TD : `keras` & perceptron multi-couches

Romain Tavenard

Dans cette séance, nous nous focaliserons sur la création et l'étude de modèles de type perceptron multi-couches à l'aide de la librairie `keras`.

Pour cela, vous utiliserez la classe de modèles `Sequential()` de `keras`. Voici ci-dessous un exemple de définition d'un tel modèle :

```
from keras.models import Sequential
from keras.layers import Dense

#1. définir les couches et les ajouter l'une après l'autre au modèle
premiere_couche = Dense(units=12, activation="relu", input_dim=24)
couche_cachee1 = Dense(units=12, activation="sigmoid")
[..]
couche_sortie = Dense(units=3, activation="linear")

model = Sequential()
model.add(premiere_couche)
model.add(couche_cachee1)
[...]
model.add(couche_sortie)

#2. Spécifier l'algo d'optimisation et la fonction de risque à optimiser
# Fonctions de risque classiques :
# * "mse" en régression,
# * "categorical_crossentropy" en classification multi-classes
# * "binary_crossentropy" en classification binaire
# On peut y ajouter des métriques supplémentaires (ici taux de bonne
# classifications)

model.compile(optimizer="sgd", loss="mse", metrics=["accuracy"])

#3. Lancer l'apprentissage
model.fit(X_train, y_train, verbose=2, epochs=10, batch_size=200)
```

1 Préparation des données

Pour ce TD, un module `dataset_utils` vous est fourni sur CURSUS. Il permet de récupérer des jeux de données sur lesquels s'entrainer en TD. Dans ce TD, nous travaillerons sur le jeu MNIST, dans lequel les exemples fournis sont des chiffres écrits à la main et la classe à prédire est le chiffre effectivement représenté.

Dans le module `dataset_utils`, une fonction `prepare_mnist` est fournie à laquelle on doit juste passer un booléen indiquant si l'on travaille depuis une machine du réseau MASS (auquel cas les jeux de données sont déjà téléchargés) ou une machine perso.

1. Observez le code des fonctions `prepare_mnist` et `prepare_boston`. Que font ces fonctions ? Quelles sont les dimensions des matrices / vecteurs à la sortie ? S'agit-il de problèmes de classification ou de régression ?

2 Premiers réseaux de neurone

2. Chargez le jeu de données MNIST et apprenez un premier modèle sans couche cachée avec une fonction d'activation raisonnable pour les neurones de la couche de sortie. Pour cette question comme pour les suivantes, limitez vous à un nombre d'itérations de l'ordre de 10 : ce n'est absolument pas réaliste, mais cela vous évitera de perdre trop de temps à scruter l'apprentissage de vos modèles.
3. Comparez les performances de ce premier modèle è celle de modèles avec respectivement 1, 2 et 3 couches cachées de 128 neurones chacune. Vous utiliserez la fonction ReLU (`"relu"`) comme fonction d'activation pour les neurones des couches cachées.
4. On peut obtenir le nombre de paramètres d'un modèle à l'aide de la méthode `count_params()`. Comptez ainsi le nombre de paramètres du modèle à 3 couches cachées et définissez un modèle à une seule couche cachée ayant un nombre comparable de paramètres. Parmi ces deux modèles, lequel semble le plus performant ?

3 Utilisation d'un jeu de validation

Bien entendu, les observations faites plus haut ne sont pas suffisantes, notamment parce qu'elles ne permettent pas de se rendre compte de l'ampleur du phénomène de sur-apprentissage.

Pour y remédier, `keras` permet de fixer, lors de l'appel à la méthode `fit()`, une fraction du jeu d'apprentissage à utiliser pour la validation. Jetez un oeil ici pour comprendre comment les exemples de validation sont choisis.

5. Répétez les comparaisons de modèles ci-dessus en vous focalisant sur le taux de bonnes classifications obtenu sur le jeu de validation (vous prendrez 30% du jeu d'apprentissage pour votre validation).

4 Régularisation et *Drop-Out*

6. Appliquez une régularisation de type L_1 à chacune des couches de votre réseau. L'aide disponible ici devrait vous aider.
7. Au lieu de la régularisation L_1, choisissez de mettre en place une stratégie de *Drop-Out* pour aider à la régularisation de votre réseau. Vous éteindrez à chaque étape 10% des poids de votre réseau.

5 Modèles `keras` dans `sklearn`

Il est possible de transformer vos modèles `keras` (en tout cas, ceux qui sont de type `Sequential`) en modèles `sklearn`. Cela a notamment pour avantage de vous permettre d'utiliser les fonctionnalités de sélection de modèles vues lors du TD précédent.

Pour cela, vous devrez utiliser au choix l'une des classes `KerasClassifier` ou `KerasRegressor` (selon le problème de *machine learning* auquel vous êtes confronté) du module `keras.wrappers.scikit-learn`.

Le principe de fonctionnement de ces deux classes est le même :

```
clf = KerasClassifier(build_fn=ma_fonction, param1=12, param2="sgd", ...)
clf.fit(X, y)
clf.predict(X_test)
```

Une fois construit, l'objet `clf` s'utilise donc exactement comme un classifieur `sklearn`. L'attribut `build_fn` prend le nom d'une fonction qui retourne un modèle `keras`. Les autres paramètres passés lors de la construction du classifieur peuvent être :

— des paramètres de votre fonction `ma_fonction` ;
— des paramètres passés au modèle lors de sa compilation (appel à la méthode `compile()`) ou de son apprentissage (appel à la méthode `fit()`).

8. Créez un réseau à deux couches cachées transformé en objet `sklearn` en spécifiant, lors de sa construction, le nombre d'itérations et la taille des *batchs* de votre descente de gradient par *mini-batchs*. Vous pourrez utiliser la méthode `score()` des objets `sklearn` pour évaluer ce modèle.
9. Utilisez les outils de validation croisée de `sklearn` pour choisir entre les algorithmes d'optimisation `"rmsprop"` et `"sgd"`.

6 La notion de `Callback`

Les *Callbacks* sont des outils qui, dans `keras`, permettent d'avoir un oeil sur ce qui se passe lors de l'apprentissage et, éventuellement, d'agir sur cet apprentissage.

Le premier *callback* auquel vous pouvez accéder simplement est retourné lors de l'appel à la méthode `fit()` (sur un objet `keras`, pas `sklearn`). Ce *callback* est un objet qui possède un attribut `history`. Cet attribut est un dictionnaire dont les clés sont les métriques suivies lors de l'apprentissage. À chacune de ces clés est associé un vecteur indiquant comment la quantité en question a évolué au fil des itérations.

10. Tracez les courbes d'évolution du taux de bonnes classifications sur les jeux d'entrainement et de validation.

La mise en place d'autres *callbacks* doit être explicite. Elle se fait en passant une liste de *callbacks* lors de l'appel à la méthode `fit()`. Lorsque l'apprentissage prend beaucoup de temps, la méthode précédente n'est pas satisfaisante car il est nécessaire d'attendre la fin du processus d'apprentissage avant de visualiser ces courbes. Dans ce cas, le *callback* `TensorBoard` peut s'avérer très pratique.

11. Visualisez dans une page TensorBoard l'évolution des métriques `"loss"` et `"accuracy"` lors de l'apprentissage d'un modèle.

De même, lorsque l'apprentissage est long, il peut s'avérer souhaitable d'enregistrer des modèles intermédiaires, dans le cas où un plantage arriverait par exemple. Cela peut se faire à l'aide du *callback* `ModelCheckpoint`.

12. Mettez en place un enregistrement des modèles intermédiaires toutes les 2 itérations, en n'enregistrant un modèle que si le risque calculé sur le jeu de validation est plus faible que celui de tous les autres modèles enregistrés aux itérations précédentes.
13. Mettez en oeuvre une politique d'arrêt précoce de l'apprentissage au cas où le risque calculé sur le jeu de validation n'a pas diminué depuis au moins 5 itérations (en utilisant le *callback* `EarlyStopping`).

7 Exercice de synthèse

14. Mettez en place une validation croisée pour choisir la structure (nombre de couches, nombre de neurones par couche) et l'algorithme d'optimisation idoines pour le problème lié au jeu de données *Boston Housing* (pour lequel une fonction de préparation des données est fournie dans le module `dataset_utils`).