Réseaux de neurones avec Keras

TP 1– Outils pour l'apprentissage automatique

Python, TensorFlow et Keras : présentation

La programmation de réseaux de neurones peut s'avérer très complexe. Heureusement, il existe aujourd'hui des bibliothèques logicielles qui facilitent grandement le travail des développeurs, généralement écrites pour le langage **Python**. Les plus connues sont **TensorFlow** https://www.tensorflow.org et PyTorch https://pytorch.org.
Ces bibliothèques constituent actuellement le cœur du développement d'un très grand nombre d'applications utilisant des réseaux de neurones, et nécessitent une certaine expérience et un niveau d'expertise suffisant pour atteindre des performances optimales, notamment en termes de vitesse et d'utilisation des ressources matérielles (mémoire, GPU, stockage...).

Afin de nous familiariser avec la construction de réseaux de neurones sans perdre de temps dans des détails de programmation, nous utiliserons une bibliothèque qui se présente comme une sur-couche à TensorFlow (entre autres), et qui facilite la manipulation des concepts clés liés aux réseaux de neurones. Cette bibliothèque se nomme **Keras** : https://keras.io
Keras permet de réaliser du prototypage très rapidement et de valider des idées avant de développer une solution plus industrielle directement à partir de TensorFlow par exemple.

I. Première partie. Installation de TensorFlow et Keras

Afin de pouvoir réaliser les exercices du TP, nous allons tout d'abord installer sur nos machines TensorFlow et Keras, sachant que nous n'avons pas les droits d'administrateur sur ces machines. Dans cette optique, un outil pour construire des environnements Python isolés qui ne nécessitent pas ces droits a été installé sur les machines de TP.
Il s'agit de **virtualenv** https://virtualenv.pypa.io/en/stable/.
Si vous travaillez sur votre propre ordinateur, il vous est conseillé d'utiliser le même type d'outils (virtualenv, ou encore conda sous linux ou macOs).

Tout d'abord, installons TensorFlow grâce à l'outil **pip** livré avec virtualenv :

```
# Current release for CPU-only
pip install tensorflow
```

→ Comme indiqué en commentaire (# Current release for CPU-only), ceci installe une version de tensorflow qui n'utilise pas la puissance d'une carte GPU (de type NVIDIA). Ces cartes n'étant pas encore installées sur les machines de TP, nous nous contenterons de la puissance de CPU, beaucoup plus faible que celle des GPU pour les calculs utilisés lors de l'apprentissage des réseaux de neurones, mais suffisante pour les exercices visés par ce TP).

Ensuite, installons Keras de la même manière :

```
pip install keras
```

Note importante : Tensorflow 2 est sorti très récemment (30 septembre 2019), et Keras y est maintenant intégré nativement. Plus d'information ici :
https://www.tensorflow.org/guide/keras/overview
Il faudra modifier un peu les 'import' des exemples suivants pour utiliser cette nouvelle version.

Nous voilà prêts pour travailler avec des réseaux de neurones.

II. Deuxième partie. Premier MLP avec Keras

Programmer un réseau de neurones de type perceptron multi-couches avec Keras est très rapide avec ce que Keras appelle le modèle *Sequential*. Cela consiste à définir l'empilage de couches qui vont se succéder dans l'architecture neuronale que nous voulons construire.

```
#Voici la création d'une première couche
model = Sequential()
model.add(Dense(32, input_shape=(120,)))
```

→ Le modèle contient une couche cachée qui traite en entrée des données à 120 dimensions et qui produit des sorties à 32 dimensions.

```
model.add(Activation('tanh'))
```

→ La fonction d'activation associée à la couche cachée est la tangente hyperbolique. Plus d'informations sur les différentes fonctions d'activations se trouve ici : https://keras.io/activations/

Keras propose plusieurs syntaxes équivalents. Par exemple le code précédent peut également s'écrire :

```
model = Sequential()
model.add(Dense(32, input_dim=120, activation='tanh'))
```

Avant de pouvoir utiliser le modèle séquentiel et les classes 'Dense' et 'Activation', il est nécessaire d'écrire les deux lignes suivant en début de fichier :

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

Question 1 : *construire un MLP avec 2 couches cachées avec 200 neurones chacune et fonction d'activation de type 'relu' et une couche de sortie de type 'softmax'. Ce réseau traite des entrées de dimension 400 et propose 10 valeurs en sortie.*

Pour pouvoir utiliser le réseau de neurones que nous avons conçu, il est nécessaire de compiler le modèle (=réseau de neurones). C'est aussi l'occasion de définir la fonction de coût et l'optimiseur qui seront utilisés durant l'apprentissage du réseau de neurones. Vous trouverez la liste des fonctions de coût disponibles dans Keras ici : https://keras.io/losses/ et les différents optimiseurs ici : https://keras.io/optimizers/
Voici un exemple d'appel de compilation :

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

Question 2 : *compiler le MLP en y associant une fonction de coût de type cross entropie catégorielle et un optimiseur de type descente stochastique du gradient (SDG). Il sera probablement nécessaire d'importer des packages de Keras.*

Après avoir conçu notre réseau de neurones, il va falloir apprendre ses paramètres pour une tâche donnée. Nous allons cibler la reconnaissance de chiffres à partir d'image d'écritures manuscrites en utilisant le jeu de données MNIST : https://en.wikipedia.org/wiki/MNIST_database

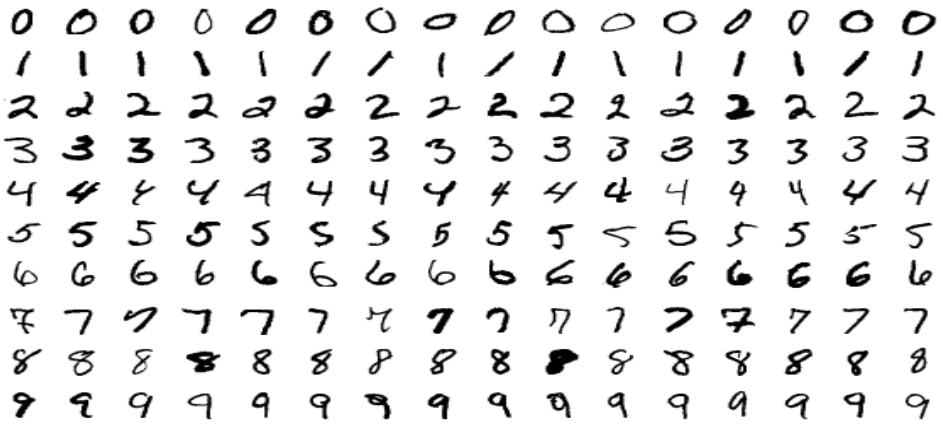

Illustration 1: Extraits du jeu de données MNIST

Ce jeu de données est très utilisé dans les tutoriaux et est intégré, parmi d'autres datasets, à Keras : https://keras.io/datasets/#mnist-database-of-handwritten-digits. Il est donc facile de le charger en mémoire durant nos expériences :

```
from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Il est alors nécessaire de formater les données pour qu'elles puissent être utilisées en entrée de notre MLP :

```
x_train = x_train.reshape(60000, nb_dimensions_entree)
x_test = x_test.reshape(10000, nb_dimensions_entree)
```

→ **Question 3** :*Que signifie ces deux lignes ? Trouvez la bonne valeur de* `nb_dimensions_entree`*. Vous pouvez afficher le contenu des différentes structures de données impliquées ici.*

Le problème que nous essayons de régler est un problème de classification automatique multi-classes. Il est nécessaire de modifier les valeurs entières des étiquettes y que nous souhaitons trouver en vecteurs de type 1-hot afin de pouvoir intégrer ces informations en sortie de l'architecture neuronale :

```
y_train = np_utils.to_categorical(y_train, nb_classes)
```

```
y_test = np_utils.to_categorical(y_test, nb_classes)
```
→ **Question 4** : *Quelle valeur doit être affectée à la variable nb_classes ?*
→ Il est nécessaire d'importer la méthode `np_utils` (en début de fichier Python) :
```
from keras.utils import np_utils
```

Pour lancer l'apprentissage du modèle neuronal, Keras propose la méthode `fit` :
```
model.fit(x_train, y_train, nb_epoch=12, verbose=1,
validation_data=(x_test, y_test))
```

Enfin, Keras permet très simplement d'évaluer les performances du système final sur les données de test (ATTENTION : en général les données de validation utilisée lors de l'apprentissage sont distinctes des données de test qui ne devraient être utilisées qu'au moment de l'évaluation finale du système) :

```
score = model.evaluate(x_test, y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

Question 5 : *Procéder à l'apprentissage et à l'évaluation de ce premier perceptron multi-couches (MLP) ? Modifier le code si nécessaire*

Question 6 : *En agissant sur les différents paramètres à votre disposition (optimiseur et paramètres liés à l'optimiseur, topologie de l'architecture neuronale, fonction de coût, fonction de régularisation, de normalisation…, construire le système le plus performant possible*