ML : Deep neural network

TP6 Apprentissage automatique

Réseaux de neurones profonds

Jérôme Buisine
jerome.buisine@univ-littoral.fr

9 janvier 2023

Durée : 3h

L'objectif de ce TP est la prise en main de la librairie Keras afin d'utiliser et d'évaluer les performances des réseaux de neurons profonds.

1 Ressources

Voici un ensemble de ressources qui peuvent vous être utiles durant ce TP :

— 1. Documentation officielle de l'outil `Git` ;

— 2. Gitlab : interface web pour la gestion de projets versionnés sous `Git` ;

— 3. pyenv : permet la gestion d'environnement Python.

— 4. matplotlib : librairie Python qui permet un affichage rapide de données.

— 5. jupyter : un outil de travail permettant une interaction rapide et visuelle avec une console Python.

— 6. scikit-learn : documentation de l'API `scikit-learn` utile pour le *machine learning*.

— 7. Keras : documentation de l'API `Keras` pour le *deep learning*.

2 Configuration de l'environnement

Créez un dossier à la racine de votre projet nommé `tp6-dnn`. Dans ce dossier et depuis votre terminal, lancez les commandes suivantes :

```
# spécifie l'environnement virtuel Python à Jupyter
ipython kernel install --user --name=ml-venv
# lance l'application Jupyter
jupyter-lab
```

3 Prédiction de la qualité du vin

Il vous faudra tout d'abord installer les dépendances qui permettront d'utiliser les API `Keras` (via `tensorflow`) et `seaborn` (librairie basée sur `matplotlib`) pour un affichage plus rapide :

```
pip install tensorflow seaborn
```

Puis créez un nouveau notebook nommé « dnn_wine.ipynb ». L'objectif est de proposer un premier réseau de neurones profond pour prédire la qualité d'un vin. On sera donc dans le contexte d'un problème de **régression**. Pour cela, récupérez la base de données « red wine quality » disponible au format `csv`.

Tâche 1 : Charger les données à l'aide de la librairie `pandas`. Puis proposez une séparation aléatoire de données à hauteur de 80% pour l'ensemble d'apprentissage et 20% pour l'ensemble de test.

Indications :

→ Le champs à prédire sera « `quality` ».

→ Vous pouvez utiliser la librairie `scikit-learn` pour effectuer cette séparation.

Une fois les données séparées, nous allons les standardiser pour faciliter l'apprentissage de notre futur modèle. Cette standardisation sera fera de la manière suivante :

$$z = \frac{x_i - \mu}{\sigma}$$

avec :

- x_i un vecteur d'entrée du modèle
- μ les moyennes de chaque caractéristique
- σ les écart-types de chaque caractéristique

Tâche 2 : À l'aide de la librairie « `numpy` », effectuer la standardisation de ces données d'entrée.

Indications :

→ La moyenne et l'écart-type seront calculés uniquement sur les données d'apprentissage.

→ Ils seront utilisés à la fois pour la standardisation des bases de test et d'apprentissage (dans la réalité, nous ne connaissons que les données d'apprentissage).

Tâche 3 : Créer maintenant une fonction `get_model` qui prend en paramètre la géométrie des données d'entrée et retourne un modèle.

- Ce modèle sera de type `Sequential` (voir `keras.models`).
- Il sera composé d'un layer `Input` qui prendra en paramètre la géométrie des données d'entrée.
- De deux layers cachés `Dense`, tous deux composés de fonctions d'activation de type `ReLu` et respectivement 128 et 32 neurones.
- Enfin, un dernier layer de type `Dense` avec un neurone. Il s'agira du layer de sortie.

Indications :

→ Dans le cadre de la régression, il ne faut pas proposer de fonction d'activation dans la couche de sortie.

→ La liste des layers disponibles est présente dans le module `keras.layers`.

Vous pouvez maintenant proposer un affichage sommaire de votre modèle à l'aide de la fonction `summary` :

```
Model: "sequential_1"
_________________________________________________________________
   Layer (type)                Output Shape              Param #
=================================================================
   layer_dense1 (Dense)        (None, 128)               1536

   layer_dense2 (Dense)        (None, 32)                4128

   output (Dense)              (None, 1)                 33

=================================================================
Total params: 5,697
Trainable params: 5,697
Non-trainable params: 0
```

On peut y retrouver des informations pour chaque couche de notre modèle, ainsi que le nombre de paramètres (poids en l'occurrence ici) qui seront à apprendre.

Avant de procéder à l'apprentissage de notre modèle, il faut le compiler. C'est-à-dire, préciser l'optimiseur, la fonction de perte et les métriques qui seront utilisés lors de l'apprentissage et son évaluation.

```
model.compile(optimizer='rmsprop', loss= 'mse', metrics= ['mae', 'mse'])
```

Il est également possible de proposer des fonctions « `callbacks` » qui seront appelées périodiquement lors de l'apprentissage, par exemple après chaque époque. En voici une, qui permet de sauvegarder le meilleur modèle obtenu et que nous utiliserons plus tard :

```
model_folder = 'models/wine/best_model.h5'
os.makedirs('models/wine', exist_ok=True)
save_callback = keras.callbacks.ModelCheckpoint(filepath=model_folder,
                                                verbose=0,
                                                save_best_only=True)
```

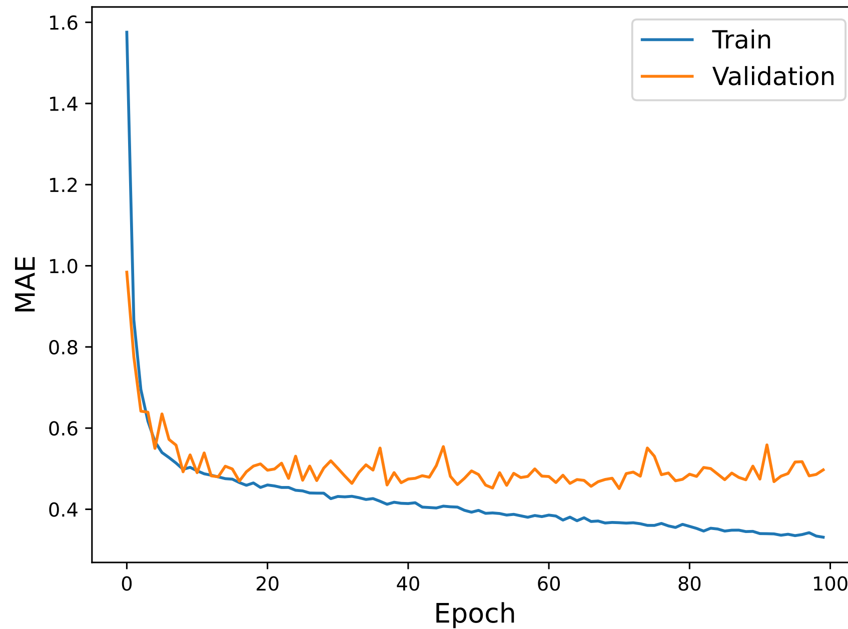
Note : le format `h5` est un format standard qui permet de charger un modèle complet, c'est-à-dire son architecture ainsi que ses poids.

Tâche 4 : Proposez un apprentissage de votre modèle à l'aide de sa fonction `fit`. Il vous faudra pour la suite stocker le retour de cette fonction dans une variable, par exemple `history`. Vous allez pouvoir lui fournir les informations suivantes :

- Les données d'entrée de la base d'apprentissage et les prédictions attendues.
- Un nombre d'époque à 50.
- Un batch de taille 8.
- Des données de validation (`validation_data`) qui seront les données de test.
- Le callback de sauvegarde dans le paramètre « `callbacks` » qui attend une liste de fonctions.

Tâche 5 : À partir de la variable `history` obtenue en retour de l'apprentissage de votre modèle, affichez les courbes d'apprentissage et de validation de votre modèle au cours des époques.

Voici un exemple de courbes courbes d'apprentissage et de validation obtenues pour la MAE sur 100 époques :



Tâche 6 : Réaliser une évaluation à partir du meilleur modèle obtenu. Il sera évalué sur les données de test à partir de sa méthode `evaluate`. Cette méthode retourne plusieurs valeurs : la loss et les métriques obtenues qui avaient été précisées.

Indications :

→ Il est possible de charger un modèle et ses poids entraînés à partir d'un fichier d'extension `h5`.

→ La méthode `load_model` du module `keras.models` charge un modèle sous ce format.

→ Pour rappel, le meilleur modèle était sauvegardé à l'aide de notre callback dans le fichier « models/wine/best_model.h5 ».

Vous devriez obtenir quelque chose du genre :

```
loss : 0.4676
mae  : 0.4971
mse  : 0.4676 # same as loss
```

Questions

- Comment est-il possible d'interpréter les courbes d'apprentissage obtenues ?
- Qu'en est t'il de la performance moyenne de notre modèle pour la prédiction de la qualité du vin ?

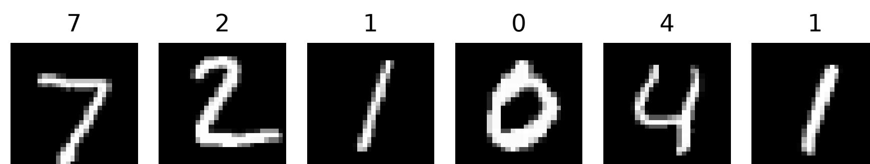
4 Prédiction de chiffres manuscrits

Nous allons maintenant proposer un modèle pour la détection de chiffres manuscrits à partir de la base de données `Mnist`. C'est un problème assez classique de la littérature en *machine learning* qui consiste à prédire à partir d'une image, un chiffre inscrit (de 0 à 9). Créez un nouveau notebook nommé « dnn_mnist.ipynb » pour traiter ce problème de classification multi-classe.

Il est possible de charge la base `Mnist` depuis `Keras` :

```
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()
print("x_train : ", x_train.shape)
```

Voici ci-dessous un exemple de chiffres proposés dans la base d'apprentissage que nous allons utiliser :



Le code qui permet cet affichage est précisé ci-dessous. Pour la suite du TP, vous pouvez vous inspirer de cet exemple pour vérifier visuellement vos prédictions :

```
nimages = 6
fig, axs = plt.subplots(1, nimages)

for i in range(nimages):
   label = y_test[i]
   image = x_test[i]

   image = np.array(image, dtype='float')
   pixels = image.reshape((28, 28))
   axs[i].imshow(pixels, cmap='gray')
   axs[i].axis('off')
   axs[i].set_title(label)

plt.show()
```

Comme d'habitude, il nous faut préparer les données et dans ce cas, les normaliser. C'est une étape incontournable pour que le modèle puisse correctement interpréter les entrées.

Tâche 7 : Proposez une normalisation des données par la valeur maximale d'une images 8 bits, soit 255 afin que chaque valeur de pixel soit $\in [0, 1]$. Cette normalisation doit être faites sur les deux ensembles d'entrées : apprentissage et validation.

Tâche 8 : Créer maintenant une fonction `get_model_mnist` qui prend en paramètre la géométrie des données de celui-ci et retourne un modèle.

- Ce modèle sera de type `Sequential` (voir `keras.models`).

- Il sera composé d'un layer `Input` qui prendra en paramètre la géométrie des données d'entrée, ici 28×28.
- D'un layer `Flatten` qui permet l'aplatissement des données en un vecteur afin de pouvoir connecter correctement nos couches suivantes.
- Puis, de deux layers `Dense`, tous deux composés de fonctions d'activation de type `ReLu` et respectivement 128 et 32 neurones.
- Enfin, un dernier layer de type `Dense` avec un nombre de neurones correspondant au nombre de classes à prédire. Il s'agira du layer de sortie.

Indications :

→ Dans le cadre de la classification multi-classe, il faudra utiliser une fonction d'activation de type `Softmax` pour émettre un vecteur de probabilité.

→ La liste des layers disponibles est présente dans le module `keras.layers`.

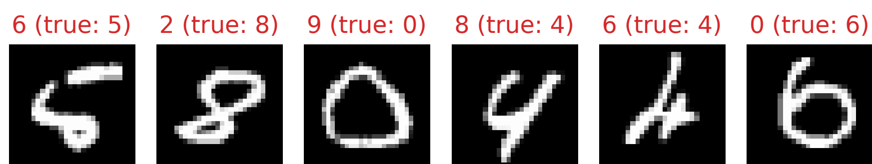
Avant de procéder à l'apprentissage de notre modèle, il faut le compiler. Ici nous utilisons une fonction de perte adaptée à notre problème de catégorisation, l'optimiseur `adam` et une métrique d'exactitude.

```
model.compile(optimizer='adam',
                loss='sparse_categorical_crossentropy',
                metrics=['accuracy'])
```

Tâche 9 : Proposez un apprentissage de votre modèle à l'aide de sa fonction `fit`, puis observez les résultats. Il vous faudra pour la suite stocker le retour de cette fonction dans une variable, par exemple `history`. Vous allez pouvoir lui fournir les informations suivantes :

- Les données d'entrée de la base d'apprentissage et les prédictions attendues.
- Un nombre d'époque à 20.
- Un batch de taille 512.
- Des données de validation (`validation_data`) qui seront les données de test.
- Une fonction callback de sauvegarde spécifique au modèle.

Voici un exemple de prédictions erronées sur la base de test par le modèle actuellement :



Il est possible pour analyser ces erreurs de dresser la matrice de confusion sur les 10 classes à prédire. Pour cela nous allons utiliser les librairies `scikit-learn` et `seaborn` :

```
from sklearn.metrics import confusion_matrix
import seaborn as sns

y_prediction = model.predict(x_test)
y_prediction = np.argmax(y_prediction, axis=1)
```

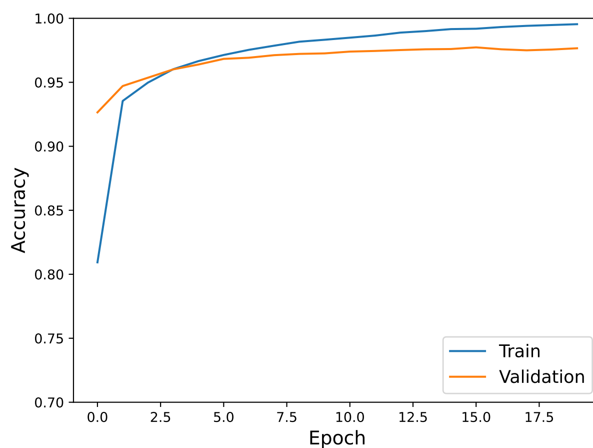
```
result = confusion_matrix(y_test, y_prediction , normalize='pred')

fig, ax = plt.subplots(1, 1, figsize=(12, 10))
sns.heatmap(result, annot=True, annot_kws={"fontsize":9}, cmap="Blues", ax=ax);
```

Note : les prédictions en sortie du modèle sont des probabilités d'appartenance de classe. Il est possible à l'aide de la fonction `argmax` de numpy de récupérer l'indice (classe) avec la plus grande probabilité.

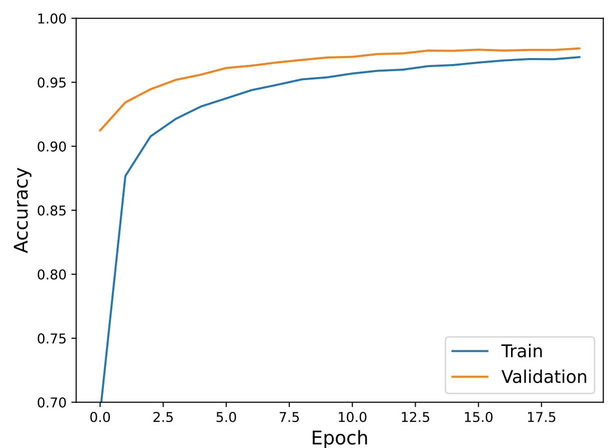
Tâche 10 : À partir de l'analyse des courbes d'apprentissage et de validation de notre modèle actuel, il est possible de visualiser que les tendances entre les deux courbes au cours de l'apprentissage ne sont pas les mêmes. Nous allons proposer une seconde version de notre modèle :

- La structure du modèle restera dans l'ensemble la même. Toutefois, une couche de `Dropout` sera ajoutée à la suite de chacune des couches cachées de notre réseau.
- Pour rappel, le `Dropout` permet lors de la *backprogation* d'éviter de mettre à jour tous les poids de notre réseau. Pour cela, il faut définir un pourcentage ($p \in [0, 1]$) de poids choisis aléatoirement à chaque rétropropagation de l'erreur et qui ne sont pas mis à jour.

Voici un aperçu des courbes d'apprentissage et de validation pour les deux versions de modèles :



(a) Modèle 1



(b) Modèle 2

Questions

- Avez-vous remarqué un gain de performance sur la base de test à l'aide des couches de `dropout` ? Si oui, à partir de combien de pourcentage ?
- Quel est l'avantage du `dropout` dans le cadre de l'apprentissage d'un modèle ?

5 Remise des travaux

N'oubliez pas de réaliser un commit de vos travaux. Taguez également votre projet avec le tag « `tp6` » et soumettez-le sur le serveur Gitlab. Il fera office de rendu.