1.00	0.98	31
détection de 8	0.90	0.88	0.89	40
détection de 9	0.88	0.88	0.88	33
accuracy			0.95	360
macro avg	0.95	0.95	0.95	360
weighted avg	0.95	0.95	0.95	360

Reprise du brief 5 avec MNIST

Importation et visualisation des données du dataset MNIST

On notera (voir ci-dessous) que les jeux de données sont quelque peu déséquilibré.

```
Entrée [17]: # Importation du jeux d'entraînement
f = h5py.File("./datas/train.hdf5", 'r')
train_x, train_y = (f['image'][...])/255.0, f['label'][...]
f.close()

# Importation du jeux de test
f = h5py.File("./datas/test.hdf5", 'r')
test_x, test_y = (f['image'][...])/255.0, f['label'][...]
f.close()

etiquette= np.unique(test_y)
print(f"Entraînement: X Shape: {train_x.shape}, Y shape: {train_y.shape}\n"+\
      f"Test: X Shape: {test_x.shape}, Y shape: {test_y.shape}\n"+\
      f"Etiquette: {etiquette}")
```

Entraînement: X Shape: (60000, 28, 28), Y shape: (60000,)

Test: X Shape: (10000, 28, 28), Y shape: (10000,)

Etiquette: [0 1 2 3 4 5 6 7 8 9]

```
Entrée [18]: # Le jeux de donnée "train" est-il équilibré ?
d= {}
for i in etiquette:
    l= len(train_x[train_y== i])
    d[i]= l
    print(f"Le chiffre {i} est représenté {l} fois", end= " - ")
mini, maxi= min(d, key= d.get), max(d, key= d.get)
print(f"\nLe chiffre le moins représenté est {mini} représenté {d[mini]} fois.")
print(f"Le chiffre le plus représenté est {maxi} représenté {d[maxi]} fois.")
```

Le chiffre 0 est représenté 5923 fois - Le chiffre 1 est représenté 6742 fois - Le chiffre 2 est représenté 5958 fois - Le chiffre 3 est représenté 6131 fois - Le chiffre 4 est représenté 5842 fois - Le chiffre 5 est représenté 5421 fois - Le chiffre 6 est représenté 5918 fois - Le chiffre 7 est représenté 6265 fois - Le chiffre 8 est représenté 5851 fois - Le chiffre 9 est représenté 5949 fois -

Le chiffre le moins représenté est 5 représenté 5421 fois.

Le chiffre le plus représenté est 1 représenté 6742 fois.

```
Entrée [19]: # Le jeux de donnée "test" est-il équilibré ?
d= {}
for i in etiquette:
    l= len(test_x[test_y== i])
    d[i]= l
    print(f"Le chiffre {i} est représenté {l} fois", end= " - ")
mini, maxi= min(d, key= d.get), max(d, key= d.get)
print(f"\nLe chiffre le moins représenté est {mini} représenté {d[mini]} fois.")
print(f"Le chiffre le plus représenté est {maxi} représenté {d[maxi]} fois.")
```

Le chiffre 0 est représenté 980 fois - Le chiffre 1 est représenté 1135 fois - Le chiffre 2 est représenté 1032 fois - Le chiffre 3 est représenté 1010 fois - Le chiffre 4 est représenté 982 fois - Le chiffre 5 est représenté 892 fois - Le chiffre 6 est représenté 958 fois - Le chiffre 7 est représenté 1028 fois - Le chiffre 8 est représenté 974 fois - Le chiffre 9 est représenté 1009 fois -

Le chiffre le moins représenté est 5 représenté 892 fois.

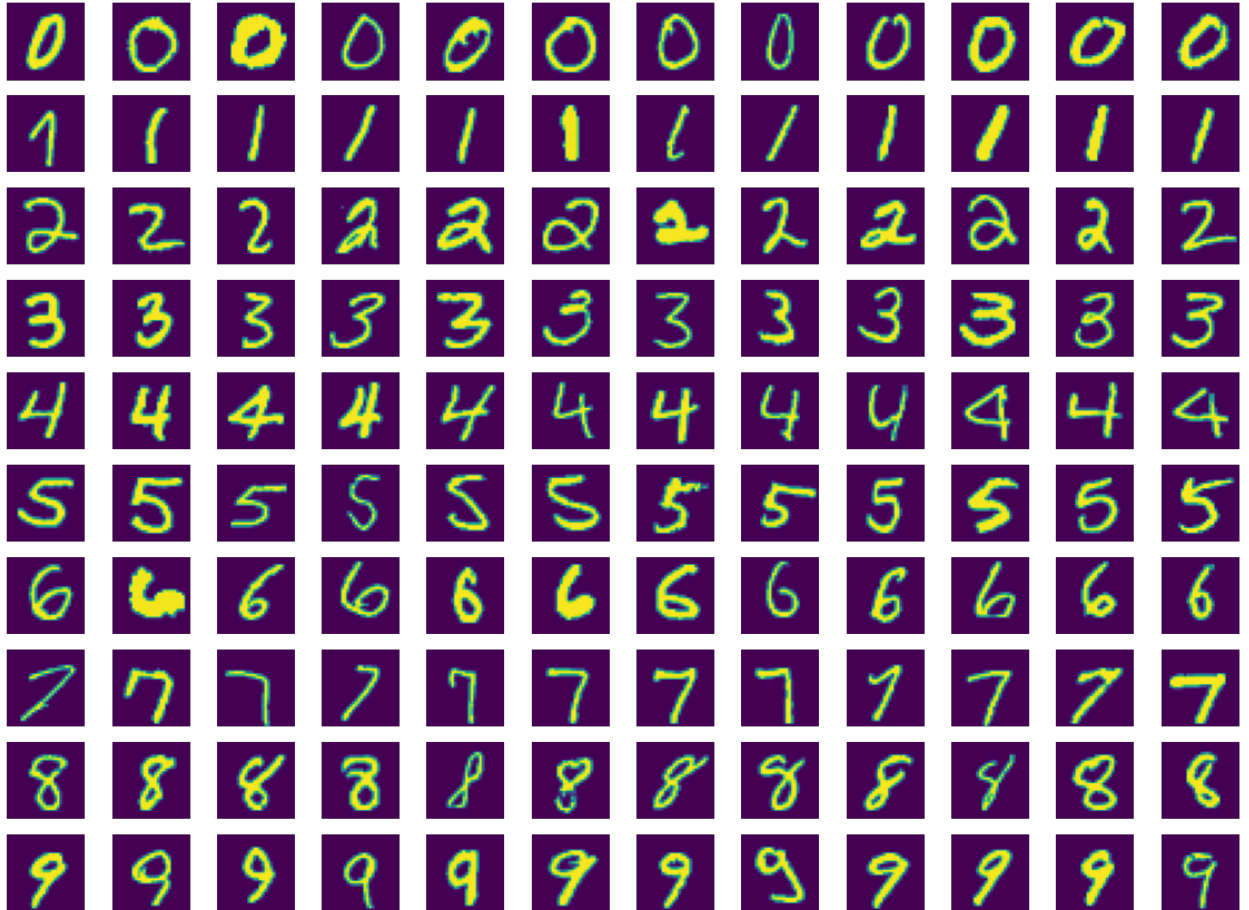
Le chiffre le plus représenté est 1 représenté 1135 fois.

```

Entrée [4]: # Préparation de l'affichage
M= 12
fig, axs = plt.subplots(len(etiquette), M, figsize=(20, 15))

np.random.seed(51165)
# Afficher M exemples de tout les digits (de 0 à 9)
for i, d in enumerate(etiquette):
    x= train_x[train_y == d]
    for j in range(M):
        num= random.randint(0, x.shape[0]-1)
        axs[i,j].imshow(x[num])
        axs[i,j].axis('off')

```



On test en se plaçant dans les mêmes conditions que pour le brief 5

```

Entrée [9]: # On récupère les données. On ne prendra qu'une partie des données situé dans le
# jeux de train. On en profite pour "aplatir" les images
data= train_x.reshape(train_x.shape[0], train_x.shape[1]*train_x.shape[2])

# On réduit le jeux de données original
data_red, _, Y_red, _ = train_test_split(data, train_y, train_size=0.1, random_state= 65)

# On split le jeux de données
X_train_red, X_test_red, Y_train_red, Y_test_red = train_test_split(data_red, Y_red, train_size=0.8,
                                                                    random_state= 65)

print(f"Entrainement: X Shape: {X_train_red.shape}, Y shape: {Y_train_red.shape}\n"+
      f"Test: X Shape: {X_test_red.shape}, Y shape: {Y_test_red.shape}\n"+
      f"Etiquette: {etiquette}")

```

```

Entrainement: X Shape: (4800, 784), Y shape: (4800,)
Test: X Shape: (1200, 784), Y shape: (1200,)
Etiquette: [0 1 2 3 4 5 6 7 8 9]

```

```

Entrée [10]: # Le jeux de donnée est-il équilibré ?
d= {}
for i in etiquette:
    l= len(X_train_red[Y_train_red== i])
    d[i]= l
    print(f"Le chiffre {i} est représenté {l} fois", end= " - ")
mini, maxi= min(d, key= d.get), max(d, key= d.get)
print(f"\nLe chiffre le moins représenté est {mini} représenté {d[mini]} fois.")
print(f"Le chiffre le plus représenté est {maxi} représenté {d[maxi]} fois.")

```

Le chiffre 0 est représenté 483 fois - Le chiffre 1 est représenté 546 fois - Le chiffre 2 est représenté 472 fois - Le chiffre 3 est représenté 483 fois - Le chiffre 4 est représenté 473 fois - Le chiffre 5 est représenté 431 fois - Le chiffre 6 est représenté 482 fois - Le chiffre 7 est représenté 480 fois - Le chiffre 8 est représenté 487 fois - Le chiffre 9 est représenté 463 fois -
 Le chiffre le moins représenté est 5 représenté 431 fois.
 Le chiffre le plus représenté est 1 représenté 546 fois.

```
Entrée [11]: deb= time.time()
# On construit et on 'fit' le modèle
model_reglog = LogisticRegression(solver='liblinear')
model_reglog.fit(X_train_red, Y_train_red)
fin= time.time()
print(f"temps d'exécution: {fin-deb:.2f} s")
```

temps d'exécution: 46.99 s

/home/jpphi/anaconda3/lib/python3.8/site-packages/sklearn/svm/_base.py:1206: ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
warnings.warn()

On teste l'algorithme

```
Entrée [12]: # on teste sur les données de... test :)
prediction = model_reglog.predict(X_test_red)

err, pre, d= erreur_prediction(valeurs_predites= prediction, valeurs_reelle= Y_test_red)
print(f"{err} erreurs sur {len(X_test_red)} images soit une précision de {pre:.2f} %")
print(f"Dictionnaire des erreurs (clé valeur réelle: valeur prédite):\n{d}")
```

225 erreurs sur 1200 images soit une précision de 81.25 %
Dictionnaire des erreurs (clé valeur réelle: valeur prédite):
{8: [3, 3, 5, 5, 5, 4, 5, 2, 3, 0, 3, 9, 1, 2, 9, 1, 2, 4, 9, 3, 3, 2, 9], 2: [3, 7, 9, 0, 7, 6, 8, 0, 9, 3, 7, 1, 3, 6, 1, 1, 5, 0, 0, 8, 5, 5, 8, 5, 8, 3, 6, 3, 8, 8, 7, 8, 3, 9, 8, 8], 5: [3, 3, 3, 9, 8, 8, 9, 3, 9, 6, 8, 4, 3, 9, 3, 3, 3, 3, 4, 3, 8, 2, 6, 6, 3, 9, 3, 3, 8, 8, 9, 9, 8, 3, 8, 6, 8], 9: [8, 8, 7, 4, 1, 8, 4, 4, 8, 4, 4, 8, 8, 2, 3, 8, 7, 8, 8, 4, 2, 7, 8, 8, 7], 3: [7, 5, 9, 8, 8, 8, 8, 2, 8, 8, 6, 2, 0, 9, 8, 0, 8, 8, 8, 5, 8, 7, 8, 8, 8, 5, 5, 8, 8, 5, 8, 8, 8, 8, 4, 8, 8, 8], 4: [9, 8, 7, 8, 2, 9, 0, 8, 3, 8, 2, 9, 2, 7, 8, 8, 9, 3, 9, 6, 9, 9, 7, 5, 0], 7: [1, 4, 4, 9, 3, 8, 3, 3, 1, 3, 9, 2, 5, 9, 9, 9, 8, 9, 2, 5], 6: [8, 0, 5, 3, 8, 5, 8], 0: [6, 5, 4, 2, 6, 9, 8], 1: [3, 8, 3, 9, 3, 2]}

```
Entrée [13]: target_names= ["détection de 0", "détection de 1", "détection de 2", "détection de 3", "détection de 4",
                           "détection de 5", "détection de 6", "détection de 7", "détection de 8", "détection de 9"]
print("Le score obtenu est de {} % de bonne prédictions".
      format(round(model_reglog.score(X_test_red, Y_test_red)*100,2)))
print("\n",classification_report(Y_test_red,prediction, target_names= target_names))
```

Le score obtenu est de 81.25 % de bonne prédictions

	precision	recall	f1-score	support
détection de 0	0.92	0.94	0.93	126
détection de 1	0.93	0.95	0.94	117
détection de 2	0.86	0.74	0.80	137
détection de 3	0.67	0.66	0.66	112
détection de 4	0.88	0.80	0.83	123
détection de 5	0.78	0.64	0.70	103
détection de 6	0.92	0.95	0.93	135
détection de 7	0.88	0.82	0.85	119
détection de 8	0.57	0.79	0.66	110
détection de 9	0.74	0.79	0.77	118
accuracy			0.81	1200
macro avg	0.82	0.81	0.81	1200
weighted avg	0.82	0.81	0.81	1200

Normalisation des données et test d'un nouveau solver

```
Entrée [14]: # On récupère les données. On ne prendra qu'une partie des données situé dans le
# jeux de train. On en profite pour "applatir" les images
data= train_x.reshape(train_x.shape[0], train_x.shape[1]*train_x.shape[2])

# On réduit le jeux de données original
data_red, _, Y_red, _= train_test_split(data, train_y, train_size=0.1, random_state= 65)

# Standardisation
scaler=StandardScaler()
datastd= scaler.fit_transform(data_red)

# On split le jeux de données
X_train_red, X_test_red, Y_train_red, Y_test_red = train_test_split(datastd, Y_red, train_size=0.8,
                                                                    random_state= 65)

print(f"Entrainement: X Shape: {X_train_red.shape}, Y shape: {Y_train_red.shape}\n"+\
      f"Test: X Shape: {X_test_red.shape}, Y shape: {Y_test_red.shape}\n"+\
      f"Etiquette: {etiquette}")
```

Entrainement: X Shape: (4800, 784), Y shape: (4800,)
Test: X Shape: (1200, 784), Y shape: (1200,)
Etiquette: [0 1 2 3 4 5 6 7 8 9]

```
Entrée [15]: deb= time.time()
# On construit et on 'fit' le modèle
model_reglog = LogisticRegression(solver='lbfgs', class_weight= "balanced", max_iter= 2000)
model_reglog.fit(X_train_red, Y_train_red)
fin= time.time()
print(f"temps d'exécution: {fin-deb:.2f} s")
```

temps d'exécution: 14.49 s

```
Entrée [16]: # on teste sur les données de... test :)
prediction = model_reglog.predict(X_test_red)
```

```
err, pre, d= erreur_prediction(valeurs_predites= prediction, valeurs_reelle= Y_test_red)
print(f"erreurs sur {len(X_test_red)} images soit une précision de {pre:.2f} %")
print(f"Dictionnaire des erreurs (clé valeur réelle: valeur prédite):\n{d}")
```

126 erreurs sur 1200 images soit une précision de 89.50 %
Dictionnaire des erreurs (clé valeur réelle: valeur prédite):
{8: [3, 5, 5, 3, 6, 2, 5, 2, 1, 3, 1, 3, 2, 1, 9], 2: [3, 8, 3, 7, 6, 1, 3, 4, 4, 3, 1, 0, 3, 1, 0, 7, 6, 7, 4, 7, 3, 6, 0, 9, 8, 3], 5: [3, 3, 8, 0, 3, 9, 4, 3, 1, 0, 9, 3, 6, 8, 4, 3, 9, 8, 9, 8, 0, 8, 8], 3: [5, 9, 8, 2, 8, 2, 0, 0, 5, 5, 7, 2, 8, 8], 9: [4, 1, 3, 4, 3, 4, 4, 2, 4, 3, 8, 8, 4, 7], 6: [8, 5, 0, 8, 5], 4: [3, 8, 2, 9, 8, 1, 9, 2, 8, 0], 1: [2, 3, 2], 0: [6, 5, 4, 6, 6, 7], 7: [4, 3, 9, 4, 9, 9, 1, 5, 9, 9]}

```
Entrée [17]: target_names= ["détection de 0", "détection de 1", "détection de 2", "détection de 3", "détection de 4",
                           "détection de 5", "détection de 6", "détection de 7", "détection de 8", "détection de 9"]
print(f"Le score obtenu est de {model_reglog.score(X_test_red, Y_test_red)*100:.2f}")
print("\n",classification_report(Y_test_red,prediction, target_names= target_names))
```

Le score obtenu est de 89.50

	precision	recall	f1-score	support
détection de 0	0.92	0.95	0.94	126
détection de 1	0.92	0.97	0.95	117
détection de 2	0.91	0.81	0.86	137
détection de 3	0.81	0.88	0.84	112
détection de 4	0.89	0.92	0.90	123
détection de 5	0.89	0.78	0.83	103
détection de 6	0.94	0.96	0.95	135
détection de 7	0.94	0.92	0.93	119
détection de 8	0.83	0.86	0.85	110
détection de 9	0.88	0.88	0.88	118
accuracy			0.90	1200
macro avg	0.89	0.89	0.89	1200
weighted avg	0.90	0.90	0.89	1200

Gridsearch CV pour déterminer la meilleure standardisation et le meilleur solveur pour la régression logistique

```
Entrée [18]: # On récupère les données. On ne prendra qu'une partie des données situé dans le
# jeux de train. On en profite pour "applatir" les images
data= train_x.reshape(train_x.shape[0], train_x.shape[1]*train_x.shape[2])

# On réduit le jeux de données original
data_red, _, Y_red, _= train_test_split(data, train_y, train_size=0.1, random_state= 65)

# On split le jeux de données
X_train_red, X_test_red, Y_train_red, Y_test_red = train_test_split(data_red, Y_red, train_size=0.8,
                                                                    random_state= 65)

print(f"Entraînement: X Shape: {X_train_red.shape}, Y shape: {Y_train_red.shape}\n"+\
      f"Test: X Shape: {X_test_red.shape}, Y shape: {Y_test_red.shape}\n"+\
      f"Etiquette: {etiquette}")
```

Entraînement: X Shape: (4800, 784), Y shape: (4800,)
Test: X Shape: (1200, 784), Y shape: (1200,)
Etiquette: [0 1 2 3 4 5 6 7 8 9]

```
Entrée [19]: # On réalise un pipe avec 2 actions: un scaler et un algorithme
pipe = Pipeline(steps= [('scaler', StandardScaler()), ('algo', LogisticRegression())])

# On testera 3 standardisations et pour la régression logistique on testera tout les solvers
param_grid = {'pipeline__scaler': [StandardScaler(), MinMaxScaler(), RobustScaler()],
              'pipeline__algo__solver': ["sag", "newton-cg", "lbfgs", "saga"],
              'pipeline__algo__class_weight': ["balanced"],
              'pipeline__algo__max_iter': [2000]}

# instantiate and run as before:

model= make_pipeline(pipe)

grid = GridSearchCV(model, param_grid, cv=5, n_jobs= -1)

grid
```

```
Out[19]: GridSearchCV(cv=5,
                    estimator=Pipeline(steps=[('pipeline',
                                              Pipeline(steps=[('scaler',
                                                                StandardScaler()),
                                                                ('algo',
                                                                LogisticRegression()))])),
                    n_jobs=-1,
                    param_grid={'pipeline__algo__class_weight': ['balanced'],
                              'pipeline__algo__max_iter': [2000],
                              'pipeline__algo__solver': ['sag', 'newton-cg', 'lbfgs',
                                                         'saga'],
                              'pipeline__scaler': [StandardScaler(), MinMaxScaler(),
                                                  RobustScaler()]})
```

```
Entrée [20]: deb= time.time()

grid.fit(X_train_red, Y_train_red)

fin= time.time()
print(f"temps d'exécution: {fin-deb:.2f} s")
```


111 erreurs sur 1200 images soit une précision de 90.75 %
 Dictionnaire des erreurs (clé valeur réelle: valeur prédite):
 {8: [3, 5, 5, 5, 3, 2, 5, 1, 5, 3, 1, 3, 1, 9], 2: [7, 8, 0, 7, 1, 3, 4, 7, 3, 1, 0, 1, 7, 6, 7, 6, 7, 3, 6, 8, 8, 8, 3],

```
Entrée [32]: target_names= ["détection de 0", "détection de 1", "détection de 2", "détection de 3", "détection de 4",
                             "détection de 5", "détection de 6", "détection de 7", "détection de 8", "détection de 9"]
print("\n",classification_report(Y_test_red,prediction, target_names= target_names))
```

	precision	recall	f1-score	support
détection de 0	0.95	0.98	0.96	126
détection de 1	0.93	0.98	0.96	117
détection de 2	0.95	0.83	0.89	137
détection de 3	0.83	0.88	0.85	112
détection de 4	0.93	0.91	0.92	123
détection de 5	0.86	0.84	0.85	103
détection de 6	0.97	0.96	0.97	135
détection de 7	0.91	0.91	0.91	119
détection de 8	0.83	0.87	0.85	110
détection de 9	0.88	0.90	0.89	118
accuracy			0.91	1200
macro avg	0.91	0.91	0.91	1200
weighted avg	0.91	0.91	0.91	1200

Entrainement de l'algorithme de la regression logistique avec la totalité du jeux MNIST normalisé avec min/max.

```
Entrée [3]: # Importation du jeux d'entraînement. On réalise directement une normalisation min/max en divisant par 255.0
# Les pixels composant l'image étant compris entre 0 et 255
f = h5py.File("./datas/train.hdf5", 'r')
train_x, train_y = (f['image'][...])/255.0, f['label'][...]
f.close()

# Importation du jeux de test
f = h5py.File("./datas/test.hdf5", 'r')
test_x, test_y = (f['image'][...])/255.0, f['label'][...]
f.close()
```

```
etiquette= np.unique(test_y)
print(f"Entrainement: X Shape: {train_x.shape}, Y shape: {train_y.shape}\n"+
      f"Test: X Shape: {test_x.shape}, Y shape: {test_y.shape}\n"+
      f"Etiquette: {etiquette}")
```

```
Entrainement: X Shape: (60000, 28, 28), Y shape: (60000,)
Test: X Shape: (10000, 28, 28), Y shape: (10000,)
Etiquette: [0 1 2 3 4 5 6 7 8 9]
```

```
Entrée [4]: X_train= train_x.reshape(train_x.shape[0], train_x.shape[1]*train_x.shape[2])
Y_train= train_y

X_test= test_x.reshape(test_x.shape[0], test_x.shape[1]*test_x.shape[2])
Y_test= test_y
```

```
Entrée [7]: model_reglog= LogisticRegression(class_weight='balanced', max_iter=2000, solver='newton-cg')
deb= time.time()
model_reglog.fit(X_train, Y_train)
fin= time.time()
print(f"fit en {fin-deb} s")

fit en 328.4471061229706 s
```

```
Entrée [8]: prediction = model_reglog.predict(X_test)

err, pre, d= erreur_prediction(valeurs_predites= prediction, valeurs_reelle= Y_test)
print(f"{err} erreurs sur {len(X_test)} images soit une précision de {pre:.2f} %")
print(f"Dictionnaire des erreurs (clé valeur réelle: valeur prédite):\n{d}")
```

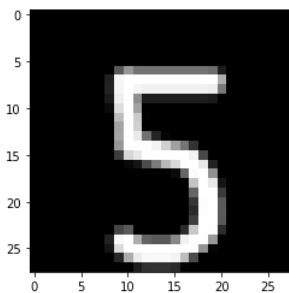
737 erreurs sur 10000 images soit une précision de 92.63 %
 Dictionnaire des erreurs (clé valeur réelle: valeur prédite):
 {5: [6, 7, 3, 0, 8, 3, 8, 8, 3, 7, 8, 9, 3, 4, 3, 8, 3, 6, 7, 6, 3, 0, 8, 0, 3, 3, 4, 9, 8, 8, 8, 4, 8, 3, 2, 8, 7, 3, 8, 0, 3, 3, 2, 0, 7, 7, 9, 9, 4, 0, 8, 4, 8, 0, 2, 3, 3, 3, 1, 6, 8, 8, 4, 3, 8, 6, 3, 8, 6, 8, 8, 1, 3, 3, 3, 3, 3, 8, 3, 8, 8, 3, 6, 4, 8, 6, 8, 3, 6, 8, 8, 4, 4, 8, 8, 3, 6, 3, 3, 3, 6, 6, 0, 0, 6, 6, 3, 6], 4: [6, 6, 2, 9, 9, 9, 9, 8, 9, 1, 6, 9, 6, 9, 9, 9, 3, 7, 3, 6, 8, 9, 9, 9, 9, 9, 7, 9, 9, 9, 9, 2, 9, 2, 2, 9, 1, 6, 9, 2, 6, 9, 9, 1, 6, 9, 3, 7, 0, 9, 8, 9, 8, 8, 9, 9, 9, 7, 2, 9, 8], 3: [2, 5, 5, 5, 7, 5, 2, 6, 5, 2, 5, 5, 8, 5, 9, 8, 7, 9, 8, 7, 7, 5, 7, 5, 1, 8, 5, 9, 5, 5, 6, 5, 0, 2, 7, 5, 5, 2, 2, 2, 8, 5, 2, 2, 7, 7, 7, 5, 9, 8, 5, 2, 8, 5, 8, 4, 0, 9, 0, 8, 8, 5, 8, 8, 8, 8, 7, 5, 8, 2, 2, 2, 2, 8, 8, 5, 8, 8, 5, 8, 2, 0, 9, 2, 8, 9, 8], 6: [2, 5, 0, 0, 0, 5, 2, 2, 1, 8, 7, 7, 5, 4, 0, 2, 1, 5, 4, 5, 1, 2, 4, 0, 3, 0, 4, 5, 0, 5, 5, 5, 5, 2, 2, 2, 5, 4, 4, 5, 5, 5, 3, 0], 7: [4, 9, 2, 1, 3, 2, 2, 9, 9, 9, 2, 2, 1, 2, 2, 9, 1, 9, 9, 3, 2, 9, 9, 8, 2, 9, 5, 9, 3, 9, 4, 4, 9, 9, 4, 1, 2, 9, 9, 8, 0, 9, 2, 2, 3, 9, 1, 4, 9, 9, 3, 9, 9, 9, 2, 2, 8, 8, 2, 9, 9, 9, 9, 9, 2, 9, 1, 3, 4, 2, 2, 2, 2, 2, 2, 2, 2, 9, 9, 3, 9], 2: [9, 7, 7, 8, 1, 8, 8, 6, 9, 8, 8, 7, 0, 3, 3, 6, 6, 1, 4, 9, 0, 7, 0, 8, 8, 3, 8, 1, 0, 4, 8, 8, 3, 3, 1, 3, 5, 8, 3, 8, 3, 8, 8, 1, 8, 4, 4, 7, 3, 8, 5, 4, 7, 1, 4, 3, 3, 3, 8, 8, 3, 8, 8, 8, 6, 6, 6, 8, 4, 8, 6, 6, 7, 8, 6, 9, 4, 4, 6, 8, 8, 8, 7, 8, 8, 3, 8, 4, 6, 1, 8, 5, 0, 0, 8, 8, 7, 8, 8], 9: [3, 8, 7, 8, 3, 7, 7, 3, 4, 4, 5, 8, 7, 7, 5, 4, 5, 2, 1, 4, 0, 0, 1, 1, 0, 4, 4, 4, 8, 7, 8, 1, 1, 3, 4, 4, 4, 3, 1, 4, 7, 4, 0, 7, 4, 4, 7, 1, 8, 7, 4, 5, 7, 7, 3, 8, 8, 0, 3, 3, 0, 0, 0, 3, 4, 7, 5, 0, 8, 5, 4, 7, 5, 7, 7, 4, 7, 8, 4, 4, 4], 8: [7, 5, 5, 7, 2, 3, 1, 3, 4, 5, 3, 7, 2, 9, 4, 4, 5, 5, 3, 1, 6, 2, 1, 7, 5, 0, 5, 5, 9, 9, 3, 3, 5, 5, 9, 6, 2, 0, 0, 9, 5, 3, 9, 3, 2, 5, 0, 1, 3, 5, 5, 3, 3, 3, 5, 5, 5, 2, 3, 5, 0, 7, 3, 4, 3, 7, 3, 7, 6, 0, 3, 6, 6, 6, 4, 1, 5, 4, 0, 1, 1, 6, 6, 9, 1, 7, 4, 5, 5, 7, 9, 9, 7, 3, 5, 5, 9, 9, 5, 1, 0, 5, 5, 5, 6, 6, 6, 0, 6, 5, 9, 6, 5, 1, 6, 1], 1: [8, 2, 6, 5, 2, 6, 2, 3, 7, 3, 6, 5, 2, 2, 8, 2, 8, 8, 8, 8, 8, 8, 8, 7], 0: [6, 7, 5, 5, 5, 5, 5, 7, 5, 4, 2, 6, 8, 3, 6, 6, 3, 7, 5, 5, 3, 5, 3, 5, 2]}

```
Entrée [9]: target_names= ["détection de 0", "détection de 1", "détection de 2", "détection de 3", "détection de 4",
                             "détection de 5", "détection de 6", "détection de 7", "détection de 8", "détection de 9"]
print("\n",classification_report(Y_test,prediction, target_names= target_names))
```


	precision	recall	f1-score	support
détection de 0	0.95	0.97	0.96	980
détection de 1	0.97	0.98	0.97	1135
détection de 2	0.93	0.90	0.92	1032
détection de 3	0.91	0.91	0.91	1010
détection de 4	0.94	0.94	0.94	982
détection de 5	0.89	0.88	0.89	892
détection de 6	0.94	0.95	0.95	958
détection de 7	0.94	0.92	0.93	1028
détection de 8	0.88	0.88	0.88	974
détection de 9	0.91	0.92	0.92	1009
accuracy			0.93	10000
macro avg	0.93	0.93	0.93	10000
weighted avg	0.93	0.93	0.93	10000

Test sur un chiffre qui n'est pas issue d'un dataset

```
Entrée [4]: img= Image.open("./images/image_test.png")
            _plt.imshow(img, cmap= "gray")
```



```
Entrée [12]: chiffre=np.array(img)/255.
            chiffre_plat= chiffre.ravel()
            tab= list(model_reglog.predict_proba([chiffre_plat])[0])
            print(f"Le chiffre le plus probable est: {tab.index(max(tab))}\nListe des probabilités:{tab}")
```

Le chiffre le plus probable est: 5
 Liste des probabilités:[0.00010985476550631836, 6.972696327770812e-07, 0.008636546996151222, 0.369270499667964, 0.0004235051156176312, 0.3821188287201187, 0.00019094647377095294, 1.4248341358281702e-10, 0.23924762938532074, 1.4914634344104556e-06]

Enregistrement du modèle

```
Entrée [21]: dump(model_reglog, 'RegressionLogistique_newton_balanced.modele')
```

```
Out[21]: ['RegressionLogistique_newton_balanced.modele']
```

Learning curve

- Le but de cette étude est de montrer que l'on pouvait ne pas prendre la totalité des données pour réaliser l'optimisation des hyperparamètres (qui aurait demandé sinon un temps de calcul très élevé). Cependant, le modèle définitif trouvé a été entraîné sur la totalité du jeu de données.
- On fera cette étude en prenant l'intégralité du jeu de données avec le meilleur solveur.

```
Entrée [33]: # On récupère les données et on en profite pour "aplatir" les images
            data_train= train_x.reshape(train_x.shape[0], train_x.shape[1]*train_x.shape[2])
            data_test= test_x.reshape(test_x.shape[0], test_x.shape[1]*test_x.shape[2])

            etiquette= np.unique(test_y)

            print(f"Entraînement: X Shape: {data_train.shape}, Y shape: {train_y.shape}\n"+\
                  f"Test: X Shape: {data_test.shape}, Y shape: {test_y.shape}\n"+\
                  f"Etiquette: {etiquette}")
```

Entraînement: X Shape: (60000, 784), Y shape: (60000,)
 Test: X Shape: (10000, 784), Y shape: (10000,)
 Etiquette: [0 1 2 3 4 5 6 7 8 9]

```
Entrée [34]: # On choisi le meilleur solveur avec une standardisation min/max
            pipe_minmax_logreg= make_pipeline(MinMaxScaler(), LogisticRegression(class_weight='balanced',
                                                                                   max_iter=2000, solver='newton-cg'))

            pipe_minmax_logreg.fit(data_train, train_y)
```

```
Out[34]: Pipeline(steps=[('minmaxscaler', MinMaxScaler()),
                          ('logisticregression',
                           LogisticRegression(class_weight='balanced', max_iter=2000,
                                                solver='newton-cg'))])
```

```
Entrée [35]: prediction = pipe_minmax_logreg.predict(data_test)

            err, pre, d= erreur_prediction(valeurs_predites= prediction, valeurs_reelle= test_y)
            print(f"{err} erreurs sur {len(data_test)} images soit une précision de {pre:.2f} %")
            print(f"Dictionnaire des erreurs (clé valeur réelle: valeur prédite):\n{d}")
```

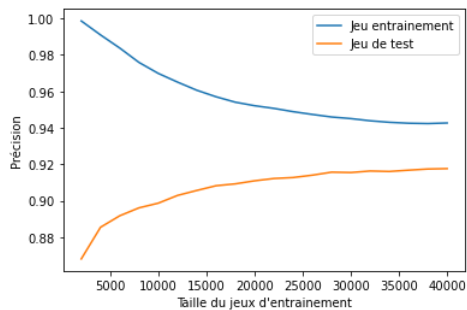
737 erreurs sur 10000 images soit une précision de 92.63 %

Dictionnaire des erreurs (clé valeur réelle: valeur prédite):

```
{5: [6, 7, 3, 0, 8, 3, 8, 8, 3, 7, 8, 9, 3, 4, 3, 8, 3, 6, 7, 6, 3, 0, 8, 0, 3, 3, 4, 9, 8, 8, 8, 4, 8, 3, 2, 8, 7, 3, 8, 0, 3, 3, 2, 0, 7, 7, 9, 9, 4, 0, 8, 4, 8, 0, 2, 3, 3, 3, 1, 6, 8, 8, 4, 3, 8, 6, 3, 8, 6, 8, 8, 1, 3, 3, 3, 3, 3, 3, 8, 3, 8, 8, 3, 6, 4, 8, 6, 8, 3, 6, 8, 8, 4, 4, 8, 8, 3, 6, 3, 3, 3, 6, 6, 0, 0, 6, 6, 3, 6], 4: [6, 6, 2, 9, 9, 9, 9, 8, 9, 1, 6, 9, 6, 9, 9, 9, 3, 7, 3, 6, 8, 9, 9, 9, 9, 9, 7, 9, 9, 9, 9, 2, 9, 9, 2, 2, 9, 1, 9, 2, 6, 9, 9, 1, 6, 9, 3, 7, 0, 9, 8, 9, 8, 8, 9, 9, 9, 7, 2, 9, 8], 3: [2, 5, 5, 5, 7, 5, 2, 6, 5, 2, 5, 5, 8, 5, 9, 8, 7, 9, 8, 7, 7, 5, 7, 5, 1, 8, 5, 9, 5, 5, 6, 5, 0, 2, 7, 5, 5, 2, 2, 2, 8, 5, 2, 2, 7, 7, 7, 5, 9, 8, 5, 2, 8, 5, 8, 4, 2, 0, 9, 0, 8, 8, 5, 8, 8, 8, 8, 7, 5, 8, 2, 2, 2, 2, 8, 8, 5, 8, 8, 5, 8, 2, 0, 9, 2, 8, 9, 8], 6: [2, 5, 0, 0, 0, 5, 2, 2, 1, 8, 7, 7, 5, 4, 0, 2, 1, 5, 4, 5, 1, 2, 4, 0, 3, 0, 4, 5, 0, 5, 5, 5, 5, 2, 2, 2, 5, 4, 4, 5, 5, 5, 5, 3, 0], 7: [4, 9, 2, 1, 3, 2, 2, 9, 9, 9, 2, 2, 1, 2, 2, 9, 1, 9, 9, 3, 2, 9, 9, 8, 0, 9, 2, 2, 3, 9, 1, 4, 9, 9, 3, 9, 9, 9, 2, 2, 8, 8, 2, 9, 9, 9, 9, 9, 2, 9, 1, 3, 4, 2, 2, 2, 2, 2, 2, 2, 9, 9, 3, 9], 2: [9, 7, 7, 8, 1, 8, 8, 6, 9, 8, 8, 7, 0, 3, 3, 6, 6, 6, 1, 4, 9, 0, 7, 0, 8, 8, 3, 8, 1, 0, 4, 8, 8, 3, 3, 1, 3, 5, 8, 3, 8, 3, 8, 8, 1, 8, 4, 4, 7, 3, 8, 5, 4, 7, 1, 4, 3, 3, 3, 8, 8, 3, 8, 8, 6, 6, 6, 8, 4, 8, 6, 6, 7, 8, 6, 9, 4, 4, 6, 8, 8, 8, 7, 8, 8, 3, 8, 4, 6, 1, 8, 5, 0, 0, 8, 8, 7, 8, 8], 9: [3, 8, 7, 8, 3, 7, 7, 3, 4, 4, 5, 8, 7, 7, 5, 4, 5, 2, 1, 4, 0, 0, 1, 1, 0, 4, 4, 4, 8, 7, 8, 1, 1, 3, 4, 4, 4, 3, 1, 4, 7, 4, 0, 7, 4, 4, 7, 1, 8, 7, 4, 5, 7, 7, 7, 3, 8, 8, 0, 3, 3, 0, 0, 0, 3, 4, 7, 5, 0, 8, 5, 4, 7, 5, 7, 4, 4, 7, 7, 8, 4, 4, 4], 8: [7, 5, 5, 7, 2, 3, 1, 3, 4, 5, 3, 7, 2, 9, 4, 4, 5, 5, 3, 1, 6, 2, 1, 7, 5, 0, 5, 5, 9, 9, 3, 3, 5, 5, 9, 6, 2, 0, 0, 9, 5, 3, 9, 3, 2, 5, 0, 1, 3, 5, 5, 3, 3, 3, 3, 5, 5, 5, 2, 3, 5, 0, 7, 3, 4, 3, 7, 3, 7, 6, 0, 6, 6, 6, 4, 1, 5, 4, 0, 1, 1, 6, 6, 9, 1, 7, 4, 5, 5, 7, 9, 9, 7, 3, 5, 5, 9, 9, 5, 1, 0, 5, 5, 5, 6, 6, 6, 0, 6, 5, 9, 6, 5, 1, 6, 1], 1: [8, 2, 6, 5, 2, 6, 2, 3, 7, 3, 6, 5, 2, 2, 8, 2, 8, 8, 8, 8, 8, 8, 8, 8, 8, 7], 0: [6, 7, 5, 5, 5, 5, 5, 7, 5, 4, 2, 6, 8, 3, 6, 6, 3, 7, 5, 5, 3, 5, 3, 5, 2]}
```

```
Entrée [36]: train_size, train_score, val_score= learning_curve(pipe_minmax_logreg, data_train, train_y,
train_sizes= np.linspace(0.05,1,20), cv= 3)
```

```
Entrée [37]: plt.plot(train_size, train_score.mean(axis=1), label= "Jeu entraînement")
plt.plot(train_size, val_score.mean(axis=1), label= "Jeu de test")
plt.xlabel("Taille du jeux d'entraînement")
plt.ylabel("Précision")
_= plt.legend()
```



Recherche de nouveau algo - KNN classifieur

```
Entrée [39]: # On récupère les données et on en profite pour "aplatir" les images
data_train= train_x.reshape(train_x.shape[0], train_x.shape[1]*train_x.shape[2])
data_test= test_x.reshape(test_x.shape[0], test_x.shape[1]*test_x.shape[2])

etiquette= np.unique(test_y)

print(f"Entraînement: X Shape: {data_train.shape}, Y shape: {train_y.shape}\n"+
      f"Test: X Shape: {data_test.shape}, Y shape: {test_y.shape}\n"+
      f"Etiquette: {etiquette}")
```

```
Entraînement: X Shape: (60000, 784), Y shape: (60000,)
Test: X Shape: (10000, 784), Y shape: (10000,)
Etiquette: [0 1 2 3 4 5 6 7 8 9]
```

```
Entrée [40]: # On choisi les meilleurs paramètres en gardant la standardisation min/max
pipe_minmax_knn= make_pipeline(MinMaxScaler(), KNeighborsClassifier(n_neighbors= 6, weights= "distance",
algorithm='ball_tree', p= 3))

pipe_minmax_knn.fit(data_train, train_y)
prediction = pipe_minmax_knn.predict(data_test)
```

```
Out[40]: Pipeline(steps=[('minmaxscaler', MinMaxScaler()),
('kneighborsclassifier',
KNeighborsClassifier(algorithm='ball_tree', n_neighbors=6, p=3,
weights='distance'))])
```

```
Entrée [42]: err, pre, d= erreur_prediction(valeurs_predites= prediction, valeurs_reelle= test_y)
print(f"{err} erreurs sur {len(data_test)} images soit une précision de {pre:.2f} %")
print(f"Dictionnaire des erreurs (clé valeur réelle: valeur prédite):\n{d}")
```

263 erreurs sur 10000 images soit une précision de 97.37 %

Dictionnaire des erreurs (clé valeur réelle: valeur prédite):

```
{4: [0, 9, 6, 1, 9, 9, 9, 6, 9, 1, 9, 8, 9, 9, 9, 6, 6, 9, 9, 9, 1, 7, 1, 9, 7, 1, 1, 9, 9], 3: [5, 7, 7, 5, 5, 9, 7, 4, 5, 2, 5, 5, 5, 7, 7, 5, 7, 5, 5, 8, 2, 7, 6, 8, 9, 5, 9, 5, 9, 5, 5, 9, 8], 9: [8, 7, 8, 4, 0, 3, 5, 3, 4, 2, 6, 1, 4, 7, 0, 7, 1, 7, 4, 1, 1, 3, 3, 2, 7, 7, 2, 4, 1, 4, 5, 3, 5, 0, 0, 0, 7, 7, 4], 2: [7, 8, 7, 6, 7, 7, 0, 7, 0, 7, 0, 7, 1, 0, 7, 6, 1, 1, 3, 7, 7, 3, 7, 7, 7, 7, 0, 7, 0, 0, 8, 7, 8, 0], 6: [4, 0, 5, 0, 1, 4, 5, 1, 0, 4, 0, 1, 0], 7: [4, 1, 4, 2, 1, 1, 1, 9, 9, 1, 1, 1, 1, 9, 1, 1, 1, 1, 9, 9, 9, 9, 1, 1, 2, 2, 2], 8: [0, 1, 3, 3, 3, 9, 4, 3, 3, 7, 4, 5, 3, 2, 4, 2, 3, 3, 6, 6, 3, 5, 0, 0, 5, 2, 7, 9, 3, 7, 6, 0, 6, 2, 4, 9, 9, 7, 9, 4, 5, 5], 5: [9, 4, 3, 6, 9, 7, 3, 6, 8, 3, 9, 9, 0, 8, 4, 6, 3, 6, 9, 6, 6, 0, 6, 0, 6], 1: [2, 2, 6], 0: [6, 2, 7, 5, 6, 1, 6]}
```

312 erreurs sur 10000 images soit une précision de 96.88 %

Dictionnaire des erreurs (clé valeur réelle: valeur prédite):

```
{4: [0, 9, 6, 1, 9, 9, 9, 9, 6, 9, 1, 9, 8, 9, 9, 1, 9, 9, 9, 9, 6, 6, 9, 9, 9, 9, 1, 7, 1, 0, 9, 7, 1, 1, 9, 9, 0], 3: [1, 7, 7, 9, 5, 5, 9, 7, 4, 1, 2, 5, 5, 5, 7, 7, 5, 7, 5, 8, 2, 6, 8, 5, 9, 5, 1, 2, 5, 5, 8, 5, 9, 5], 9: [8, 7, 3, 8, 4, 3, 3, 3, 4, 2, 1, 6, 1, 1, 4, 7, 0, 4, 5, 1, 7, 4, 1, 1, 3, 3, 2, 0, 7, 2, 1, 7, 4, 7, 5, 3, 3, 5, 0, 0, 7, 7, 7, 7, 4], 2: [0, 7, 8, 7, 4, 7, 7, 0, 7, 0, 7, 1, 0, 7, 1, 0, 7, 6, 1, 1, 3, 7, 7, 3, 1, 7, 7, 0, 7, 0, 1, 8, 1, 7, 1, 0, 0, 0, 8, 7, 0], 6: [4, 0, 5, 0, 1, 4, 5, 1, 0, 4, 0, 1, 0], 7: [4, 1, 1, 2, 1, 1, 9, 4, 9, 1, 1, 1, 1, 4, 1, 1, 9, 1, 1, 1, 1, 1, 1, 1, 9, 9, 9, 1, 1, 1, 9, 9, 9, 9, 9, 1, 1, 2, 2, 2], 8: [0, 1, 3, 4, 3, 3, 6, 9, 4, 5, 3, 5, 3, 2, 5, 7, 0, 7, 5, 4, 7, 5, 3, 2, 4, 1, 2, 3, 0, 3, 5, 0, 6, 3, 5, 0, 0, 5, 3, 3, 5, 2, 7, 9, 3, 7, 6, 0, 6, 2, 4, 5, 0, 6, 9, 1, 9, 3, 4, 5, 5], 5: [9, 4, 3, 6, 9, 7, 3, 3, 3, 3, 3, 6, 8, 3, 3, 9, 0, 8, 4, 0, 3, 9, 3, 0, 3, 3, 3, 6, 0, 6, 0], 1: [2, 2], 0: [2, 7, 5, 6, 1, 6]}
```

Plutôt que d'intégrer le min max dans le modèle on demandera à ce que la normalisation soit faite avant

Pour avoir un modèle compatible avec les autre modèle pour le traitement et la comparaison avec les autres algorithme, on "sort" la normalisation du modèle.

```
Entrée [3]: # Importation du jeux d'entraînement. On réalise directement une normalisation min/max en divisant par 255.0
# Les pixels composant l'image étant compris entre 0 et 255
f = h5py.File("./datas/train.hdf5", 'r')
train_x, train_y = (f['image'][...])/255.0, f['label'][...]
f.close()

# Importation du jeux de test
f = h5py.File("./datas/test.hdf5", 'r')
test_x, test_y = (f['image'][...])/255.0, f['label'][...]
f.close()

etiquette= np.unique(test_y)
print(f"Entraînement: X Shape: {train_x.shape}, Y shape: {train_y.shape}\n"+\
      f"Test: X Shape: {test_x.shape}, Y shape: {test_y.shape}\n"+\
      f"Etiquette: {etiquette}")
```

```
Entraînement: X Shape: (60000, 28, 28), Y shape: (60000,)
Test: X Shape: (10000, 28, 28), Y shape: (10000,)
Etiquette: [0 1 2 3 4 5 6 7 8 9]
```

```
Entrée [4]: X_train= train_x.reshape(train_x.shape[0], train_x.shape[1]*train_x.shape[2])
Y_train= train_y

X_test= test_x.reshape(test_x.shape[0], test_x.shape[1]*test_x.shape[2])
Y_test= test_y
```

```
Entrée [5]: model_knn= KNeighborsClassifier(algorithm='ball_tree', n_neighbors=6, p=3, weights='distance')
deb= time.time()
model_knn.fit(X_train, Y_train)
fin= time.time()
print(f"fit en {fin-deb} s")
```

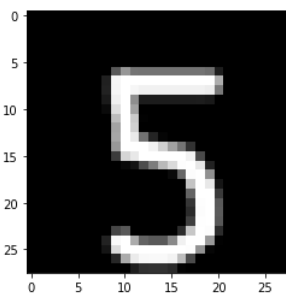
```
fit en 16.260356187820435 s
```

```
Entrée [6]: dump(model_knn, "KNN_n6p3_distance_balltreep3.modele")
```

```
Out[6]: ['KNN_n6p3_distance_balltreep3.modele']
```

Test sur une image non issue de MNIST

```
Entrée [6]: img= Image.open("./images/image_test.png")
_ = plt.imshow(img, cmap= "gray")
```



```
Entrée [14]: #model_knn= load('./modeles/KNN_n6p3_distance_balltreep3.modele')
chiffre=np.array(img)/255.0
chiffre=chiffre.ravel()
tab= list(model_knn.predict_proba([chiffre])[0])
print(tab.index(max(tab)))
print(100 * np.round(tab,4))
```

```
5
[ 0.  0.  0.  0.  0. 100.  0.  0.  0.  0.]
```

Test d'un CNN

On normalisera le jeux de données en divisant chaque élément par 255.0 ce qui les transforme en flottant compris entre 0 et 1.

```
Entrée [4]: # Importation du jeux d'entraînement
f = h5py.File("./datas/train.hdf5", 'r')
train_x, train_y = (f['image'][...])/255.0, f['label'][...]
f.close()

# Importation du jeux de test
f = h5py.File("./datas/test.hdf5", 'r')
test_x, test_y = (f['image'][...])/255.0, f['label'][...]
f.close()
```

```

Entrée [5]: # On récupère les données et on les met en forme pour qu'elles soient compatible avec le CNN.
# la dimension des données de test et de train sont de type:
# (shape[0],shape[1], shape[2]), il sera nécessaire d'y ajouter une dimension supplémentaire
data_train_pour_cnn= train_x.reshape(train_x.shape[0], train_x.shape[1], train_x.shape[2], 1)
data_test_pour_cnn= test_x.reshape(test_x.shape[0], test_x.shape[1], test_x.shape[2], 1)

etiquette= np.unique(test_y)

# convert class vectors to binary class matrices
train_y_conv = keras.utils.to_categorical(train_y, len(etiquette))
test_y_conv = keras.utils.to_categorical(test_y, len(etiquette))

print(f"Entraînement: X Shape: {data_train_pour_cnn.shape}, Y shape: {train_y.shape}\n"+\
      f"Test: X Shape: {data_test_pour_cnn.shape}, Y shape: {test_y.shape}\n"+\
      f"Etiquette: {etiquette}")

```

```

Entraînement: X Shape: (60000, 28, 28, 1), Y shape: (60000,)
Test: X Shape: (10000, 28, 28, 1), Y shape: (10000,)
Etiquette: [0 1 2 3 4 5 6 7 8 9]

```

Conception du réseau

2 versions de se réseau ont été faite, une avec dropout, l'autre sans. Sur les jeux de test et d'entraînement, les courbes sont très proche et les résultats obtenus similaire. A voir sur des cas "réel" si les 2 réseaux se distinguent.

```

Entrée [70]: batch_size = 128
epochs = 250

input_shape= (train_x.shape[1], train_x.shape[2], 1)
nb_classe= len(etiquette)

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),activation='relu',input_shape= input_shape ))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(nb_classe, activation='softmax'))
model.compile(loss=keras.losses.categorical_crossentropy, optimizer=keras.optimizers.Adadelta(),
              metrics=['accuracy'])

model.summary()

```

Model: "sequential_4"

Layer (type)	Output Shape	Param #
conv2d_8 (Conv2D)	(None, 26, 26, 32)	320
conv2d_9 (Conv2D)	(None, 24, 24, 64)	18496
max_pooling2d_4 (MaxPooling 2D)	(None, 12, 12, 64)	0
dropout_8 (Dropout)	(None, 12, 12, 64)	0
flatten_4 (Flatten)	(None, 9216)	0
dense_8 (Dense)	(None, 256)	2359552
dropout_9 (Dropout)	(None, 256)	0
dense_9 (Dense)	(None, 10)	2570
Total params: 2,380,938		
Trainable params: 2,380,938		
Non-trainable params: 0		

```

Entrée [9]: hist = model.fit(data_train_pour_cnn, train_y_conv, batch_size=batch_size, epochs=epochs,
                             verbose=1,validation_data=(data_test_pour_cnn, test_y_conv), workers= -1)

Epoch 1/251
469/469 [=====] - 278s 589ms/step - loss: 5.2512 - accuracy: 0.4805 - val_loss: 1.7019 - val_accu
racy: 0.7154
Epoch 2/251
469/469 [=====] - 253s 540ms/step - loss: 1.2893 - accuracy: 0.7795 - val_loss: 0.9239 - val_accu
racy: 0.8355
Epoch 3/251
469/469 [=====] - 255s 544ms/step - loss: 0.8350 - accuracy: 0.8492 - val_loss: 0.6893 - val_accu
racy: 0.8729
Epoch 4/251
469/469 [=====] - 259s 551ms/step - loss: 0.6502 - accuracy: 0.8787 - val_loss: 0.5698 - val_accu
racy: 0.8959
Epoch 5/251
469/469 [=====] - 259s 551ms/step - loss: 0.5416 - accuracy: 0.8957 - val_loss: 0.4912 - val_accu
racy: 0.9078
Epoch 6/251
469/469 [=====] - 258s 551ms/step - loss: 0.4677 - accuracy: 0.9079 - val_loss: 0.4409 - val_accu
racy: 0.9135
Epoch 7/251
469/469 [=====] - 258s 552ms/step - loss: 0.4137 - accuracy: 0.9160 - val_loss: 0.3886 - val_accu
racy: 0.9200

```

Les noms seront par la suite changer pour pouvoir être manipuler plus simplement en fonction des contraintes du programme de visualition et de test.

Entrée [25]: hist.history

Tracé des courbes d'évolution de la perte et de la précision

The figure consists of two side-by-side line graphs. The left graph, titled 'Courbes de loss', plots 'loss' on the y-axis (ranging from 0.1 to 0.6) against 'Epoch' on the x-axis (ranging from 0 to 250). It shows two curves: 'Jeux d'entraînement' (blue) and 'Jeux de validation' (orange). Both curves decrease over time, with the training loss starting at approximately 0.58 and ending at 0.14, and the validation loss starting at approximately 0.25 and ending at 0.06. The right graph, titled 'Courbes d'accuracy', plots 'précision' on the y-axis (ranging from 0.84 to 0.98) against 'Epoch' on the x-axis (ranging from 0 to 250). It shows two curves: 'Jeux d'entraînement' (blue) and 'Jeux de validation' (orange). Both curves increase over time, with the training accuracy starting at approximately 0.83 and ending at 0.96, and the validation accuracy starting at approximately 0.93 and ending at 0.98.

```
Entrée [6]: batch_size = 128
epochs = 200

input_shape= (train_x.shape[1], train_x.shape[2], 1)
nb_classe= len(etiquette)

model2 = Sequential()

model2.add(Conv2D(16, kernel_size=(3, 3),activation='relu',input_shape= input_shape ))

model2.add(Conv2D(32, (3, 3), activation='relu'))
model2.add(MaxPooling2D(pool size=(2, 2)))
```

```

model2.add(Dropout(0.2))

model2.add(Conv2D(32, (3, 3), activation='relu'))
model2.add(MaxPooling2D(pool_size=(2, 2)))

model2.add(Flatten())

model2.add(Dense(256, activation='relu'))
model2.add(Dropout(0.3))

model2.add(Dense(nb_classe, activation='softmax'))
model2.compile(loss=keras.losses.categorical_crossentropy,optimizer=keras.optimizers.Adadelta(),metrics=['accuracy'])

model2.summary()

```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 16)	160
conv2d_1 (Conv2D)	(None, 24, 24, 32)	4640
max_pooling2d (MaxPooling2D)	(None, 12, 12, 32)	0
dropout (Dropout)	(None, 12, 12, 32)	0
conv2d_2 (Conv2D)	(None, 10, 10, 32)	9248
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 32)	0
flatten (Flatten)	(None, 800)	0
dense (Dense)	(None, 256)	205056
dropout_1 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 10)	2570
Total params: 221,674		
Trainable params: 221,674		
Non-trainable params: 0		

2022-03-28 18:20:43.866453: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libcuda.so.1'; dLError: libcuda.so.1: cannot open shared object file: No such file or directory
2022-03-28 18:20:43.866486: W tensorflow/stream_executor/cuda/cuda_driver.cc:269] failed call to cuInit: UNKNOWN ERROR (303)
2022-03-28 18:20:43.866511: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:156] kernel driver does not appear to be running on this host (debian-jpphi): /proc/driver/nvidia/version does not exist
2022-03-28 18:20:43.867898: I tensorflow/core/platform/cpu_feature_guard.cc:151] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations: AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.

Entrée [7]:

```

batch_size = 128
epochs = 200

input_shape= (train_x.shape[1], train_x.shape[2], 1)
nb_classe= len(etiquette)

model2 = Sequential()

model2.add(Conv2D(16, kernel_size=(3, 3),activation='relu',input_shape= input_shape ))

model2.add(Conv2D(32, (3, 3), activation='relu'))
model2.add(MaxPooling2D(pool_size=(2, 2)))
model2.add(Dropout(0.2))

model2.add(Conv2D(32, (3, 3), activation='relu'))
model2.add(MaxPooling2D(pool_size=(2, 2)))

model2.add(Flatten())

model2.add(Dense(256, activation='relu'))
model2.add(Dropout(0.3))

model2.add(Dense(nb_classe, activation='softmax'))
model2.compile(loss=keras.losses.categorical_crossentropy,optimizer=keras.optimizers.Adadelta(),metrics=['accuracy'])

model2.summary()

```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 26, 26, 16)	160
conv2d_4 (Conv2D)	(None, 24, 24, 32)	4640
max_pooling2d_2 (MaxPooling 2D)	(None, 12, 12, 32)	0
dropout_2 (Dropout)	(None, 12, 12, 32)	0
conv2d_5 (Conv2D)	(None, 10, 10, 32)	9248
max_pooling2d_3 (MaxPooling 2D)	(None, 5, 5, 32)	0