Projet de Machine Learning - MU4RBI05

Dié Jacques,
Etudiant au Master 1 Ingénierie des Systèmes Intelligents à Sorbonne Université
Email : jacques.die@etu.sorbonne-universite.fr

Abstract— Ce papier rend compte de différentes méthodes de Machine Learning ainsi que leurs optimisations utilisées pour reconnaître des caractères manuscrits numérisés à l'aide d'un scanner.

I. INTRODUCTION

Dans le cadre de notre cursus à Sorbonne Université, l'unité d'enseignement (UE) de Machine Learning nous a permis de manipuler de nombreux outils au cours de travaux pratiques (TP) pour traiter différents cas de classification et de s'intéresser à leur fonctionnement ainsi qu'à leurs optimisations respectives.
Le projet présenté ici est pour ainsi dire un compte-rendu de l'expérience de l'UE.
Nous allons nous intéresser à quatre méthodes d'apprentissages supervisées déjà vues lors de précédents TP, à savoir :

- K plus proches voisins (avec ou sans analyses en composantes principales)
- Forêt d'arbres aléatoires
- Machines à vecteur de support multi-classes ;
- Réseaux de neurones

Ces dernières vont être appliquées à la reconnaissance de caractères manuscrits numérisés de taille 12x12 pixels (avec des lettres variant de B à K) sur une base de données composées de 2500 exemples.
Après discussion sur le principe des méthodes et de leur optimisation, nous choisirons quelle est la méthode la plus appropriée dans le cadre de notre exemple.
A noter que ce rapport doit être conjointement regardé avec le code fourni pour être le plus complet possible.

II. PRÉTRAITEMENT DES DONNEES

A. A. Format d'entrée

Avant de commencer à analyser les techniques évoquées en introduction, nous devons nous assurer que les données qu'elles traitent soient intelligibles.

Nous sommes sur des images de 12x12 pixels. Nous devons donc s'assurer que nos méthodes soient compatibles avec le format de nos données. Il faudra par exemple pour la méthode des KPPV les aplatir et les normaliser pour créer des vecteurs manipulables (de taille 12x12 = 144).

B. Création de la base de données

Pour la scission en données d'apprentissage et de tests, tout est déjà donné dans le fichier du projet 'data.npy'. Nous avons juste à utiliser numpy.load qui retourne 1875 échantillons pour les données d'apprentissage et 625 échantillons pour les données de test, soit une scission de 66% et 33% respectivement pour chaque dataset.

III. METHODE K PLUS PROCHES VOISINS (KPPV)

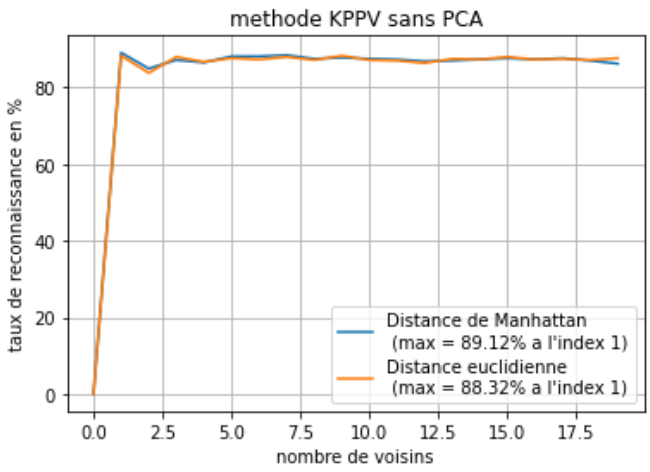
A. Principe de fonctionnement

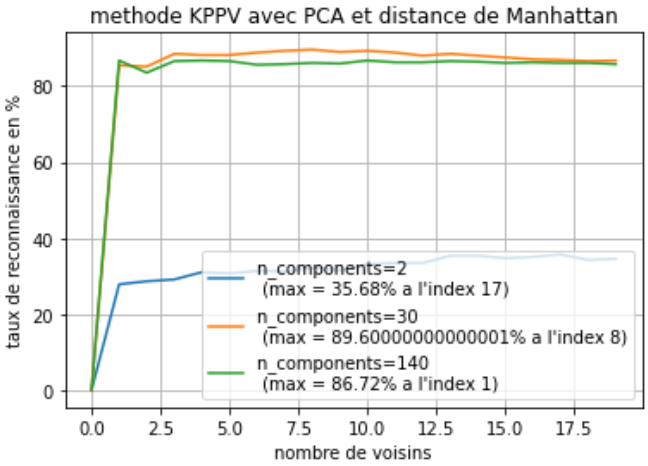
La méthode KPPV consiste à déterminer pour une observation donnée son groupe d'appartenance à partir du groupe d'appartenance de ses K les plus proches voisins (K étant donc le nombre de voisins à considérer).

Dans la situation donnée, on va prendre en compte deux paramètres : le nombre de voisins K et la méthode d'évaluation de la distance entre chaque observation (distance de Manhattan ou euclidienne).

Nous allons aussi appliquer une analyse en composante principale (ACP) de sorte à projeter les données dans un espace de dimension plus faible de sorte à réduire le nombre de variables. Le facteur que nous allons modifier est le nombre de components que nous voulons garder.

B. Résultats





On s'aperçoit que malgré les paramètres qu'on a modifié, le taux de reconnaissance avoisine les 90%. La distance de Manhattan semble plus adaptée pour le cas que l'on étudie et un nombre de components moyens que l'on garde semble mieux fonctionner.

IV. FORET D'ARBRES ALEATOIRES

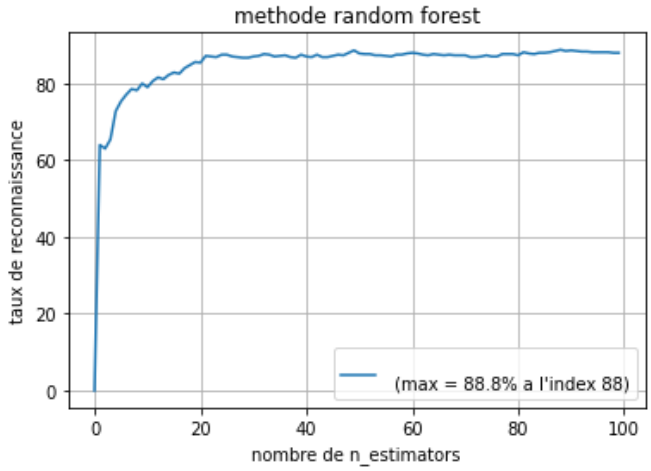
A. Principe de fonctionnement

Pour la méthode de « random forest », nous allons apprendre en parallèle sur plusieurs arbres de décisions construits aléatoirement et entraînés sur des sous-ensembles contenant des données différentes. Chaque arbre propose alors une prédiction et la prédiction finale consiste à réaliser la moyenne de toutes les prédictions, ou dans notre cas de classification, un vote est réalisé parmi les propositions de classification.

Ici, l'optimisation que l'on doit réaliser se trouve dans le bon nombre d'arbres à utiliser avec une bonne profondeur et un maximum de features que l'on doit considérer pour le meilleur split.
Heureusement, l'outil de gridsearch de sklearn va nous permettre d'automatiser la recherche de ces paramètres.

B. Résultats

En observant seulement la variation des n_estimators avec le reste des paramètres automatiques nous avons :



88,8% pour 88 arbres.

Au bout de 41 minutes, gridsearch nous retourne des paramètres « optimisés » pour notre random forest :

```
{'max_depth': 9, 'max_features': 0.4,
'n_estimators': 151}
91.03999999999999
```

Ce qui représente une optimisation d'environ 2.5%. A l'utilisateur de voir si 40 minutes d'attente valent la peine pour cette amélioration.

V. SVM

A. Principe de fonctionnement

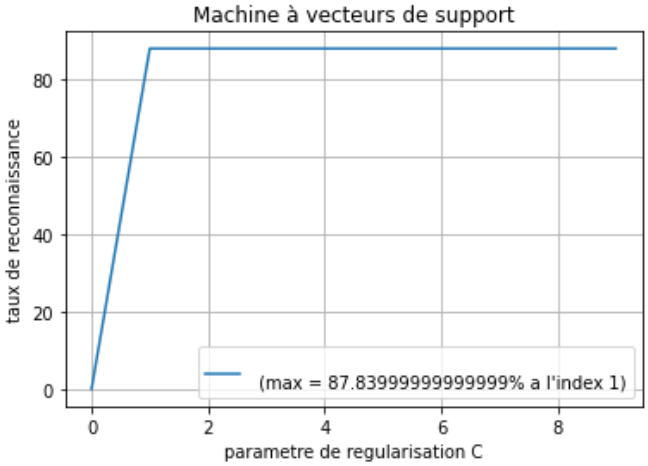
Le but de la méthode machine à vecteurs de support est de déterminer une frontière afin de séparer les observations en groupes distincts tout en maximisant la marge de séparation.

On peut déjà supposer que les données que nous avons sont difficilement linéairement séparables. On essayera alors de les projeter dans un espace vectoriel de plus grande dimension en appliquant un kernel par exemple.

Ici, nous allons manipuler trois paramètres :
C, qui correspond au paramètre de pénalité du terme d'erreur (contrôle de l'échange entre une décision de frontière agile et classer les échantillons de façon adéquate)
Gamma, le paramètre pour les hyper plans non linéaires (plus gamma est élevé, plus le SVM va essayer de faire correspondre exactement le training data set à la réalité)
Et le kernel, soit le type d'hyperplan pour séparer les datas (linear, rbf ou poly).

B. *Résultats*

D'une façon naïve, en ne faisant varier que le C nous avons :



Ce qui est en dessous des 90% auxquels les autres méthodes (à remarquer qu'un trop grand C ne rime à rien et amène à de l'over-fitting dans la plupart des cas)

Mais nous allons essayer de l'optimiser avec d'optimiser C et gamma avec GridSearch en prenant ces métaparamètres (avec 4 folds) :

```
Cs = [0.001, 0.01, 0.1, 1, 10]
gammas = [0.001, 0.01, 0.1, 1]
kernel=['rbf','poly','linear']
```

et nous obtenons alors :

```
Fitting 4 folds for each of 60 candidates,
totalling 240 fits
[Parallel(n_jobs=1)]: Using backend Se-
quentialBackend with 1 concurrent workers.

[Parallel(n_jobs=1)]: Done 240 out of 240
| elapsed:  2.5min finished
{'C': 0.001, 'gamma': 1, 'kernel': 'poly'}
```

Avec ces paramètres, nous obtenons un taux de reconnaissance à 92% pour un temps de calcul raisonnable.

VI. Reseaux de neurones

A. *Principe de fonctionnement*

Calqué sur le fonctionnement du cerveau humain, les réseaux de neurones artificiels sont utilisés sur un ensemble de données combinés avec une fonction coût qui indique dans quelle mesure il est éloigné du résultat souhaité. Le réseau s'adapte alors pour augmenter la précision de l'algorithme au fur et à mesure des epochs et des itérations.

On utilisera tensorflow ici pour matérialiser notre architecture neuronale.
Sachant que notre database n'est composé que de 2500 échantillons en tout, il nous est possible de déployer un grand dispositif de classification sans pour autant que ça prenne beaucoup de temps.

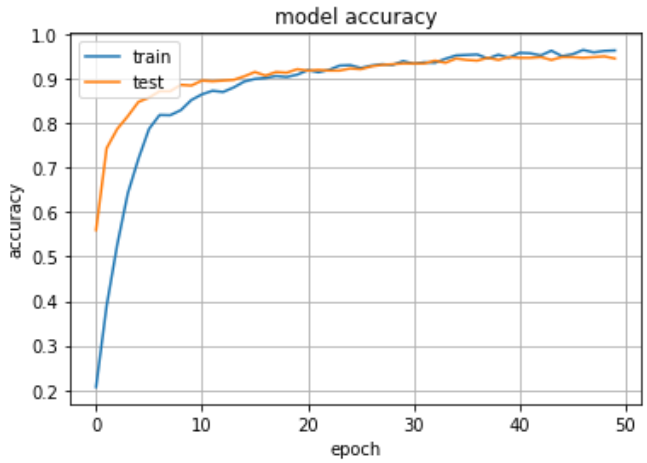
B. *Résultats*

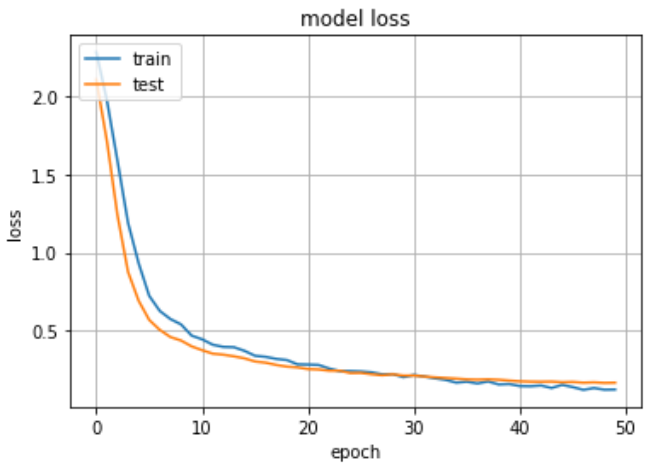
En reprenant les exemples vus dans les travaux pratiques de l'ue, nous avons cette architecture :

```
Model: "model_28"
_________________________________________________
_________________________
Layer (type)                 Output Shape
Param #
=================================================
=========================
input_41 (InputLayer)        [(None, 12,
12)]          0
_________________________________________________
_________________________
conv1d_40 (Conv1D)           (None, 12,
32)           1184
_________________________________________________
_________________________
conv1d_41 (Conv1D)           (None, 12,
64)           6208
_________________________________________________
_________________________
dropout_36 (Dropout)         (None, 12,
64)           0
_________________________________________________
_________________________
max_pooling1d_18 (MaxPooling (None, 6, 64)
0
_________________________________________________
_________________________
flatten_30 (Flatten)         (None, 384)
0
_________________________________________________
_________________________
dense_46 (Dense)             (None, 128)
49280
_________________________________________________
_________________________
dropout_37 (Dropout)         (None, 128)
0
_________________________________________________
_________________________
dense_47 (Dense)             (None, 11)
1419
=================================================
=========================
Total params: 58,091
Trainable params: 58,091
Non-trainable params: 0
```

On utilise ici un filtre 32 suivi d'un filtre 64 puis d'un dropout de 0.25 (pour éviter l'overfitting et améliorer la vitesse d'éxécution) suivie d'un maxpooling pour downsampler l'image et encore une fois réduire le temps de calcul de notre machine.

Une fois que nos données transformées ont été flatten, on le passe alors dans un réseau neuronal caché de taille 128 avant d'utiliser un autre dropout (de 0.5) puis finalement de sortir le résultat :






```
lr= 0.1 batch_size= 256 epochs= 50
Temps d apprentissage 13.198487999999998
```

Avec evaluate :

```
20/20 [==============================] - 0s 3ms/step - loss: 0.1574 - accuracy: 0.9472
[0.15736033022403717, 0.9472000002861023]
Test loss: 0.15736033022403717
Test accuracy: 0.9472000002861023
```

C'est un bon résultat pour très peu d'efforts et on pourrait s'en contenter mais, comme nous cherchons ici le meilleur taux de reconnaissance, voyons voir ce que l'on peut essayer pour obtenir un meilleur taux :

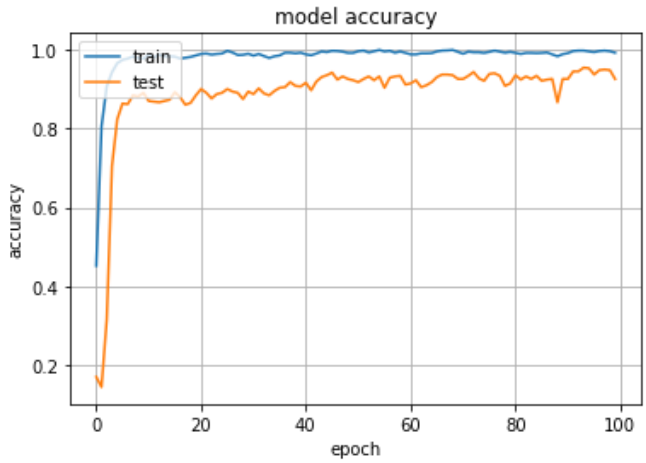
- On pourrait utiliser des architectures qui sont déjà disponibles dans les library données comme VGG16 ou ResNet (en faisant attention à bien convertir nos données d'entrées)

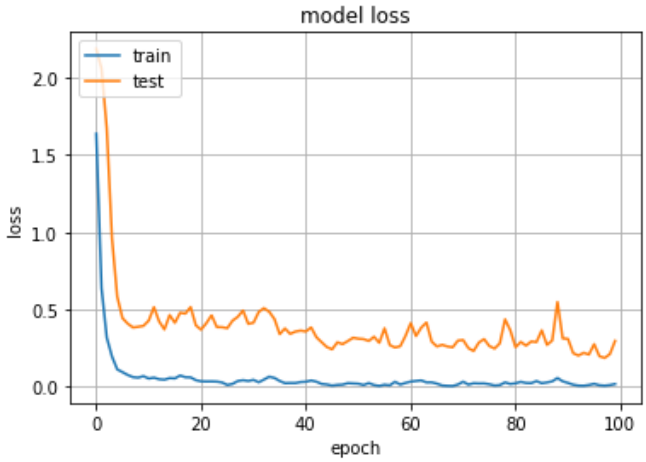
- Dans le même esprit que gridsearch, nous pouvons utiliser kerasTuner pour essayer de trouver des métaparamètres sur un modèle

En utilisant HyperREsNet (qui reprend le modèle de ResNet) de KerasTuner avec ces paramètres (ici les data données ont été converties en 28x28 et augmenté d'une dimension pour qu'elles soient compatibles avec l'architecture proposé) :

```
hypermodel2 = HyperResNet(input_shape=(28, 28, 1), classes=10)

hp = HyperParameters()
hp.Choice('learning_rate', values=[1e-3, 1e-4])
hp.Fixed('optimizer', value='adam')


tuner_hb = Hyperband(
            hypermodel2,
            hyperparameters=hp,
            tune_new_entries=False,
            max_epochs=50,
            objective='val_accuracy',
            seed=42,
            executions_per_trial=3
        )

tuner_hb.search(rgb_X_train, Y_train,epochs=2,batch_size=128, validation_split=0.1)
```

Nous obtenons alors ces résultats :



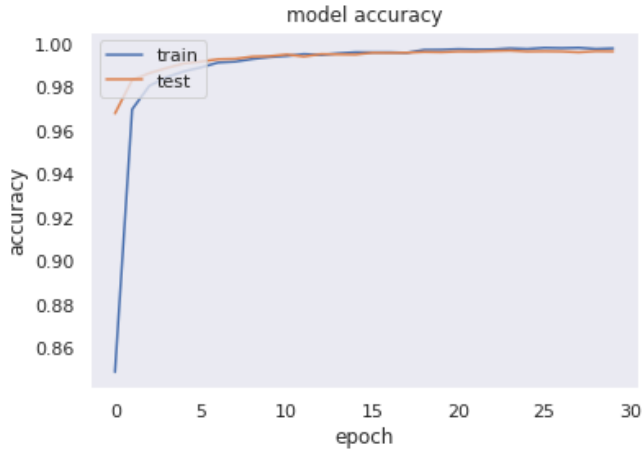


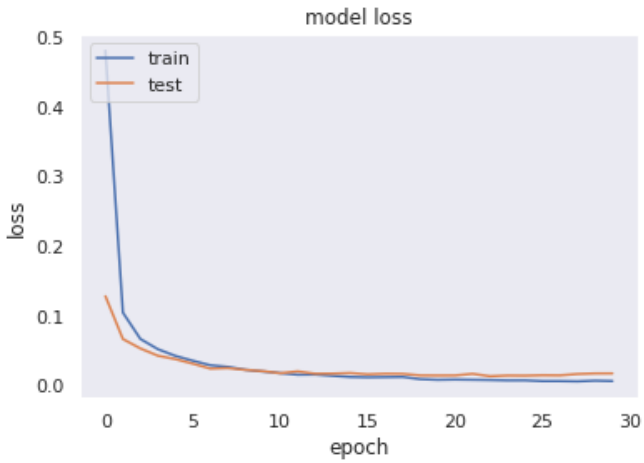
Avec une val_accuracy en moyenne dans les 0.94 ce qui est assez faible pour les calculs faits. Peut être est-ce dû à la faible database ou un mauvais paramétrage de l'outil…

- On peut aussi étendre notre base de données en créant des images transformées avec les images de base pour rendre notre précision plus robuste.

Mais au vu de la database assez commune, Kaggle.com possède des bases de données déjà toutes faites que nous pouvons manipuler.
On va utiliser alors « az-handwritten-alphabets-in-csv-format » qui contient 372450 exemples de lettres de l'alphabet. Ici, nous ne nous intéressons qu'aux lettres de B à K et nous allons filtrer cette database pour n'avoir plus que les labels qui vont nous intéresser(environ 90 000 exemples).
On reshape alors nos données pour les manipuler dans notre architecture et on obtient alors :

```
779/779  [==============================] - 2s 3ms/step - loss: 0.0171 - accuracy: 0.9965
[0.017055269330739975, 0.9965064525604248]
Test loss: 0.017055269330739975
Test accuracy: 0.9965064525604248
```





99,6% de taux de reconnaissance avec une database vingt fois plus grandes mises dans notre architecture vue dans nos TPS.

VII. CONCLUSION

Si ce qui nous intéresse est l'optimisation du taux de reconnaissance, il n'y a pas vraiment besoin de réfléchir : la méthode passant par le réseau de neurones est la meilleure réponse que l'on peut apporter !
Cependant il faut satisfaire beaucoup de conditions avec notamment une puissante capacité de calcul (l'exemple précédent a été obtenu avec une accélération GPU de Kaggle), une architecture adaptée et une database assez grande pour permettre un entrainement robuste.

Les autres méthodes ne sont pas pour autant à proscrire vu qu'elles obtiennent de bons résultats pour un moindre effort, reste à déterminer dans quelle situation les employer.