TP2 : DEEP Learning

Ce Tp permet d'apprendre à développer un premier modèle de classification sur le datase, en utilisant l'API Keras.

TensorFlow

C’est une bibliothèque open-source d'apprentissage automatique développée par Google. Elle est largement utilisée pour la création, l'entraînement et le déploiement de modèles de réseaux de neurones artificiels. TensorFlow fournit un ensemble d'outils et de bibliothèques pour les tâches liées à l'apprentissage automatique, en particulier pour le deep learning.

TensorFlow est utilisé dans une grande variété d'applications d'apprentissage automatique, y compris la vision par ordinateur, le traitement du langage naturel, la reconnaissance vocale, la recommandation, la détection d'anomalies, etc. Il est largement adopté par l'industrie et la recherche pour ses performances, sa flexibilité et sa robustesse.

Keras

Keras est l’une des bibliothèques Python les plus puissantes et les plus faciles à utiliser pour les modèles d’apprentissage profond et qui permet l’utilisation des réseaux de neurones de manière simple. Keras englobe les bibliothèques de calcul numérique Theano et TensorFlow. Pour définir un modèle de deep learning, on définit les caractéristiques suivantes :

- Nombre de couches ;
- Types des couches ;
- Nombre de neurones dans chaque couche;
- Fonctions d’activation de chaque couche ;
- Taille des entrées et des sorties.

Le type de couches qu’on va utiliser est le type dense. Les couches denses sont les plus populaires car il s’agit d’une couche de réseau neuronal ordinaire où chacun de ses neurones est connecté aux neurones de la couche précédente et de la couche suivante.

Comme mentionné précédemment, il existe plusieurs types de fonctions d’activation, chacune d’entre elles relie l’entrée et le poids du neurone d’une manière différente ce qui influe sur le comportement du réseau. Les fonctions les plus utilisées sont la fonction Unité Linéaire Rectifiée (ReLU en anglais, Rectified Linear Unit), la fonction Sigmoïde et la fonction linéaire.

Pour ce qui est du nombre de neurones dans la couche d’entrée il est le même que le nombre de caractéristiques de l’ensemble de données. Dans la suite de ce tutoriel, nous allons essayer de construire un modèle d’apprentissage profond.

Exercice 1

```
import numpy as np
import matplotlib.pyplot as plt
from tensorflow import keras
```

1. Chargement des données et Normalisation

```
# Chargement des données MNIST
    (X_train, y_train) , (X_test, y_test) =
    keras.datasets.mnist.load_data()

    print('trainset:', X_train.shape) # 60,000 images
    print('testset:', X_test.shape) # 10,000 images

    # Normalisation des données
    X_train = X_train / 255
    X_test = X_test / 255
```

2. Visualisation des données

```
# visualisation de quelques images
fig, ax = plt.subplots(nrows=1, ncols=10, figsize=(20, 4))
for i in range(10):
  ax[i].imshow(X_train[i], cmap='gray')

plt.tight_layout()
plt.show()
```

3. Configuration des Couches du Réseau de Neurones

```
# Configuration des couches du réseau
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28, 28)),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(128, activation='relu'),
    keras.layers.Dense(10)
model.summary()
```

4. Entrainement du Réseau de Neurones

```
# Compilation du modele
model.compile(optimizer='adam',
              loss=
keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

```
# Entrainement du modele
model.fit(X_train, y_train, epochs=10)
```

5. Évaluation du réseau de neurone sur les données de Test

```
# Evaluation du modele
test_loss, test_acc = model.evaluate(X_test,  y_test)
print('Test accuracy:', test_acc)
```

6. Création d'un modele prédictif

```
# modele prédictif (softmax)
prediction_model = keras.Sequential([model, keras.layers.Softmax()])
predict_proba = prediction_model.predict(X_test)
predictions = np.argmax(predict_proba, axis=1)

print(predictions[:10])
print(y_test[:10])
```

Exercice 2

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs
```

1. Dataset

```
X, y = make_blobs(n_samples=100, n_features=2, centers=2,
random_state=0)
y = y.reshape((y.shape[0], 1))

print('dimensions de X:', X.shape)
print('dimensions de y:', y.shape)

plt.scatter(X[:,0], X[:, 1], c=y, cmap='summer')
plt.show()
```

2. Fonctions du modele

```
def initialisation(X):
    W = np.random.randn(X.shape[1], 1)
    b = np.random.randn(1)
    return (W, b)
```

```python
def model(X, W, b):
    Z = X.dot(W) + b
    A = 1 / (1 + np.exp(-Z))
    return A
```

```python
def log_loss(A, y):
    return 1 / len(y) * np.sum(-y * np.log(A) - (1 - y) * np.log(1 -
A))
```

```python
def gradients(A, X, y):
    dW = 1 / len(y) * np.dot(X.T, A - y)
    db = 1 / len(y) * np.sum(A - y)
    return (dW, db)

def update(dW, db, W, b, learning_rate):
    W = W - learning_rate * dW
    b = b - learning_rate * db
    return (W, b)
```

```python
def predict(X, W, b):
    A = model(X, W, b)
    # print(A)
    return A >= 0.5
```

```python
from sklearn.metrics import accuracy_score
```

```python
def artificial_neuron(X, y, learning_rate = 0.1, n_iter = 100):
    # initialisation W, b
    W, b = initialisation(X)

    Loss = []

    for i in range(n_iter):
        A = model(X, W, b)
        Loss.append(log_loss(A, y))
        dW, db = gradients(A, X, y)
        W, b = update(dW, db, W, b, learning_rate)

    y_pred = predict(X, W, b)
    print(accuracy_score(y, y_pred))

    plt.plot(Loss)
    plt.show()
```

```
    return (W, b)
```

```
W, b = artificial_neuron(X, y)
```

3. Frontiere de décision

```
fig, ax = plt.subplots(figsize=(9, 6))
ax.scatter(X[:,0], X[:, 1], c=y, cmap='summer')

x1 = np.linspace(-1, 4, 100)
x2 = ( - W[0] * x1 - b) / W[1]

ax.plot(x1, x2, c='orange', lw=3)
```

Exercice 3

Prétraitement des données

Tout d'abord, nous devons nettoyer l'ensemble des données que nous utiliserons pour le modèle d'apprentissage.

- On importe les bibliothèques nécessaires, à savoir : matplotlib, numpy, pandas, seaborn et tensorflow.
- On importe ensuite l'ensemble des données sur lequel on va travailler sous forme de data-frame.
- On affiche dans la partie du code suivante le nombre de valeurs manquantes pour chaque colonne de notre ensemble de données ainsi que le pourcentage de celles-ci.

```
#Valeurs manquantes
VM = pd.DataFrame({
'Colonne': df.columns.values,
'nbr de VM': df.isna().sum().values,
'% de VM': 100 * df.isna().sum().values / len(df),})
VM = VM[VM['nbr de VM'] > 0]
print(VM.sort_values(by='nbr de VM',
ascending=False).reset_index(drop=True))
```

- On supprime les colonnes contenant un pourcentage élevé de valeurs manquantes.
- On affiche les variables catégorielles et les variables numériques de la dataset afin de pouvoir traiter les valeurs manquantes qui restent dans la dataset.

```
- #on affiche les variables catégorielles et les variables
  numériques
  colonnes_avec_VM = df.columns[df.isna().sum() > 0]
  for col in colonnes_avec_VM :
   print(col)
```

```
print(df[col].unique()[:5])
print('*'*30)
```

On remplace ensuite les valeurs manquantes numériques d'une colonne par la moyenne de celles-ci.

```
#on remplace les VM numériques par la moyenne de la colonne
num_VM = ['Lot Frontage', 'Mas Vnr Area', 'BsmtFin SF 1', 'BsmtFin SF
2',
'Bsmt Unf SF', 'Total Bsmt SF', 'Bsmt Full Bath',
'Bsmt Half Bath', 'Garage Yr Blt', 'Garage Cars', 'Garage Area']
for n_col in num_VM:
 df[n_col] = df[n_col].fillna(df[n_col].mean())
```

- On remplace également les valeurs manquantes catégorielles d'une colonne par le mode de celles-ci.

```
#On remplace les VM nominales par le mode variable
nom_VM = [x for x in colonnes_avec_VM if x not in num_VM]
for nom_col in nom_VM:
  df[nom_col] =    df[nom_col].fillna(df[nom_col].mode().to_numpy()[0])
```

Pour appliquer un modèle de deep learning, l'ensemble de données ne doit contenir que des variables numériques. Alors, on va encoder les variables catégorielles. Dans un premier temps, on affiche le type de données de chaque colonne de la dataset.

```
#encodage des variables catégorielles
#on affiche tout d'abord les types de données pour chaque colonne
types = pd.DataFrame({
'Colonne': df.select_dtypes(exclude='object').columns.values,
'Type': df.select_dtypes(exclude='object').dtypes.values})
print(types)
```

Même si Pandas ne considère pas les données de la colonne MS SubClass étant nominaux, le descriptif de la dataset en dit autrement. Il arrive des fois que Pandas retourne incorrectement le type d'une colonne.

Alors, on va sélectionner des variables qui représenteront notre dataset (pour ne pas avoir beaucoup de colonnes après l'encodage des valeurs catégorielles ce qui pourrait impliquer le sur-apprentissage du modèle) puis on transforme ces données catégorielles en utilisant la fonction ***get_dummies()*** de la bibliothèque pandas.

```
#encodage des variables catégorielles,
#selon le descriptif de la dataset la colonne MS SubClass est nominale,
#Pandas a du mal à retourner son vrai type
df['MS SubClass'] = df['MS SubClass'].astype(str)
```

```
selection_de_variables = ['MS SubClass', 'MS Zoning', 'Lot Frontage',
'Lot Area',
             'Neighborhood', 'Overall Qual', 'Overall Cond',
             'Year Built', 'Total Bsmt SF', '1st Flr SF', '2nd Flr SF',
             'Gr Liv Area', 'Full Bath', 'Half Bath', 'Bedroom AbvGr',
             'Kitchen AbvGr', 'TotRms AbvGrd', 'Garage Area',
             'Pool Area', 'SalePrice']
df = df[selection_de_variables]
#la dataset comprendera maintenant que 67 variables
df = pd.get_dummies(df)
```

Maintenant on fractionne l'ensemble de données en des données d'entrainement et des données de test.

```
#fractionner dataset en des données de test et de train
train = df.sample(frac = 0.8, random_state = 9)
test = df.drop(train.index)
#variable cible
train_cible = train.pop('SalePrice')
test_cible = test.pop('SalePrice')
```

On standardise ensuite les variables afin de les avoir sur la même échelle.

```
#Standardisation
variables_pred = train.columns
for col in variables_pred:
    col_moyenne = train[col].mean()
    col_ecart_type = train[col].std()
    if col_ecart_type == 0:
       col_ecart_type = 1e-20
    train[col] = (train[col] - col_moyenne) / col_ecart_type
    test[col] = (test[col] - col_moyenne) / col_ecart_type
```

Construction du modèle d'apprentissage profond

```
modele = keras.Sequential([
    layers.Dense(64, activation = 'relu', input_shape =
[train.shape[1]]),
    layers.Dropout(0.3, seed = 2),
    layers.Dense(64, activation = 'swish'),
    layers.Dense(64, activation = 'relu'),
    layers.Dense(64, activation = 'swish'),
    layers.Dense(64, activation = 'relu'),
    layers.Dense(64, activation = 'swish'),
    layers.Dense(1)])
```

On crée une instance du ***Sequential()*** de la bibliothèque Keras, celle-ci superpose un ensemble de couches pour en créer un seul modèle. On lui passe en paramètre une liste de couches qu'on souhaite utiliser pour notre modèle.
Comme vous allez le remarquer dans ce modèle, on a créé plusieurs couches denses et une seule couche de type Dropout. La première couche est la couche d'entrée, son nombre de neurones est égal au nombre de caractéristiques de l'ensemble de données.

Dans chaque couche on retrouve 64 neurones, ce nombre est le résultat optimal de plusieurs tests. En effet, 64 neurones par couches pour l'exemple de cet ensemble de données donnent un résultat assez précis.

Remarque :

Il est recommandé d'essayer plusieurs chiffres jusqu'à l'obtention de résultats précis.

Pour la couche Dropout, on a diminué de 30% le nombre des données d'entrée afin d'éviter le phénomène du ***overfitting***. La graine prend une valeur de 2 pour avoir des résultats plus reproductibles.
Finalement, la dernière couche dense avec un seul neurone est la couche de sortie. Par défaut, elle prend la fonction d'activation linéaire.

Compilation du modèle

L'étape suivante est la compilation du modèle. Pour ce faire, il faut choisir :

- La fonction de perte : utilisée pour diminuer l'erreur. Plus l'erreur est faible plus le modèle est précis ;
- L'optimiseur : aide à obtenir de meilleurs résultats pour la fonction de perte ;
- Métriques : utilisées pour évaluer le modèle.

Dans le code suivant nous avons utilisé la fonction de perte comme étant l'erreur quadratique moyenne avec l'optimiseur ***RMSprop*** qu'on lui donne un taux d'apprentissage de 0,001. Et pour la métrique nous avons utilisé l'erreur absolue moyenne qui va avec la fonction de perte.

```
optimiseur = tf.keras.optimizers.RMSprop(learning_rate = 0.001)
modele.compile(loss = tf.keras.losses.MeanSquaredError(),optimizer =
optimiseur,metrics = ['mae'])
```

Entrainement du modèle

Dans ce qui suit, on entraine le modèle sur le jeu de données ***train.*** Celui-ci passe 70 fois sur l'ensemble de données pour en tirer le meilleur résultat possible. Ensuite, 20% des données d'entrainement sont utilisées pour la validation.
La fonction ***fit()*** affiche la valeur de la fonction de perte et la métrique pour chaque époque.

```
training = modele.fit (train, train_cible, epochs = 70,
validation_split = 0.2)
```

Prévision

Avant de commencer d'utiliser le modèle pour nos prévisions, nous allons illustrer comment la fonction de perte et la **mae** vont changer. Remarquons que ces valeurs ont diminué au fil du temps, alors le modèle est devenu plus précis.

```
historique = pd.DataFrame(training.history)
historique['epoque'] = training.epoch

figure, axe = plt.subplots(figsize = (14,8))
num_epoque = historique.shape[0]
axe.plot(np.arange(0, num_epoque), historique["mae"],label = "Training
MAE", lw = 3, color = 'red')
axe.plot(np.arange(0, num_epoque), historique["val_mae"],label =
"Validation MAE", lw = 3, color = 'blue')
axe.legend ()
plt.tight_layout()
plt.show()
```

On prend une unité des valeurs de test pour avoir sa prédiction en utilisant notre modèle. Ensuite, on affiche la valeur de la prédiction et la valeur actuelle pour les comparer.

```
test1 = test.iloc[[10]]
test_prediction = modele.predict(test1).squeeze()
test_label = test_cible.iloc[10]
print("Prédiction du modèle = {:.2f}".format(test_prediction))
print("Valeur actuelle = {:.2f}".format(test_label))
```

Travail à réaliser

1. Reprenez la même dataset utilisée dans l'exercice 3.
2. Créez un modèle d'apprentissage profond qui utilise l'optimiseur 'Adam'.
3. Modifiez dans les étapes précédemment suivies pour obtenir de meilleurs résultats.