Prédiction de séries temporelle avec Python

Dans ce notebook, nous allons essayer de préduire le cours du bitcoin minute par minute en utilisant un réseau de neurone. Le réseau choisi est un LSTM.

Import Général

In [ ]:
```
import torch
import torch.nn as nn
import datetime
from datetime import date
import numpy as np
import math
import matplotlib.pyplot as plt
import os
%matplotlib inline
```

Les données

Etape 0 : la collecte des données

Comme nous partons de rien, nous devons d'abord collecter les données. Pour cela, il existe plusieurs solutions. Avant de pouvoir collecter les données, il faut savoir où les trouver.

En faisant une recherche sur google, on trouve beaucoup de site web proposant le cours du bitcoin. Pour ne citer que les plus connues : https://fr.finance.yahoo.com/ ou https://www.google.com/finance/?hl=fr

Une fois qu'on sait où trouver les données, il faut l'extraire. Pour cela, nous pouvons utiliser le web scrapping. Il s'agit de la première approche que j'ai choisi d'utiliser. Elle comporte son lot d'avantage et d'inconvénient. Vous retrouverez les tableaux récapitulatifs ci-dessous.

Selenium

Selenium est une librairie développée sur plusieurs langage et disponible sur python. Cette librairie permet de réaliser toutes les interactions faisables sur un navigateur web. Une fois qu'on a bien configuré Selenium et qu'on a pris ses marques avec il est relativement facile à utiliser. On peut facilement adapter son script à un autre site web. Il suffit juste de connaître les balises html adéquat. Voici la documentation de la librairie : https://selenium-python.readthedocs.io/.

Ainsi, le script mis en place actualisait toutes les minutes la page web et récupérer la valeur du cours affichée sur le site. Puis on enregistrait ces valeurs dans un fichier csv.

Le problème de cette solution est qu'elle nécessite une bonne bande passante. Car à chaque minute, nous actualisons la page du site. Si cette page est volumineuse et si votre réseau internet a de la latence alors vous n'aurez pas le cours minutes par minutes. C'est ce qu'il s'est passé lors de la mise en place de cette solution. Il me manquait des valeurs pour certaines minutes. Ce qui peut fortement impacter notre modèle, car il nous faut des données continues dans le temps. Une solution pour remédier à ce problème

aurait été de remplacer les valeurs manquantes par des valeurs fictives qu'on doit créer. On peut faire se procéder avec des interpolations ou en faisant la moyenne des valeurs qui précédent et suivent notre valeur manquante. C'est pourquoi, l'autre solution a été privilégiée.

API REST

Une autre solution possible pour récupérer les données consiste à utiliser les requêtes API REST envoyées lors de la navigation sur le site. Cette méthode a l'avantage d'être plus fiable, car nous ne rechangeons pas toute la page, mais nous demandons seulement les données qui nous intéressent. Ainsi une fois que nous avons identifié la bonne requête à envoyer (ceci peut se faire via l'onglet network de la touche F12), nous pouvons facilement enregistrer les données via un script et automatiser ce script sur un serveur via CRON par exemple. C'est ce qui est mis en place actuellement.

Ainsi, avec cette solution, la requête API nous permet de récupérer la valeur du bitcoin minute par minute et ceux sur les 5 derniers jours. Donc, nous lançons le script tous les 5 jours puis nous fusionnons les données avec celle déjà enregistrées. L'avantage, c'est que nos données sont continues, car c'est le site web qui nous les envoie.

Le problème avec cette solution, est qu'il est assez difficile d'identifier la bonne requête API sans aucune documentation du site. Il faut alors décortiquer les requêtes du site et trouver les bons paramètres, etc. De plus, cette méthode à l'inconvénient de ne pas s'adapter à d'autres sites. Si nous voulons récolter des données sur un autre site alors il faut tout réanalyser les requêtes de ce dernier.

Selenium		API REST	
Avantages	Inconvénients	Avantages	Inconvénients
1. Facile à déployer sur plusieurs sites 2. Faire plusieurs actions de manières simple	1. Configuration peut être un peu laborieuse avec les pilotes à installer 2. Nécéssite une bonne bande passante pour recharger une page web régulièrement	1. Très fiable, permet de récupérer des données sans trop de discontinuité 2. facile à automatiser avec une tâche CRON	1. Ne fonctionne plus si votre site change ses requêtes 2. Analyser le traffic réseau lors de la navigation sur le site pour trouver la bonne REST

Les données sont les valeurs du bitcoin avec différentes informations relatives au cours. Comme la valeur à la clôture ou à la fermeture par exemple. Nous devons traiter les données pour pouvoir les exploiter.

Pour cela, il existe une bibliothèque sur python : *pandas*. Pandas nous permet de générer des données et de réaliser du traitement dessus assez facilement.

Vous retrouverez la documentation de Pandas sur le lien suivant : https://pandas.pydata.org/docs/.

Etape 1 : Charger les données

Nous chargeons les données stockées dans un fichier csv dans un objet DataFrame de *Pandas*. On précise que nos données sont séparées par un ';' et que la colonne correspondant à l'index est la colonne 'time'.

In [ ]:
```
import pandas as pd

dataFromExcel = pd.read_csv('bitcoin.csv',sep=';',index_col='time')
```

Etape 2 : nettoyer les données

Cette étape est très importante. Elle permet de traiter les données manquantes ou les données erronées. Cela peut être des données avec des informations manquantes par exemple. Dans notre cas, nous n'avons aucune donnée manquante. Toutes les données que nous avons sont continues dans le temps.

Etape 3 : traiter les données

Maintenant, nous devons traiter les données. C'est-à-dire que nous avons des données qui sont différentes entre elles. Par exemple, nous pouvons imaginer que nous avons des données avec comme première caractéristique un poids en Tonne et nous pouvons imaginer une autre caractéristique avec un poids en g. L'écart d'échelle est très grand et la comparaison des deux caractéristiques n'est pas possible. L'étude des corrélations devient alors plus compliquée à réaliser avec des plages de variation très grande.

Ainsi, nous normalisons les données pour pouvoir mieux les utiliser. Normaliser les données veut dire que la plage de variation de celle-ci se situe entre 0 et 1. L'autre avantage de normaliser les données, c'est que cela demande moins de ressource aux algorithmes pour faire les calcules.

Ici, nous utilisons la normalisation standard qui consiste à soustraire la moyenne et à la diviser par l'écart-type soit X l'ensemble de nos données, on normalise X par $|X| = \frac{X-\mu}{\sigma}$. Ou μ représente la moyenne et σ représente l'écart-type. Dans ce cas, chaque valeur refléterait la distance par rapport à la moyenne en unités d'écart-type. Si nous supposons que toutes les variables proviennent d'une distribution normale, la normalisation les rapprocherait toutes de la distribution normale standard. La distribution résultante a une moyenne de 0 et un écart-type de 1.

D'autres normes existent comme la normalisation de moyenne (normalisation min max), dont la distribution résultante sera entre -1 et 1 avec une moyenne = 0. Ou les normes L_p et encore bien d'autres, cette notion est découle de l'algèbre cf https://fr.wikipedia.org/wiki/Norme_(math%C3%A9matiques)

In [ ]:
```python
# Fonction qui traite le DataFrame, il est subit la normalisation standard
# Input :
#     - un DataFrame qu'on souhaite normaliser
# Output :
#     - le DataFrame normaliser
#     - la moyenne (ou les moyennes) du DataFrame
#     - l'écart-type (ou les écarts-types) du DataFrame
def prepross_data(data):
    #enlève les valeurs vides
    data.dropna(axis=0,inplace=True)
    mean = data.mean()
    std = data.std()
    data =(data-mean)/std
    return mean,std,data
```

Maintenant que nous avons normé nos données, il faut formaliser une structure de donnée pour pouvoir les utiliser avec nos algorithmes. Il faut pour cela choisir un nombre de caractéristiques qu'on peut extraire de nos données. Cela peut-être les valeurs brutes ou on peut réaliser préalablement un traitement sur les données tel qu'une combinaison linéaire des données brutes.

Ainsi construisons un vecteur $\overrightarrow{x}$ qui sera passé à notre modèle pour l'apprentissage mais aussi pour la prédiction.

Ce vecteur est composé de caractéristiques (features). Il nous reste à définir ce nombre de caractéristiques notons le $N_{features}$.

Donc $\overrightarrow{x} \in \mathbb{R}^{N_{features}}$.

Ici les features correspondent à la valeur du bitcoin à la clôture à chaque minute. Nous choisissons donc un vecteur qui contient les valeurs du bitcoin à la clôture sur $N_{features}$ minutes consécutives.

Une autre possibilité est de passer les valeurs de la clôture, l'ouverture et d'autre valeur qu'on connait ou qu'on a calculé chaque minutes.

Au lieu de passer un seul vecteur à notre modèle pour qu'il puisse apprendre. Il existe un mode appelé le mode batch, qui permet de passer plusieurs vecteurs à notre modèle ce qui lui permet d'apprendre sur plusieurs vecteurs. Cela permet d'éviter certains biais introduits par un vecteur car notre modèle moyenne l'apprentissage sur l'ensemble des vecteurs.

Maintenant, il faut labelliser les données pour pouvoir indiquer à notre modèle quelle est valeur de terrain par rapport à celle qui a prédit. Concrètement, imaginons que nous ayons les valeurs du bitcoin sur 6 minutes, les 5 premières minutes correspondent aux caractéristiques (features) et forment un « vecteur de features » et la $6^{ème}$ minute correspond au label associé à ce vecteur. Ainsi, il pourra calculer son erreur entre la valeur prédite et la vraie valeur.

In [ ]:
```
#Constante

numFeature = 5
```

In [ ]:
```
# Fonction qui créer un vecteur x à partir d'un index et d'un DataFrame
# Input :
#     - l'index sur la première valeur du vecteur
#     - un DataFrame contenant toutes les valeurs
# Output : un vecteur x, contenant les valeur de cloture du bitcoin des N prochaines min
def create_input(index,pdData):
    return pdData["cloture"].iloc[index:index+numFeature+1].values.tolist()
```

In [ ]:
```
# Méthode qui permet de convertir un index de notre DataFrame en un dateTime
def convert_index_to_datetime(index):
    #la première date de notre base de donnée
    date = datetime.datetime.strptime(str(index)[:6],"%y%m%d")
    hour = int(str(index)[7:-1])//60
    minute = int(str(index)[7:-1])%60
    index_datetime = date + datetime.timedelta(hours=hour) + datetime.timedelta(minutes=
    return index_datetime
```

In [ ]:
```
#Methode qui créer un batch contenant l'ensemble des vecteurs et des labels associées ut
# input: DataFrame contenant les données
# output: le batch contenant tous les vectueurs du DataFrame avec les labels
def create_batchs_and_labels(data):

    iterator = 0
    labels =[]
    batch =  []
    #On boucle sur la base de donnée :
    while (iterator <= data.shape[0]-numFeature-1):

        dataExtracted = create_input(iterator,data)
        if (len(dataExtracted) == numFeature+1):
            sequence = []
            for x in dataExtracted[:-1]:
```

```
                sequence.append([x])
            batch.append(sequence)
            labels.append([dataExtracted[-1]])
        iterator+=1
    return torch.FloatTensor(batch),torch.FloatTensor(labels)
```

Pour avoir une structure plus facile d'utilisation, on va utiliser la bibliothèque pytorch. On va utiliser l'object Dataset. Ce qui va nous permettre de mieux gérer nos données pour l'apprentissage. Voici la documentation : https://pytorch.org/tutorials/beginner/basics/data_tutorial.html#creating-a-custom-dataset-for-your-files

In [ ]:
```
from torch.utils.data import Dataset

class Dataset(Dataset):
    def __init__(self, data_file):
        pf = pd.read_csv(str(data_file),sep=';',index_col='time')
        self.mean,self.std,self.prepross_pf = prepross_data(pf)
        self.batch,self.labels = create_batchs_and_labels(self.prepross_pf)

    def __len__(self):
        return self.batch.shape[0]

    def __getitem__(self, idx):
        return self.batch[idx],self.labels[idx]
```

Ainsi on créer notre Dataset à partir de notre csv.

In [ ]:
```
# training = Dataset("./bitcoin.csv")
training = Dataset("bitcoin.csv")
```

Puis on créer un Dataloader qui va charger les données pour pouvoir les utiliser et faire des calculs sur celles-ci. Voir la documentation pour plus d'information https://pytorch.org/tutorials/beginner/basics/data_tutorial.html#preparing-your-data-for-training-with-dataloaders.

Ici, on demande de faire une permutation sur nos vecteurs pour que l'apprentissage soit fait sur des vecteurs qui ne sont pas liés les uns aux autres par leur temporalité. Ici, le but de notre modèle est d'avoir en entrée un vecteur contenant les valeurs de clôture des $N_{feature}$ dernières minutes et le modèle doit prédire la valeur de la $N_{feature} + 1$ minute

In [ ]:
```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training, batch_size=5000, shuffle=True)
```

Voici du code pour afficher la tailler des différents batchs qu'on a ainsi chargé pour l'apprentissage

In [ ]:
```
for train_features, train_labels in iter(train_dataloader):
    print(f"taille d'un batch de feature : {train_features.size()}")
    print(f"taille d'un batch de labels : {train_labels.size()}")
```

In [ ]:
```

```