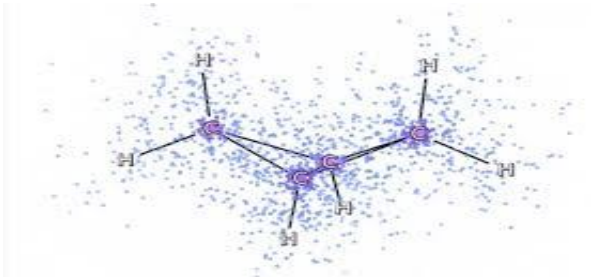
Chapitre_1

Rapport TP3:

MLP sous keras

Réalisé par:
CHARAFI Asma
BOUJIDA Hafssa
EL-ASRI Noussaiba
AIT JILALI Nouhaila

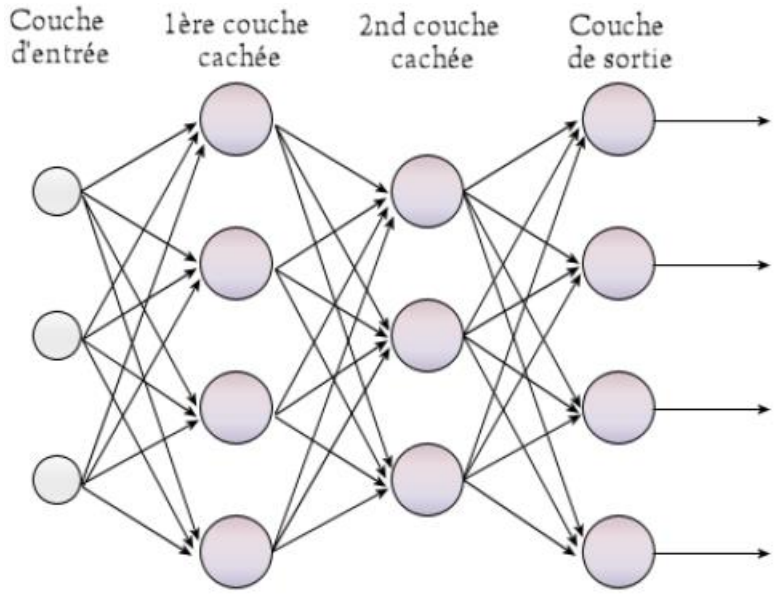
Encadré par:
M. IBN ELHAJ



Sommaire :

Introduction

Le **perceptron multicouche** *(multi layer perceptron* MLP) est un type de réseau neuronal artificiel organisé en plusieurs couches (au moins trois, une couche d'entrée, l'autre cachée et une de sortie) au sein desquelles une information circule de la couche d'entrée vers la couche de sortie uniquement ; il s'agit donc d'un réseau à propagation directe (*feedforward*). Chaque couche est constituée d'un nombre variable de neurones, les neurones de la dernière couche (dite « de sortie ») étant les sorties du système global.



Deux principales techniques utilisées par MLP sont :

- Classification
- Régression

Keras

• Bibliothèque de réseaux de neurones profonds en Python – API de réseaux neuronaux de haut niveau – Modulaire - Le modèle de construction consiste simplement à empiler des couches et à connecter des graphiques de calcul. – Fonctionne sur TensorFlow ou Theano ou CNTK.

• **Pourquoi utiliser Keras?**

- Utile pour le prototypage rapide, en ignorant les détails de la mise en œuvre du backprop ou de la procédure d'optimisation de l'écriture
- Prend en charge la convolution, la couche récurrente et la combinaison des deux.
- Fonctionne de manière transparente sur CPU et GPU
- Presque toutes les architectures peuvent être conçues en utilisant ce Framework.
- Code Open Source - Large support communautaire.

Première partie A

2.1 Le code du MLP:

En utilisant Spyder, le code utilisé est le suivant :

```
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout

#Generate dummy data
x_train = np.random.random((1000, 20))
y_train = np.random.randint(2, size=(1000, 1))
x_test = np.random.random((100, 20))
y_test = np.random.randint(2, size=(100, 1))


model = Sequential()
model.add(Dense(64, input_dim=20, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',
optimizer='rmsprop', metrics=['accuracy'])
model.fit(x_train, y_train, epochs=20, batch_size=128)


score = model.evaluate(x_test, y_test, batch_size=128)
```

Après exécution, le résultat est le suivant :

```
Epoch 1/20
8/8 [==============================] - 1s 0s/step - loss: 0.7427 - accuracy: 0.5031
Epoch 2/20
8/8 [==============================] - 0s 2ms/step - loss: 0.7274 - accuracy: 0.4710
Epoch 3/20
8/8 [==============================] - 0s 2ms/step - loss: 0.7207 - accuracy: 0.4840
Epoch 4/20
8/8 [==============================] - 0s 2ms/step - loss: 0.7136 - accuracy: 0.5012
Epoch 5/20
8/8 [==============================] - 0s 2ms/step - loss: 0.7138 - accuracy: 0.4786
Epoch 6/20
8/8 [==============================] - 0s 2ms/step - loss: 0.6976 - accuracy: 0.5351
Epoch 7/20
8/8 [==============================] - 0s 2ms/step - loss: 0.7030 - accuracy: 0.4922
Epoch 8/20
8/8 [==============================] - 0s 2ms/step - loss: 0.6980 - accuracy: 0.5218
Epoch 9/20
8/8 [==============================] - 0s 2ms/step - loss: 0.6921 - accuracy: 0.5062
Epoch 10/20
8/8 [==============================] - 0s 2ms/step - loss: 0.6901 - accuracy: 0.5271
Epoch 11/20
8/8 [==============================] - 0s 0s/step - loss: 0.6951 - accuracy: 0.4986
Epoch 12/20
8/8 [==============================] - 0s 0s/step - loss: 0.6949 - accuracy: 0.5281
Epoch 13/20
8/8 [==============================] - 0s 2ms/step - loss: 0.6937 - accuracy: 0.5117
Epoch 14/20
8/8 [==============================] - 0s 2ms/step - loss: 0.6944 - accuracy: 0.4986
Epoch 15/20
8/8 [==============================] - 0s 2ms/step - loss: 0.6903 - accuracy: 0.5336
Epoch 16/20
8/8 [==============================] - 0s 0s/step - loss: 0.6927 - accuracy: 0.5026
Epoch 17/20
8/8 [==============================] - 0s 0s/step - loss: 0.6920 - accuracy: 0.5431
Epoch 18/20
8/8 [==============================] - 0s 0s/step - loss: 0.6956 - accuracy: 0.5082
Epoch 19/20
8/8 [==============================] - 0s 2ms/step - loss: 0.6866 - accuracy: 0.5371
Epoch 20/20
8/8 [==============================] - 0s 2ms/step - loss: 0.6878 - accuracy: 0.5549
Out[14]: <tensorflow.python.keras.callbacks.History at 0x219f3b5cbe0>

In [16]: score = model.evaluate(x_test, y_test, batch_size=128)
1/1 [==============================] - 0s 13ms/step - loss: 0.7036 - accuracy: 0.4900
```

2.2 Réponses aux questions :

2.2.1 Le rôle de « model.add(Dropout(..)) » :

La méthode du dropout est utilisée pour "éteindre" les neurones aléatoirement (avec une probabilité prédéfinie, souvent un neurone sur deux) ainsi que les neurones périphériques. Ainsi, avec moins de neurones, le réseau est plus réactif et peut donc apprendre plus rapidement.
Cette technique a montré non seulement un gain dans la vitesse d'apprentissage, mais en déconnectant les neurones, on a aussi limité des effets marginaux, rendant le réseau plus robuste et capable de mieux généraliser les concepts appris.

2.2.2 Variation du nombre d'epochs et interprétation des résultats :

Comme cité dans le code par la ligne suivante :

model.fit(x_train, y_train, epochs=20, batch_size=128)

La valeur d'epochs pour notre algorithme d'apprentissage MLP pour une classification binaire est fixée à 20.

Variation « epochs » :

Pour epochs=200 :

```
model.compile(loss='binary_crossentropy',
optimizer='rmsprop', metrics=['accuracy'])
model.fit(x_train, y_train, epochs=200, batch_size=128)
```

Pour epochs=300 :

```
model.compile(loss='binary_crossentropy',
optimizer='rmsprop', metrics=['accuracy'])
model.fit(x_train, y_train, epochs=300, batch_size=128)
```

Pour epochs=500 :

```
model.compile(loss='binary_crossentropy',
optimizer='rmsprop', metrics=['accuracy'])
model.fit(x_train, y_train, epochs=500, batch_size=128)
```

Après exécuter ces différentes valeurs d'epochs, on remarque qu'à chaque fois d'epochs augmente, la valeur de la fonction de perte noté « loss » diminue, par contre la valeur de « accuracy » est relativement constante.

2.2.3 Changement d'optimizer et interprétation des résultats

Dans le code initial, le choix d'optimizer était rmsprop :

```
model.compile(loss='binary_crossentropy',
optimizer='rmsprop', metrics=['accuracy'])
model.fit(x_train, y_train, epochs=20, batch_size=128)
```

« Rmsprop » : L'essentiel de RMSprop est de:

- Maintenir une moyenne mobile (actualisée) du carré des dégradés.
- Divisez le gradient par la racine de cette moyenne.

Généralement, les optimisateurs disponibles dans keras sont :

- SGD - Descente de gradient stochastique
- SGD avec momentum
- Adam
- AdaGrad
- RMSprop
- AdaDelta

On change l'optimizer et le remplacé par: « AdaDelta »

L'optimisation AdaDelta est une méthode de descente de gradient stochastique basée sur un taux d'apprentissage adaptatif par dimension pour remédier à deux inconvénients:

- La dégradation continue des taux d'apprentissage tout au long de la formation.
- La nécessité d'un taux d'apprentissage global sélectionné manuellement.

Deuxième partie B

3.1 Le code et explication

```
from pandas import read_csv
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
import numpy as np
```

Nous utilisons l'API séquentielle et le type de couche densément connecté pour créer la structure particulière du MLP. L'utilisation du Numpy pour importer nos données.

```
# load dataset
dataframe = read_csv("housing.csv")
dataset = dataframe.values
```

L'importation du dataset par l'utilisation du modèle pandas et la convertir en une matrice.

```
# split into input (X) and output (Y) variables
X = dataset[:,0:-1]
Y = dataset[:,-1]
```

La division les données en vecteurs de caractéristiques et cibles. L'hypothèse est que le prix d'une maison peut être prédit à partir des trois autres. Plus précisément, l'utilisation les huit premières colonnes (0: 7, alias zéro à sept mais excluant huit) de l'ensemble de données comme variables prédictives, tandis que l'utilisation de la neuvième (colonne 8) comme variable prédite.

```
# define base model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(16, input_dim=input_shape, kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal'))

    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
```

Nous utilisons l'API séquentielle Keras car elle nous facilite la vie compte tenu de la simplicité de notre modèle.

Nous spécifions ensuite huit couches de neurones densément connectées: une avec 16 sorties et une avec 1 sortie. De cette façon, le réseau neuronal sera autorisé à «penser» d'abord plus large, avant de converger vers la prédiction réelle.

La couche d'entrée est spécifiée par la forme d'entrée et contient donc 8 neurones; un par fonction d'entrée.

Notez que nous utilisons l'activation basée sur ReLU car il s'agit de l'une des fonctions d'activation standard utilisées aujourd'hui.

Le code complet :

```
from pandas import read_csv
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
import numpy as np

# load dataset
dataframe = read_csv("housing.csv")
del dataframe["ocean_proximity"]
dataset = dataframe.values

# split into input (X) and output (Y) variables
X = dataset[:,0:-1]
Y = dataset[:,-1]

# define base model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(16, input_dim=8, kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal'))

    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

# evaluate model
estimator = KerasRegressor(build_fn=baseline_model, epochs=100, batch_size=5, verbose=0)
kfold = KFold(n_splits=10)
results = cross_val_score(estimator, X, Y, cv=kfold)

print("Baseline: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

Le résultat est :

```
Baseline: -32.65 (23.33) MSE
```

L'erreur quadratique moyenne est négative car scikit-learn s'inverse de sorte que la métrique est maximisée au lieu d'être minimisée.

De même pour la dataset concernant le réservoir de l'eau d'après le site, sous code au dessous, on remarque la même chose après exécution du programme sue la fonction de perte augmente après chaque itération telque notre perte de validation semble être de l'ordre de 290-320. C'est relativement mauvais; nous sommes au large de quelques centaines de millions de pieds carrés d'eau.

```
Epoch 1/10
4517/4517 [==============================] - 14s 3ms/step - loss: 332.6803 - mean_squ
Epoch 2/10
4517/4517 [==============================] - 13s 3ms/step - loss: 276.1181 - mean_squ
Epoch 3/10
4517/4517 [==============================] - 13s 3ms/step - loss: 274.3100 - mean_squ
Epoch 4/10
4517/4517 [==============================] - 14s 3ms/step - loss: 273.0496 - mean_squ
Epoch 5/10
4517/4517 [==============================] - 14s 3ms/step - loss: 273.0190 - mean_squ
Epoch 6/10
4517/4517 [==============================] - 14s 3ms/step - loss: 272.5061 - mean_squ
Epoch 7/10
4517/4517 [==============================] - 15s 3ms/step - loss: 271.1735 - mean_squ
Epoch 8/10
4517/4517 [==============================] - 15s 3ms/step - loss: 270.2527 - mean_squ
```

```python
from pandas import read_csv
from keras.models import Sequential
from keras.layers import Dense
from keras.wrappers.scikit_learn import KerasRegressor
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
import numpy as np

# load dataset
dataframe = read_csv("housing.csv")
del dataframe["ocean_proximity"]
dataset = dataframe.values

# split into input (X) and output (Y) variables
X = dataset[:,0:-1]
Y = dataset[:,-1]

# define base model
def baseline_model():
    # create model
    model = Sequential()
    model.add(Dense(16, input_dim=8, kernel_initializer='normal', activation='relu'))
    model.add(Dense(1, kernel_initializer='normal'))

    # Compile model
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

# evaluate model
estimator = KerasRegressor(build_fn=baseline_model, epochs=100, batch_size=5, verbose=0)
kfold = KFold(n_splits=10)
results = cross_val_score(estimator, X, Y, cv=kfold)

print("Baseline: %.2f (%.2f) MSE" % (results.mean(), results.std()))
```

3.2 Conclusion

L'objectif de ce TP est de savoir les deux techniques d'apprentissage régression et classification pour un réseau MLP sous keras, cela pour nous aider à traiter tous type de problème sous Deep Learning, on se basant à minimiser la fonction perte et la moyenne quadratique.