Introduction à Keras

INF8460 - TP3

Polytechnique Montréal

Automne 2019

Aperçu de Keras

Introduction

Keras est une librairie Python pour l'apprentissage profond et l'utilisation de réseaux de neurones.
Attention : pour des datasets volumineux ou des modèles complexes, l'entraînement peut demander beaucoup de temps et de ressources. Il est très important d'utiliser une machine avec un GPU (par exemple Google Colab).

Introduction

Pour créer et entraîner un modèle Keras, on suit généralement trois étapes :

- **définir l'architecture du modèle** : spécifier les différentes couches du réseau et leurs paramètres, le format d'entrée et de sortie, etc.
- **compiler le modèle** : définir les modalités de l'entraînement, c'est-à-dire l'optimiseur (descente de gradient ou autre), la fonction de perte, et une liste de métriques à mesurer pendant les phases d'entraînement et de test
- **entraîner le modèle** sur les données d'entraînement

Définir un modèle

On initialise un modèle avec la classe `Sequential`:

```
from keras import Sequential

model = Sequential()
```

Puis on ajoute les couches (*layers*) qui le composent :

```
from keras.layers import Dense

# Ici, notre réseau a deux couches "fully-connected"
model.add(Dense(32, input_shape=(16,),
                activation="relu"))
model.add(Dense(1, activation="sigmoid"))
```

Définir un modèle (suite)

Syntaxe équivalente au code précédent :

```
model = Sequential([
    Dense(32, input_shape=(16,), activation="relu"),
    Dense(1, activation="sigmoid")
])
```

Ici, on a défini un perceptron avec :

- une couche d'entrée de dimension 16 : en Keras, on ne déclare pas la couche d'entrée, on indique simplement sa dimension avec `input_shape`
- une couche cachée de dimension 32
- une couche de sortie de dimension 1, qui retourne donc une valeur numérique. Comme sa fonction d'activation est une sigmoïde, cette valeur est comprise entre 0 et 1. Le réseau est donc adapté à la classification binaire

Voir la diapo 27 pour plus de détails sur les différents types de couches.

Définir un modèle (suite)

À ce stade, on peut afficher notre modèle avec `model.summary()` qui va renvoyer :

```
"""
Model: "sequential_1"
_________________________________________________
Layer (type)                 Output Shape              Param #
=================================================
dense_1 (Dense)              (None, 32)                544
_________________________________________________
dense_2 (Dense)              (None, 1)                 33
=================================================
Total params: 577
Trainable params: 577
Non-trainable params: 0
"""
```

Définir un modèle (fin)

Vérifions que le nombre de paramètres indiqués est correct :

- `dense_1` : 16 neurones d'entrée reliés à 32 neurones cachés, plus 32 biais : $32 \times 16 + 32 = 544$
- `dense_2` : 32 neurones en entrée reliés à 1 neurone de sortie, plus 1 biais : $1 \times 32 + 1 = 33$

Soit 577 paramètres au total.

Compiler un modèle

Une fois notre modèle défini, il faut spécifier les paramètres de l'entraînement :

- `optimizer`: l'algorithme d'optimisation
- `loss` : la fonction de perte
- `metrics` : une liste de métriques d'évaluation à calculer lors de l'entraînement

Par exemple :

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

Se reporter à la diapo 24 pour des détails sur les optimiseurs et les fonctions de perte

Entraîner un modèle

On peut alors entraîner un modèles sur nos données.
Commençons par générer des données :

```
from numpy.random import random, randint

# On génère 500 vecteurs de dimension 16
# et 500 labels valant soit 0 soit 1
X = random((500, 16))
y = randint(2, size=(500, 1))
```

Entraîner un modèle (suite)

On peut alors entraîner le modèle sur nos données avec la méthode `fit` :

```
model.fit(X, y,
          epochs=20,
          batch_size=32)
```

`epochs` désigne le nombre d'époques. Une époque correspond à un cycle d'entraînement complet, au cours duquel le modèle voit chaque élément du jeu de données.
`batch_size` désigne la taille des batchs (lots). Pendant l'entraînement, le modèle traite un lot de données de taille fixe, calcule son erreur, et rétropropage l'erreur dans le réseau, puis recommence avec un nouveau lot.

Entraîner un modèle : ensemble de validation

Remarque : c'est une bonne pratique de réserver une partie des données à des fins de validation (ici 10% du training set) :

```
from sklearn.model_selection import train_test_split

X_train, X_val, y_train, y_val = train_test_split(X,
                                                  y, test_size=0.1)
```

On passe alors cet ensemble de validation à la méthode `fit`; les métriques définies lors de la compilation seront alors évaluées sur l'ensemble de validation à chaque fin d'époque.

```
model.fit(X_train, y_train,
          epochs=20,
          batch_size=32, validation_data=[X_val, y_val])
```

Entraîner un modèle : historique d'apprentissage

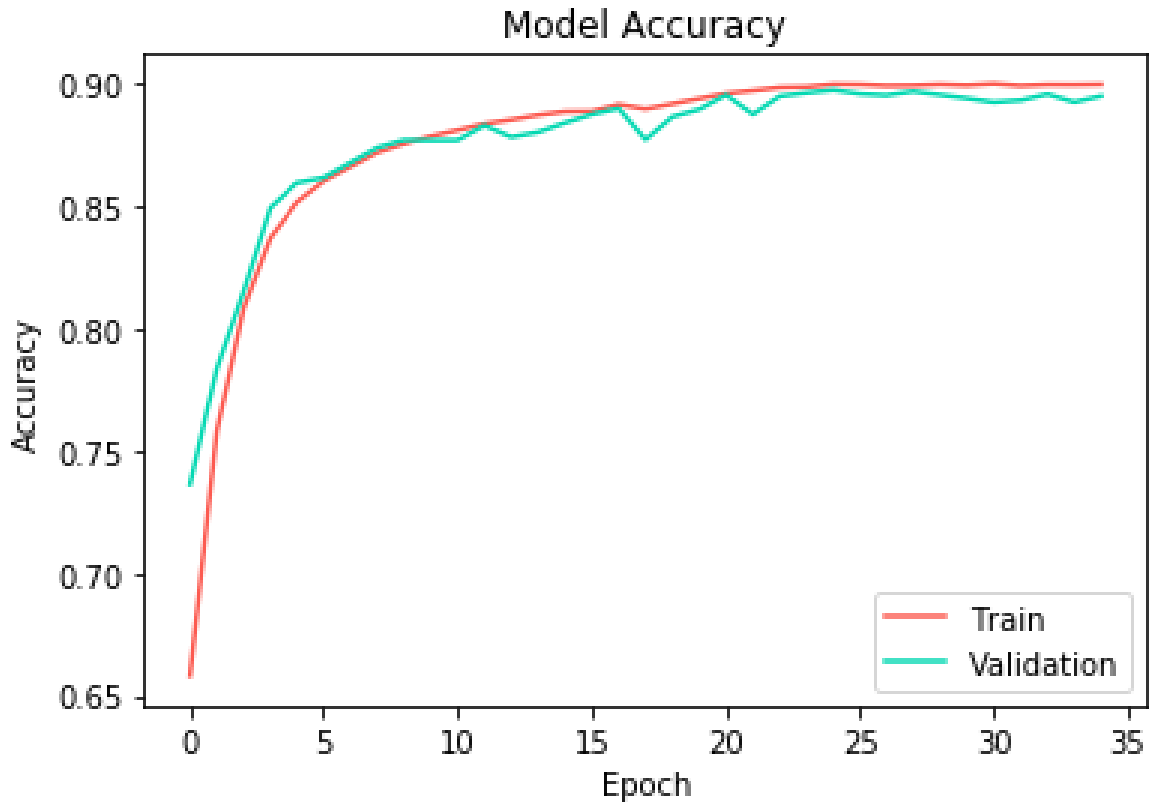
Remarque 2 : la méthode `fit` renvoie un objet `History`, qui contient les valeurs des différentes métriques à chaque époque. On peut l'utiliser pour afficher les courbes d'apprentissage.

```
import matplotlib.pyplot as plt

h = model.fit(...)
plt.plot(h.history['acc'], label="Training set")
plt.plot(h.history['val_acc'],label="Validation set")
plt.legend()
plt.show()
```

Entraîner un modèle : historique d'apprentissage

Le code précédent donnera une figure de ce type :



Évaluer et utiliser un modèle

Pour évaluer la performance d'un modèle sur un ensemble de test, on utilise `model.evaluate`, qui renvoie la perte (*loss*) ainsi que les métriques définies lors de la compilation :

```
results = model.evaluate(X_test, y_test)
```

Pour utiliser un modèle, on utilise `model.predict` :

```
y_pred = model.predict(X_test)
```

Résumé

En résumé, la structure générale d'un code Keras ressemble à :

```
model = Sequential([...])
model.compile(...)
print(model.summary())

h = model.fit(X_train, y_train, ...)
res = model.evaluate(X_test, y_test, ...)
```

Principales couches (*layers*)

`keras.layers`

Les couches de base

- `Dense(units, activation=None, use_bias=True, **args)` : tous les neurones d'entrée sont connectés à tous les neurones de sortie. C'est la couche de base d'un réseau de neurones
- `Dropout(rate, **args)` : annule aléatoirement certains poids lors de l'entraînement, pour éviter le sur-apprentissage.
- `Activation(activation)` : applique une fonction d'activation (ReLU, sigmoïde, tanh) à l'entrée. Pour la plupart des couches, vous pouvez utiliser le paramètre `activation` à la place

LSTM

Keras fournit une couche `LSTM` :

```
from keras.layers import LSTM

n_features = 10
sequence_len = 50
lstm_model = Sequential([
    LSTM(64, input_shape=(sequence_len, n_features),
    Dense(1, activation="softmax")
])
```

Ici, l'entrée doit être composée de séquences de longueur `sequence_len`, dont chaque élément est un vecteur de longueur `n_features`. Si vous avez des séquences de longueurs variables, utilisez `pad_sequences` du module `keras.preprocessing.sequence`.

LSTM (exemple)

Par exemple, si un corpus est composé de phrases de longueur 50 et a un vocabulaire de 1000 mots, une représentation possible des données serait :

- un mot est représenté par un vecteur one-hot de longueur 1000
- une phrase est une suite de 50 mots, soit une matrice 50×1000

Si on prend un LSTM de dimension cachée 256, on aurait :

```
lstm_model = Sequential([
    LSTM(256, input_shape=(50, 1000)),
    ...
])
```

Embedding

Pour gérer les *word embeddings* comme word2vec, Keras met à disposition une couche `Embedding`.

```
Embedding(input_dim, output_dim,
    input_len=None, weights=None, **args)
```

Pour un entier *i* compris entre 0 et `input_dim`, l'*embedding layer* retourne un vecteur dense de dimension `output_dim`.
`weights` permet d'utiliser des embeddings pré-entraînés (cf diapo suivante).

Utiliser des embeddings avec un LSTM

On doit disposer d'une matrice M d'embeddings déjà entraînés, ainsi que d'un dictionnaire qui associe un mot w à un entier i_w, tel que la i_w-ème ligne de M représente l'embedding de w.

- Étape 1 : on transforme chaque phrase $(u, v, w, \ldots)$ du corpus en séquence d'entiers $(i_u, i_v, i_w, \ldots)$.
- Étape 2 : on utilise `pad_sequences` pour que toutes les séquences fassent la même longueur `seq_len`.

Les données sont prêtes, on va pouvoir définir notre modèle.

Utiliser des embeddings avec un LSTM (suite)

```
vocab_size, embedding_dim = M.shape
model = Sequential([
    Embedding(vocab_size,
        embedding_dim,
        input_len=seq_len,
        weights=[M], trainable=False),
    LSTM(...), ...])
```

`vocab_size` contient la taille du vocabulaire, soit le nombre de lignes de M;
`embedding_dim` représente la dimension des *embeddings*, soit le nombre de colonnes de M;
`input_len` doit être égal à la longueur de l'entrée, ici les séquences sont de longueur `seq_len`
`trainable=False` permet de ne pas mettre à jour les embeddings lors de l'entraînement (puisqu'ils sont déjà entraînés)

Entraînement des modèles

Optimiseurs

L'optimiseur est l'algorithme qui va être utilisé pour mettre à jour les paramètres lors de l'entraînement.
Pour choisir un optimiseur, il suffit de passer son nom (`"sgd"`, `"rmsprop"`, `"adam"`, `"adadelta"`) à la méthode `compile`.
Cependant, pour utiliser d'autres paramètres que ceux par défaut, on peut aussi instancier un objet `optimizer`. Par exemple, pour régler un taux d'apprentissage (*learning rate*, ou `lr`) égal à $0,01$:

```
from keras.optimizers import SGD

sgd = SGD(lr=0.01)
model.compile(optimizer=sgd, loss='mse')
```

Parmi les optimiseurs classiques, on trouve RMSprop, Adadelta ou Adam.

Fonctions de perte

La fonction de perte est l'objectif qui va être minimisé lors de l'entraînement.
Comme pour les optimiseurs, on peut passer le nom d'une fonction de perte existante, ou déclarer sa propre fonction. Les plus utilisées sont :

- `mean_squared_error` pour les problèmes de régression
- `categorical_crossentropy` ou `binary_crossentropy` pour les problèmes de classification

Utilitaires

Prétraitement

Keras fournit quelques fonctions utiles pour le prétraitement des données, notamment :

- `to_categorical` pour l'encodage one-hot
- `pad_sequences` pour produire des séquences de même taille avec padding (dans le module `keras.preprocessing.text`)

Callbacks

Les *callbacks* sont des fonctions qui vont être appelées pendant l'entraînement et qui permettent par exemple :

- de sauvegarder votre modèle à chaque fin d'époque (`ModelCheckpoint`). Cela permet de recharger un modèle depuis le disque sans avoir à le ré-entraîner
- d'implémenter une procédure d'*early stopping*, qui consiste à arrêter l'entraînement lorsque les performances cessent d'augmenter (`EarlyStopping`)
- d'utiliser `TensorBoard` pour visualiser l'entraînement du modèle

Vous pouvez aussi définir vos propres callbacks avec `LambdaCallback` : par exemple, si vous entraînez un modèle de langue neuronal, vous pourriez définir un callback qui affiche un segment de texte généré par le modèle à chaque fin d'époque.

Callback (suite)

Pour utiliser des callbacks, il faut les passer à la méthode `fit` via le paramètre `callbacks` :

```
from keras.callbacks import ModelCheckpoint, EarlyStopping

# On peut paramétriser le
# nom du fichier où sauvegarder le modèle
save = ModelCheckpoint("weights.{epoch:02d}.hdf5")

# Ici, si 'val_loss' ne s'est pas améliorée depuis
# 2 époques, on interrompt l'entraînement
stop = EarlyStopping(monitor='val_loss', patience=2)
model.fit(X, y, ..., callbacks=[save, stop])
```

Ici, `"val_loss"` désigne la perte sur l'ensemble de validation