Choix du langage de programmation

Nous avons choisi d'utiliser le langage *Python* pour coder le réseau de neurones.

En effet, *Python* est un langage qui possède de nombreux outils afin de simplifier la mise en œuvre d'applications mathématiques. Aussi, *TensorFlow (Google)*, *Theano*, *MXNet* et *CNTK (Microsoft)* sont quatre bibliothèques très utilisées pour mettre en place des réseaux de neurones bien qu'il en existe d'autres.

Ces bibliothèques ne sont pas spécialisées dans la création de réseaux de neurones mais proposent un plus large éventail d'applications concernant l'apprentissage automatique (« machine learning »). Par exemple Theano permet de manipuler et d'évaluer des expressions matricielles en utilisant la syntaxe de *NumPy*, une autre bibliothèque *Python* qui permet la manipulation de matrices (similaire à *Matlab*).

De plus, les fonctions regroupées dans ces bibliothèques sont très optimisées et sont souvent compilées ce qui permet d'obtenir une vitesse d'exécution supérieure à l'utilisation du Python interprété. L'utilisation d'un GPU (Graphical Processing Unit / carte graphique) est aussi très facilitée grâce à ces programmes. Cela augmente aussi énormément la vitesse de calcul car ce type de processeur est spécialisé dans le calcul matriciel et parallèle contrairement au CPU (Central Processing Unit / processeur) qui est efficace en calcul séquentiel (un processeur possède une dizaine de cœurs logiques quand une carte graphique en possède plusieurs milliers).

Cependant, la non spécificité de ces bibliothèques peut conduire à des écritures lourdes pour construire un réseau de neurones alors que l'on n'utilise pas toutes les capacités offertes par la bibliothèque.

Ainsi, nous n'utiliserons pas directement ces bibliothèques mais implémenterons notre code à l'aide de *Keras* une bibliothèque Python qui agit comme une API (Application Program Interface) pour les bibliothèques précédemment citées. C'est-à-dire que *Keras* sert d'intermédiaire avec *TensorFlow* par exemple.

Keras est une API de réseau de neurones de haut niveau : elle permet de programmer un réseau de neurone avec une syntaxe plus facilement appréhendable puisqu'elle est spécifiquement pensée pour ce type d'apprentissage automatique.

En utilisant *Keras*, un programme ne perd que très peu en vitesse d'exécution puisque ce sont les bibliothèques très optimisées qui vont être utilisées en arrière-plan.

Une seule bibliothèque à la fois peut être utilisée par Keras. Cependant, un programme écrit avec la syntaxe de *Keras* pourra être réutilisé après avoir changé de bibliothèque. Nous avons choisi d'utiliser *Keras* avec *TensorFlow* car étant tous deux développés par *Google*, leur couplage est facilité. En effet, *TensorFlow* a ajouté la prise en charge de *Keras* dans sa bibliothèque en 2017.

Syntaxe Keras

Création d'un modèle

Un réseau de neurones est considéré comme un « model ». On instancie un modèle séquentiel puis on peut ajouter des couches selon nos besoins.

Une couche où chaque nœud est connecté avec les suivant (graph complet) est appelée « Dense ». Pour les utiliser il faut tout d'abord les inclure dans le programme avec la commande « import ».

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

Pour créer une couche il faut donc instancier la classe *Dense*, son premier paramètre est le nombre de neurones et son deuxième le nombre de variables en entrées. Il est nécessaire de préciser le nombre de variable en entré de la première couche mais pas des suivantes : *Keras* le détermine directement.

On peut définir la dimension du vecteur en entrée avec *input_shape* ou *input_dim* et *input_length*.

```
# On ajoute une couche au modele "model"
# La couche possede 32 neurones et attend 784 variables en entré
model = Sequential()
model.add(Dense(32, input_shape=(784,)))
# Est équivalent
model = Sequential()
model.add(Dense(32, input_dim=784))
```

On définit aussi les fonctions d'activation pour chaque couche. Keras propose de nombreuses fonctions d'activation comme la sigmoid ou ReLu. On peut définir la fonction d'activation directement lors de l'instanciation de la couche (attribut de Dense) ou après. Il faut d'abord importer le module « Activation ».

```
from keras.layers import Activation, Dense

# On ajoute une couche qui possède 64 neurones
# La fonction d'activation est une sigmoide

model.add(Dense(64))
model.add(Activation('sigmoid')
# Est équivalent
model.add(Dense(64, activation='sigmoid'))
```

Finalement, un modèle peut ressembler à :

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

# On instancie le modèle
model = sequential()
# ajoute la première couche
model.add(Dense(n1, input_shape(300,), activation='relu'))
# on ajoute une deuxième couche
model.add(Dense(n2, activation='sigmoid'))
# la dernière couche
model.add(Dense(n3, activation='tanh'))
```

Finalisation du modèle

Lorsque le modèle est terminé, il faut le compiler. C'est cette étape qui va faire appel à la bibliothèque en arrière-plan (TensorFlow dans notre cas).

La compilation va déterminer la meilleure façon de représenter le réseau de neurone afin de l'entrainer. Pour cela, il faut préciser trois arguments : la **fonction de coût** à minimiser, la **méthode d'optimisation** chargée de trouver cette minimisation et une métrique.

La fonction de coût est sous la forme d'une fonction python prenant en paramètre deux arguments : *y_true* et *y_pred*. Le premier est le vecteur des valeurs attendues et le second représente les prédictions. Des fonctions très utilisées comme l'erreur quadratique moyenne sont directement présentes dans Keras sous la forme d'alias (*mse* pour « Mean Squared Error » par exemple).

La méthode d'optimisation peut être implémentée par un alias présent dans *Keras* ou par une instance de la classe *Optimizer*. Les fonctions très utilisées comme la méthode du gradient sont donc présentes dans *Keras*. Une méthode de descente de gradient est implémentée par SGD (« Stochastique Gradient Descent ») c'est-à-dire l'algorithme du gradient stochastique.

Finalement une compilation de modèle se fait sous la forme suivante :

```
from keras import optimizers
from keras import losses

# Instanciation du modèle
# ...

# Compilation

# La fonction d'optimisation
# Ici on prend une méthode de descente de gradient
opti = optimizers.SGD()
# ou
opti = 'sgd'

# La fonction de coût
# Ici on prend l'erreur quadratique moyenne
cout = 'mse'
# ou
cout = losses.mean_squared_error
# ou
cout = 'mean_squared_error'

model.compile(optimizer=opti, loss=cout, metrics=['accuracy'])
```

Entrainement du réseau de neurones

Pour entrainer le modèle, on utilise la fonction *fit*. Elle prend plusieurs arguments : premièrement les données de test sous la forme d'un tableau utilisant la syntaxe de la bibliothèque *Numpy*. Ces données sont accompagnées de leur label en deuxième argument.

Le nombre d'itérations pour l'entrainement du modèle sur les données est géré par les attributs *epochs* et *initial_epoch*. Un « epoch » est défini comme une itération sur l'ensemble des données.

La mise à jour des poids peut être effectuée après avoir entrainé le modèle sur un certain nombre d'éléments (des musiques dans notre cas). Ce nombre peut être modifié par l'attribut *batch_size* (un « batch » étant un groupe d'éléments). Plus le nombre d'éléments pris en compte est grand, plus l'approximation sera bonne. Cependant, l'utilisation d'un nombre important d'éléments utilisera

beaucoup plus de mémoire et allongera le temps de calcul. Un compromis doit donc être trouvé par l'expérience.

```
# Entrainement du modèle
# sur 10 itérations avec une mise à jour des poids à l'aide de 5 éléments
model.fit(data, labels, epochs=10, batch_size=5)
```

Prévision

La prévision est effectuée en exécutant la méthode *predict* sur le modèle. Les données sont données en paramètre sous la forme d'un tableau suivant la syntaxe *Numpy*.

```
# Calculer une prévision avec un vecteur X en entrée
previsions = model.predict(X)
```

Sauvegarder le modèle

Afin de ne pas devoir effectuer l'entrainement du réseau lors de chaque démarrage du programme, il est possible de sauvegarder un modèle dans un fichier. Il est aussi possible de reprendre l'entrainement au point où la dernière exécution s'était arrêtée.

Ainsi, l'architecture du réseau, ses poids, sa configuration (fonction de coût, d'optimisation) et son dernier état d'avancement son sauvegardés dans le fichier.

On utilise la fonction suivante :

```
from keras.models import load_model

# instanciation du modèle, compilation et entrainement
# ...

# Enregistrement du moèdle dans un fichier
model.save('fichier_du_modele.h5')

# Chargement du modèle stocké dans le fichier
model = load_model('fichier_du_modele.h5')
```

Keras → doc Keras ; modèles ; fonctions d'activation ; Optimizer ; Fonctions de coût ;

Tuto Keras

Theano

TensorFlow