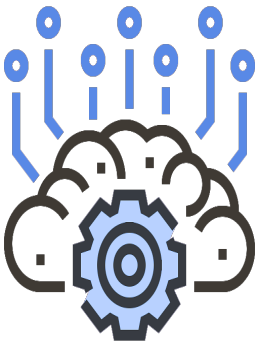
INPT

المعهد الوطني للبريد والمواصلات

ⵓⵙⴷⵍⵓⵙ ⵓⵏⵎⵙⵙⵓ ⵏ ⵜⵙⵓⵏⵓⵔⵜ ⴰ ⵙⵎⵔⵓⵏⵉ

Institut National des Postes et Télécommunications



PROJET2:

Stock Market Predictions with LSTM in Python

REALISE PAR :

- *EL-ASRI NOSSAIBA*
- *CHARAFI ASMAA*
- *AITJILLALI NOUHAILA*
- *BOUJIDA HAFSSA*

ENCADRE PAR :

- *M. IBN ELHAJ*

Remerciement :

Au terme de ce travail, nous désirons exprimer nos remerciements à tous ceux qui nous ont encouragés dans notre projet.

Tout d’abord nous tenons à exprimer notre gratitude à Monsieur **H. Ibn Elhaj**qui nous a aidé et qui nous a consacré du temps. Et aussi d’avoir donner l’opportunité de réaliser ce projet et acquérir toute ces compétences et développer notre information, pour faciliter notre assimilation du cours Deep Learning.

Introduction :

Dans ce projet, nous souhaitons modéliser correctement stock prices ;construire un modèle qui prédira si le prix augmentera ou diminuera.

Donc, nous avons besoin de bons modèles d'apprentissage automatique capables d'examiner l'historique d'une séquence de données et de prédire correctement quels seront les futurs éléments de la séquence, et nous choisissons LSTM, parce que Les LSTM sont très puissants dans les problèmes de prédiction séquentielle car ils sont capables de stocker des informations passées. C'est important dans notre cas, car le prix antérieur d'une action est crucial pour prédire son prix futur.

Il est important de noter qu'il existe toujours d'autres facteurs qui affectent les prix des actions, tels que l'atmosphère politique et le marché. Cependant, nous ne nous concentrerons pas sur ces facteurs

Les bibliothèques :

Nous avons besoin de ces bibliothèques pour pouvoir exécuter le

```python
from pandas_datareader import data
import matplotlib.pyplot as plt
import pandas as pd
import datetime as dt
import urllib.request, json
import os
import numpy as np
import tensorflow as tf
from sklearn.preprocessing import MinMaxScaler
```

code python

Downloading the data :

- On a installé le data à partir de ce site '' https://www.kaggle.com/borismarjanovic/price-volume-data-for-all-us-stocks-etfs ''

- Stock prices come in several different flavors. They are,
  - Open: Opening stock price of the day
  - Close: Closing stock price of the day
  - High: Highest stock price of the data
  - Low: Lowest stock price of the day

- Pour lire le fichier qui contient les données, on utilise la bibliothèque Pandas (pd)

```
df = pd.read_csv(os.path.join('Stocks','hpq.us.txt'),delimiter=',',usecols=['Date','Open','High','Low','Close'])
print('Loaded data from the Kaggle repository')
```

Data Exploration :

- Trier les données par date, car l'ordre des données est crucial dans la modélisation de séries chronologiques.

Le code :

```
# Sort DataFrame by date
df = df.sort_values('Date')

# Double check the result
df.head()
```

df - DataFrame

Index	Date	Open	High	Low	Close
0	1970-01-02	0.30627	0.30627	0.30627	0.30627
1	1970-01-05	0.30627	0.31768	0.30627	0.31385
2	1970-01-06	0.31385	0.31385	0.30996	0.30996
3	1970-01-07	0.31385	0.31385	0.31385	0.31385
4	1970-01-08	0.31385	0.31768	0.31385	0.31385
5	1970-01-09	0.31385	0.31768	0.31385	0.31768
6	1970-01-12	0.31768	0.32534	0.31768	0.32534
7	1970-01-13	0.32534	0.32916	0.32152	0.32152
8	1970-01-14	0.32152	0.32534	0.31768	0.32152
9	1970-01-15	0.32152	0.32916	0.32152	0.32916

Format | Resize | Background color | Column min/max | Save and Close | Close

Le résultat :

Data Visualization :

- On a tracé un graphe pour voir la différence du mid price par rapport au temps (mid price =(low + hight)/2)

```
plt.figure(figsize = (18,9))
plt.plot(range(df.shape[0]),(df['Low']+df['High'])/2.0)
plt.xticks(range(0,df.shape[0],500),df['Date'].loc[::500],rotation=45)
plt.xlabel('Date',fontsize=18)
plt.ylabel('Mid Price',fontsize=18)
plt.show()
```

Le code :

Le résultat :

- L'avantage de choisir cette société c'est qu'on a des comportements différents de 'stock prices' au fil du temps,Cela rendra l'apprentissage plus robuste et nous donnera un changement pour tester la qualité des prédictions pour une variété de situations.
- les valeurs proches de 2017 sont beaucoup plus élevées que les valeurs proches des années 1970, donc nous devons vous assurer que les données se comportent dans des plages de valeurs similaires tout au long de la période, pour cela nous fait la partie de normalisation de données.

Splitting Data into a Training set and a Test set:

- on a utilisé le prix moyen calculé en prenant la moyenne des prix enregistrés les plus élevés et les plus bas sur une journée.

```
# First calculate the mid prices from the highest and lowest
high_prices = df.loc[:,'High'].to_numpy()
low_prices = df.loc[:,'Low'].to_numpy()
mid_prices = (high_prices+low_prices)/2.0
```

- maintenant on a divisé les données d'entraînement et les données de test.
- Les données d'entrainement sont les 11 000 premiers points de données de la série chronologique et le reste est des données de test.

```
train_data = mid_prices[:11000]
test_data = mid_prices[11000:]
```

Normalizing the Data :

- Pour normaliser les données, nous avons besoin d'un scalaire.
- MinMaxScaler met à l'échelle toutes les données pour qu'elles soient dans la région de 0 et 1
- Après il faut remodeler les données d'entraînement et les données de test

```
scaler = MinMaxScaler()
train_data = train_data.reshape(-1,1)
test_data = test_data.reshape(-1,1)
```

- En raison de l'observation que nous avons faite précédemment, c'est-à-dire que différentes périodes de temps de données ont des plages de valeurs différentes

- Nous avons normalisé les données en divisant la série complète en fenêtres.

- Nous choisissons une taille de fenêtre de 2500. Car la taille ne doit pas être très petite, si on la choisit petite quand on fait la normalisation,elle peut introduire une rupture à la toute fin de chaque fenêtre, car chaque fenêtre est normalisée indépendamment

```
# Train the Scaler with training data and smooth data
smoothing_window_size = 2500
for di in range(0,10000,smoothing_window_size):
    scaler.fit(train_data[di:di+smoothing_window_size,:])
    train_data[di:di+smoothing_window_size,:] = scaler.transform(train_data[di:di+smoot

# You normalize the last bit of remaining data
scaler.fit(train_data[di+smoothing_window_size:,:])
train_data[di+smoothing_window_size:,:] = scaler.transform(train_data[di+smoothing_wind
```

- Remodeler les données :

```
# Reshape both train and test data
train_data = train_data.reshape(-1)

# Normalize test data
test_data = scaler.transform(test_data).reshape(-1)
```

- Nous pouvons maintenant lisser les données à l'aide de la moyenne exponentielle. Cela nous aide à vous débarrasser de l'irrégularité inhérente aux données des cours boursiers et à produire une courbe plus douce.

```
# Now perform exponential moving average smoothing
# So the data will have a smoother curve than the original ragged data
EMA = 0.0
gamma = 0.1
for ti in range(11000):
  EMA = gamma*train_data[ti] + (1-gamma)*EMA
  train_data[ti] = EMA

# Used for visualization and test purposes
all_mid_data = np.concatenate([train_data,test_data],axis=0)
```

One-Step Ahead Prediction via Averaging

Standard average:

$$x_{t+1} = 1/N \sum_{i=t-N}^{t} x_i$$

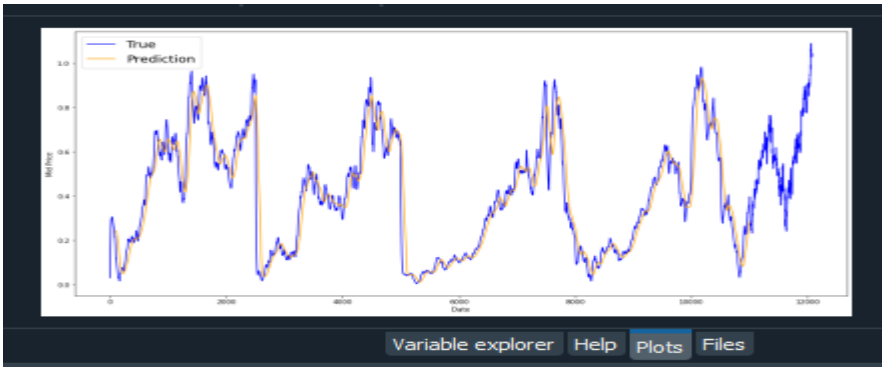
- la prédiction à t + 1 est la valeur moyenne de tous les cours boursiers que vous avez observés dans une fenêtre de t à t – N

```
window_size = 100
N = train_data.size
std_avg_predictions = []
std_avg_x = []
mse_errors = []

for pred_idx in range(window_size,N):

    if pred_idx >= N:
        date = dt.datetime.strptime(N, '%Y-%m-%d').date() + dt.timedelta(days=1)
    else:
        date = df.loc[pred_idx,'Date']

    std_avg_predictions.append(np.mean(train_data[pred_idx-window_size:pred_idx]))
    mse_errors.append((std_avg_predictions[-1]-train_data[pred_idx])**2)
    std_avg_x.append(date)

print('MSE error for standard averaging: %.5f'%(0.5*np.mean(mse_errors)))
```

```
MSE error for standard averaging: 0.00418
```

- Afficher le résultat :

```
plt.figure(figsize = (18,9))
plt.plot(range(df.shape[0]),all_mid_data,color='b',label='True')
plt.plot(range(window_size,N),std_avg_predictions,color='orange',label='Prediction')
#plt.xticks(range(0,df.shape[0],50),df['Date'].loc[::50],rotation=45)
plt.xlabel('Date')
plt.ylabel('Mid Price')
plt.legend(fontsize=18)
plt.show()
```



- Ces résultats moyens suivent de très près le comportement réel du stock. Ensuite, nous examinerons une méthode de prédiction en une étape plus précise.

Exponential Moving Average :

In the exponential moving average method, you calculate x_{t+1} as,

- $x\_t + 1 = EMA\_t = \gamma \times EMA_{t-1} + (1 - \gamma)x_t$ where $EMA_0 = 0$ and EMA is the exponential moving average value you maintain over time.

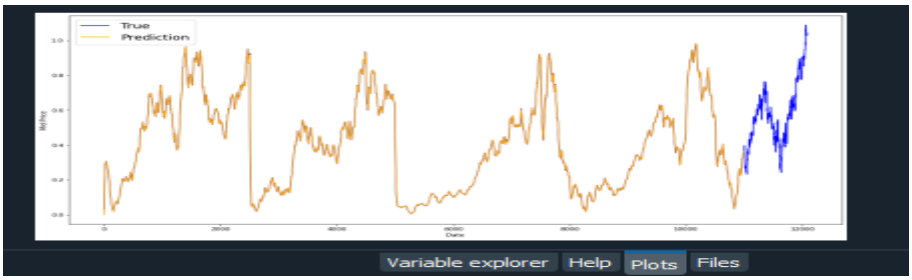
- γ décide de la contribution de la prédiction la plus récente à l'EMA. Par exemple, un γ = 0,1 obtient seulement 10% de la valeur actuelle dans l'EMA.

```
window_size = 100
N = train_data.size

run_avg_predictions = []
run_avg_x = []

mse_errors = []

running_mean = 0.0
run_avg_predictions.append(running_mean)

decay = 0.5

for pred_idx in range(1,N):

    running_mean = running_mean*decay + (1.0-decay)*train_data[pred_idx-1]
    run_avg_predictions.append(running_mean)
    mse_errors.append((run_avg_predictions[-1]-train_data[pred_idx])**2)
    run_avg_x.append(date)

print('MSE error for EMA averaging: %.5f'%(0.5*np.mean(mse_errors)))
```

```
MSE error for EMA averaging: 0.00003
```

- tracer la figure

```
plt.figure(figsize = (18,9))
plt.plot(range(df.shape[0]),all_mid_data,color='b',label='True')
plt.plot(range(0,N),run_avg_predictions,color='orange', label='Prediction')
#plt.xticks(range(0,df.shape[0],50),df['Date'].loc[::50],rotation=45)
plt.xlabel('Date')
plt.ylabel('Mid Price')
plt.legend(fontsize=18)
plt.show()
```



En pratique, on ne peut pas faire grand-chose avec juste la valeur boursière du lendemain. La chose importante est de savoir est ce que les cours boursiers augmenteraient ou diminueraient dans les 30 prochains jours. C'est l'incapacité de la méthode EMA.

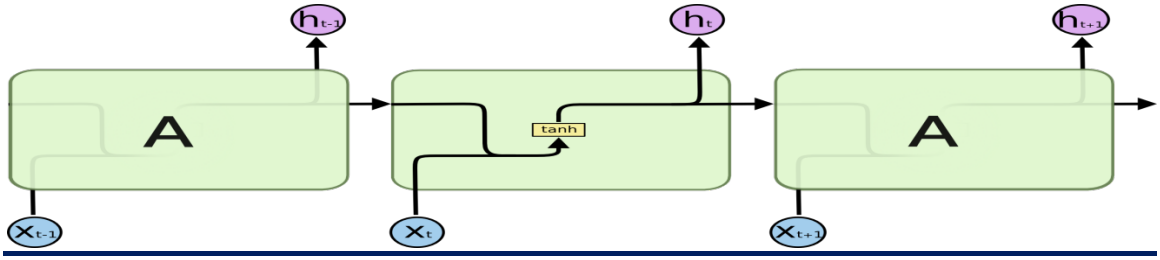
Pour cela on a utilisé un modèle plus complexe : un modèle LSTM.

Réseaux LSTM

Les réseaux de mémoire à long terme - généralement appelés «LSTM» - sont un type spécial de RNN, capable d'apprendre les dépendances à long terme. Ils ont été introduits par Hochreiter & Schmidhuber (1997) , et ont été affinés et popularisés par de nombreuses personnes dans les travaux suivants. Ils travaillent énormément bien sur une grande variété de problèmes, et sont maintenant largement utilisés.

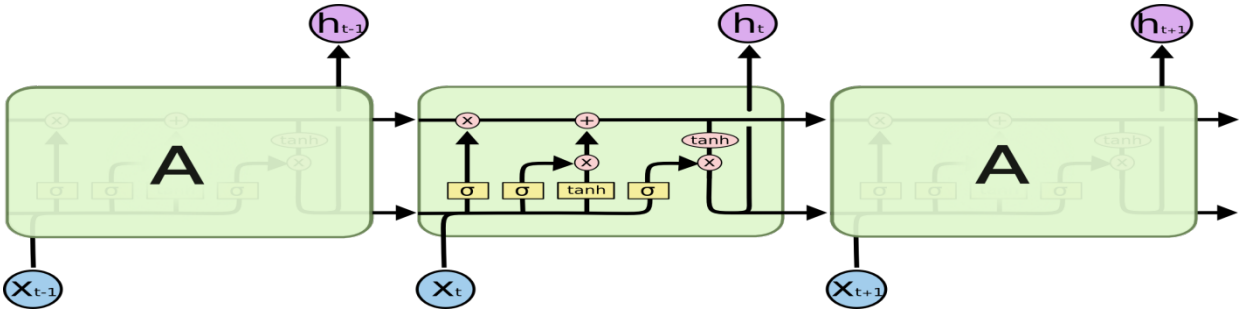
Les LSTM sont explicitement conçus pour éviter le problème de dépendance à long terme. Se souvenir des informations pendant de longues périodes est pratiquement leur comportement par défaut.

Tous les réseaux de neurones récurrents ont la forme d'une chaîne de modules répétitifs de réseau de neurones. Dans les RNN standard, ce module répétitif aura une structure très simple, telle qu'une seule couche de tanh.

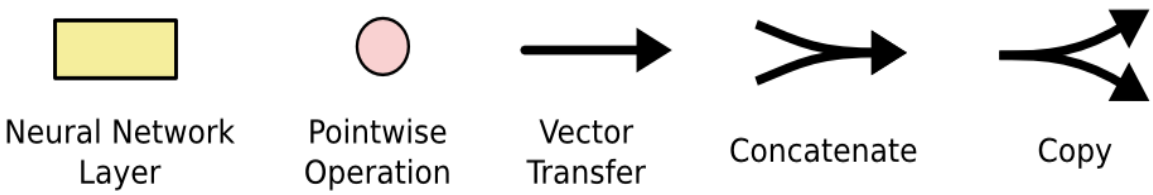


Le module répétitif dans un RNN standard contient une seule couche.

Les LSTM ont également cette structure en chaîne, mais le module répétitif a une structure différente. Au lieu d'avoir une seule couche de réseau neuronal, il y en a quatre, qui interagissent d'une manière très spéciale.



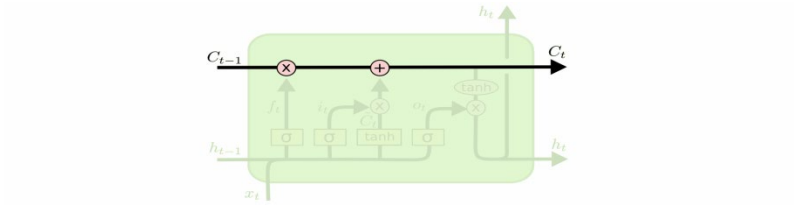
Le module répétitif dans un LSTM contient quatre couches interactives.



Dans le diagramme ci-dessus, chaque ligne porte un vecteur entier, de la sortie d'un nœud aux entrées des autres. Les cercles roses représentent des opérations ponctuelles, comme l'ajout de vecteurs, tandis que les cases jaunes sont des couches de réseau neuronal apprises. Les lignes qui fusionnent indiquent la concaténation, tandis qu'une fourchette de ligne indique que son contenu est copié et que les copies vont à des emplacements différents.

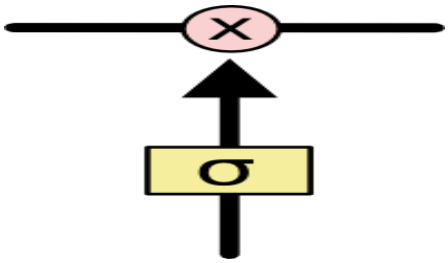
L'idée de base derrière les LSTM

La clé des LSTM est l'état de la cellule, la ligne horizontale qui traverse le haut du diagramme.L'état de la cellule est un peu comme une bande transporteuse. Il parcourt toute la chaîne, avec seulement quelques interactions linéaires mineures. Il est très facile pour les informations de circuler inchangées.



Le LSTM a la capacité de supprimer ou d'ajouter des informations à l'état de la cellule, soigneusement régulé par des structures appelées portes.

Les portes sont un moyen de laisser éventuellement passer des informations. Ils sont composés d'une couche de réseau neuronal sigmoïde et d'une opération de multiplication ponctuelle.

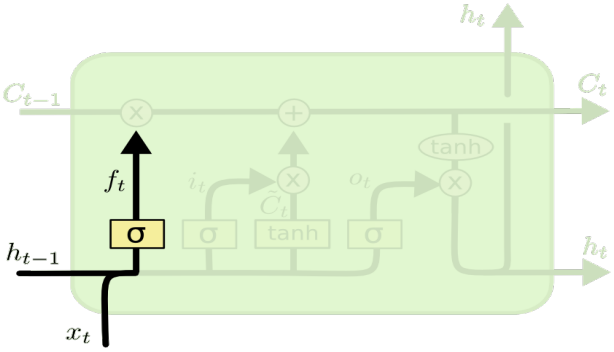


La couche sigmoïde produit des nombres entre zéro et un, décrivant la quantité de chaque composant à laisser passer. Une valeur de zéro signifie "ne rien laisser passer", tandis qu'une valeur de un signifie "tout laisser passer!"

Un LSTM a trois de ces portes, pour protéger et contrôler l'état de la cellule.

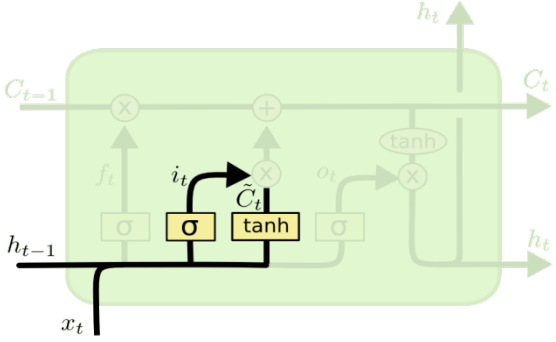
Présentation détaillée du LSTM

La première étape de notre LSTM est de décider quelles informations nous allons jeter de l'état de la cellule. Cette décision est prise par une couche sigmoïde appelée «couche de porte oublié». Il regarde **ht - 1** et **xt**, et génère un nombre entre 0 et 1 pour chaque nombre dans l'état de la cellule **Ct - 1**. Un 1 signifie «garder complètement ça» tandis qu'un 0 signifie «se débarrasser complètement de cela».



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

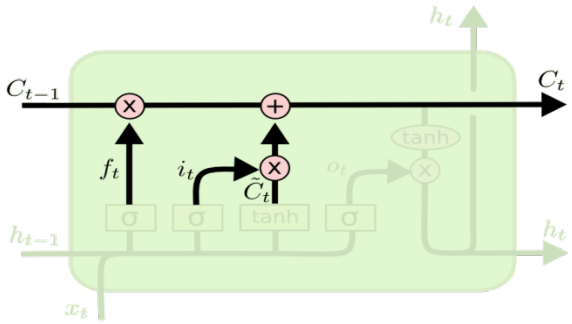
L'étape suivante consiste à décider quelles nouvelles informations nous allons stocker dans l'état de la cellule. Cela comporte deux parties. Tout d'abord, une couche sigmoïde appelée «couche de porte d'entrée» décide des valeurs que nous mettrons à jour. Ensuite, une couche tanh crée un vecteur de nouvelles valeurs candidates Ct, cela pourrait être ajouté à l'État. Dans l'étape suivante, nous combinerons ces deux éléments pour créer une mise à jour de l'état.



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

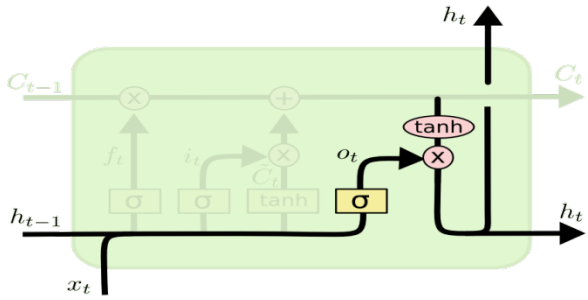
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Il est maintenant temps de mettre à jour l'ancien état de la cellule, Ct - 1, dans le nouvel état de cellule Ct. Les étapes précédentes ont déjà décidé ce qu'il faut faire, il suffit de le faire.
Nous multiplions l'ancien état par ft, oubliant les choses que nous avons décidé d'oublier plus tôt. Ensuite, nous ajoutons it∗C~t. Il s'agit des nouvelles valeurs candidates, mises à l'échelle de la mesure dans laquelle nous avons décidé de mettre à jour chaque valeur d'état.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Enfin, nous devons décider de ce que nous allons produire. Cette sortie sera basée sur l'état de notre cellule, mais sera une version filtrée. Tout d'abord, nous exécutons une couche sigmoïde qui décide quelles parties de l'état de la cellule nous allons générer. Ensuite, nous mettons l'état de la cellule à travers tanh (pour pousser les valeurs entre −1 et 1) et multipliez-le par la sortie de la porte sigmoïde, de sorte que nous ne sortions que les parties que nous avons décidées.

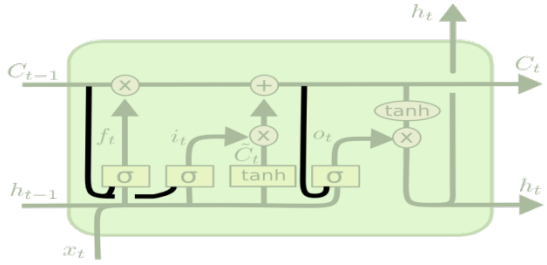


$$o_t = \sigma\left(W_o\ [\,h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Variantes sur la mémoire à long terme

Ce qu'on a décrit jusqu'à présent est un LSTM assez normal. Mais tous les LSTM ne sont pas identiques à ceux ci-dessus. En fait, il semble que presque tous les articles impliquant des LSTM utilisent une version légèrement différente. Les différences sont mineures, mais il convient d'en mentionner quelques-unes.

Une variante LSTM populaire, introduite par Gers & Schmidhuber (2000) , consiste à ajouter des «connexions judas». Cela signifie que nous laissons les couches de grille regarder l'état de la cellule.

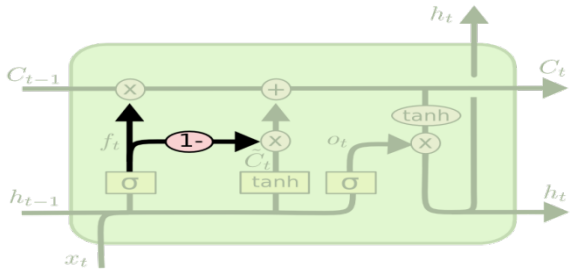


$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t]\ +\ b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t]\ +\ b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t]\ +\ b_o\right)$$

Le diagramme ci-dessus ajoute des judas à toutes les portes, mais de nombreux papiers donneront des judas et pas d'autres.

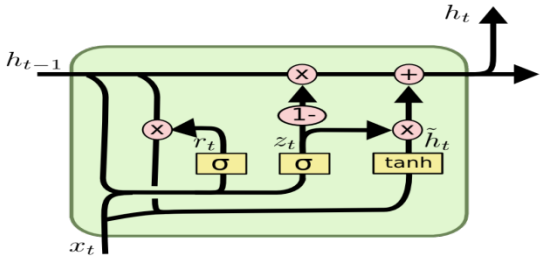
Une autre variante consiste à utiliser des portes d'entrée et d'entrée couplées. Au lieu de décider séparément ce qu'il faut oublier et ce à quoi nous devrions ajouter de nouvelles informations, nous prenons ces décisions

ensemble. On oublie seulement quand on va entrer quelque chose à sa place. Nous n'entrons de nouvelles valeurs dans l'état que lorsque nous oublions quelque chose de plus ancien.



$$C_t = f_t * C_{t-1} + (\mathbf{1} - \boldsymbol{f_t}) * \tilde{C}_t$$

Une variation légèrement plus dramatique sur le LSTM est l'unité récurrente fermée, ou GRU, introduite par Cho et al. (2014) . Il combine les portes d'oubli et d'entrée en une seule «porte de mise à jour». Il fusionne également l'état de la cellule et l'état masqué et apporte d'autres modifications. Le modèle résultant est plus simple que les modèles LSTM standard et est de plus en plus populaire.



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

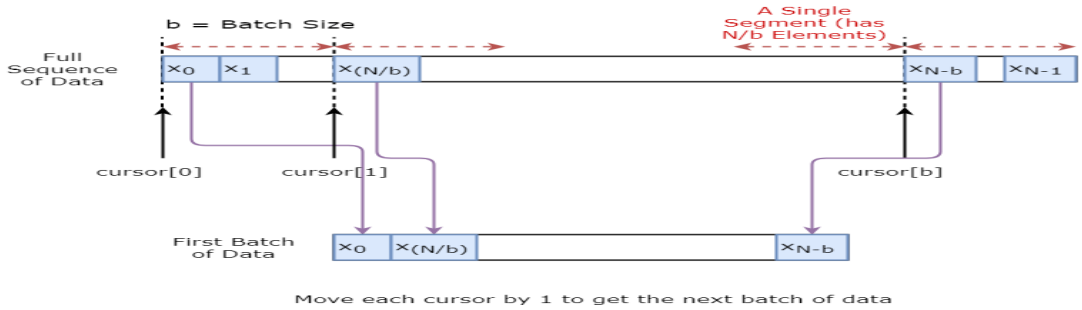
Ce ne sont que quelques-unes des variantes LSTM les plus notables. Il y en a beaucoup d'autres, comme les RNN à seuil de profondeur de Yao et al. (2015) . Il existe également une approche complètement différente pour lutter contre les dépendances à long terme, comme les RNN Clockwork de Koutnik et al. (2014) .

Implémentation du code :

1/Générateur de données :

Nous allons d'abord implémenter un générateur de données pour entraîner votre modèle. Ce générateur de données aura une méthode appelée .unroll_batches(...)qui produira un ensemble de lots *num_unrollings* de données d'entrée obtenues séquentiellement, où un lot de données est de taille [batch_size, 1]. Ensuite, chaque lot de données d'entrée aura un lot de données de sortie correspondant.

Ci-dessous, vous illustrez comment un lot de données est créé visuellement.



On définit la classe DataGeneratorSeq :

```
class DataGeneratorSeq(object):
```

Puis la méthode def _init_ :

```
def __init__(self,prices,batch_size,num_unroll):
    self._prices = prices
    self._prices_length = len(self._prices) - num_unroll
    self._batch_size = batch_size
    self._num_unroll = num_unroll
    self._segments = self._prices_length //self._batch_size
    self._cursor = [offset * self._segments for offset in range(self._batch_size)]
```

On définit les paramètres suivants :

- prices : c'est la séquence entière de data
- prices_lenght : la longueur de la séquence
- batch_size : un seul pas de temps
- num_unroll :le nombre de pas de temps utilisé dans un seul entrainement
- segment : le nombre de num_unroll
- cursor : les indices du prices qui vont constituer les num_unroll

La méthode next_batch :

Créer un lot de données

```
def next_batch(self):

    batch_data = np.zeros((self._batch_size),dtype=np.float32)
    batch_labels = np.zeros((self._batch_size),dtype=np.float32)

    for b in range(self._batch_size):
        if self._cursor[b]+1>=self._prices_length:
            #self._cursor[b] = b * self._segments
            self._cursor[b] = np.random.randint(0,(b+1)*self._segments)

        batch_data[b] = self._prices[self._cursor[b]]
        batch_labels[b]= self._prices[self._cursor[b]+np.random.randint(0,5)]

        self._cursor[b] = (self._cursor[b]+1)%self._prices_length

    return batch_data,batch_labels
```

La méthode unroll_batches :

Elle produira un ensemble de lots *(num_unroll)* de données d'entrée obtenues séquentiellement, où un lot de données est de taille (batch_size). Ensuite, chaque lot de données d'entrée aura un lot de données de sortie correspondant.

```
def unroll_batches(self):

    unroll_data,unroll_labels = [],[]
    init_data, init_label = None,None
    for ui in range(self._num_unroll):

        data, labels = self.next_batch()

        unroll_data.append(data)
        unroll_labels.append(labels)

    return unroll_data, unroll_labels
```

La méthode reset_indices :

Réinitialiser les cursors pour construire le batch suivant

```
def reset_indices(self):
    for b in range(self._batch_size):
        self._cursor[b] = np.random.randint(0,min((b+1)*self._segments,self._prices_length-1))
```

2/ Définitions des hyperparamètres du modèle :

- D est la dimensionnalité de l'entrée. C'est simple, car vous prenez le cours de l'action précédent comme entrée et prédisez le prochain, qui devrait l'être 1.
- num_unrolling : il s'agit d'un hyper paramètre lié à la back propagation dans le temps (BPTT) qui est utilisé pour optimiser le modèle LSTM. Cela indique le nombre de pas de temps continus que vous considérez pour une seule étape d'optimisation. Vous pouvez penser à cela comme, au lieu d'optimiser le modèle en regardant une seule étape de temps, vous optimisez le réseau en regardant les num_unrolling étapes de temps. Le plus grand sera le mieux.
- batch_size : La taille du lot correspond au nombre d'échantillons de données que vous considérez en une seule étape.
- num_nodes : le nombre de neurones cachés dans chaque cellule.

```
D = 1 # Dimensionality of the data. Since your data is 1-D this would be 1
num_unrollings = 50 # Number of time steps you look into the future.
batch_size = 500 # Number of samples in a batch
num_nodes = [200,200,150] # Number of hidden nodes in each layer of the deep LSTM stack we're using
n_layers = len(num_nodes) # number of layers
dropout = 0.2 # dropout amount

tf.reset_default_graph() # This is important in case you run this multiple times
```

3/ Définitions des entrées et sorties :

Définitions des espaces réservés pour les entrées et les placeholders de formation. On a une liste d'espaces réservés d'entrée, où chaque espace réservé contient un seul lot de données. Et la liste a des num_unrollings espaces réservés, qui seront utilisés à la fois pour une seule étape d'optimisation.

```
# Input data.
train_inputs, train_outputs = [],[]

# You unroll the input over time defining placeholders for each time step
for ui in range(num_unrollings):
    train_inputs.append(tf.placeholder(tf.float32, shape=[batch_size,D],name='train_inputs_%d'%ui))
    train_outputs.append(tf.placeholder(tf.float32, shape=[batch_size,1], name = 'train_outputs_%d'%ui))
```

4/ Définition des paramètres de la couche LSTM et de régression :

Vous aurez trois couches de LSTM et une couche de régression linéaire, désignée par w et b, qui prend la sortie de la dernière cellule de mémoire à long court terme et génère la prédiction pour le pas de temps suivant. Vous pouvez utiliser MultiRNNCell dans TensorFlow pour encapsuler les trois LSTMCell objets que vous avez créés. En outre, vous pouvez avoir les cellules LSTM implémentées par abandon, car elles améliorent les performances et réduisent le overfitting.

```
lstm_cells = [
    tf.contrib.rnn.LSTMCell(num_units=num_nodes[li],
                            state_is_tuple=True,
                            initializer= tf.contrib.layers.xavier_initializer()
                           )
 for li in range(n_layers)]

drop_lstm_cells = [tf.contrib.rnn.DropoutWrapper(
    lstm, input_keep_prob=1.0,output_keep_prob=1.0-dropout, state_keep_prob=1.0-dropout
) for lstm in lstm_cells]
drop_multi_cell = tf.contrib.rnn.MultiRNNCell(drop_lstm_cells)
multi_cell = tf.contrib.rnn.MultiRNNCell(lstm_cells)

w = tf.get_variable('w',shape=[num_nodes[-1], 1], initializer=tf.contrib.layers.xavier_initializer())
b = tf.get_variable('b',initializer=tf.random_uniform([1],-0.1,0.1))
```

5/ Calculer la sortie LSTM et la transmettre à la couche de régression pour obtenir la prédiction finale :

Création des variables TensorFlow c et h qui contiendront l'état de la cellule et l'état masqué de la cellule de mémoire à long court terme. Ensuite, on transforme la liste de train_inputs pour avoir une forme de[num_unrollings, batch_size, D], ceci est nécessaire pour calculer les sorties avec la tf.nn.dynamic_rnn fonction. On calcule ensuite les sorties LSTM avec la tf.nn.dynamic_rnn fonction et on divise la sortie en une liste de num_unrollings tensors, la perte entre les prévisions et les cours réels des actions.

```
# Create cell state and hidden state variables to maintain the state of the LSTM
c, h = [],[]
initial_state = []
for li in range(n_layers):
  c.append(tf.Variable(tf.zeros([batch_size, num_nodes[li]]), trainable=False))
  h.append(tf.Variable(tf.zeros([batch_size, num_nodes[li]]), trainable=False))
  initial_state.append(tf.contrib.rnn.LSTMStateTuple(c[li], h[li]))

# Do several tensor transofmations, because the function dynamic_rnn requires the output to be of
# a specific format. Read more at: https://www.tensorflow.org/api_docs/python/tf/nn/dynamic_rnn
all_inputs = tf.concat([tf.expand_dims(t,0) for t in train_inputs],axis=0)

# all_outputs is [seq_length, batch_size, num_nodes]
all_lstm_outputs, state = tf.nn.dynamic_rnn(
    drop_multi_cell, all_inputs, initial_state=tuple(initial_state),
    time_major = True, dtype=tf.float32)

all_lstm_outputs = tf.reshape(all_lstm_outputs, [batch_size*num_unrollings,num_nodes[-1]])

all_outputs = tf.nn.xw_plus_b(all_lstm_outputs,w,b)

split_outputs = tf.split(all_outputs,num_unrollings,axis=0)
```

Loss calculation and Optimizer:

Maintenant, nous allons calculer la valeur du LOSS. Cependant, nous devons noter qu'il existe une caractéristique unique lors du calcul du loss :

Pour chaque lot de prédictions et de sorties vraies, on calcule l'erreur quadratique moyenne. Et on additionne (pas en moyenne) toutes ces pertes moyennes au carré. Enfin, on définit l'optimiseur qu'on va utiliser pour optimiser le réseau de neurones. Dans ce cas, nous avons utilisé Adam, qui est un optimiseur très récent et performant.

Nous allons commencer par définir :

Training loss :

```
print('Defining training Loss')
loss = 0.0
with tf.control_dependencies([tf.assign(c[li], state[li][0]) for li in range(n_layers)]+
                             [tf.assign(h[li], state[li][1]) for li in range(n_layers)]):
  for ui in range(num_unrollings):
    loss += tf.reduce_mean(0.5*(split_outputs[ui]-train_outputs[ui])**2)
```

Learning rate :

```
print('Learning rate decay operations')
global_step = tf.Variable(0, trainable=False)
inc_gstep = tf.assign(global_step,global_step + 1)
tf_learning_rate = tf.placeholder(shape=None,dtype=tf.float32)
tf_min_learning_rate = tf.placeholder(shape=None,dtype=tf.float32)

learning_rate = tf.maximum(
    tf.train.exponential_decay(tf_learning_rate, global_step, decay_steps=1, decay_rate=0.5, staircase=True),
    tf_min_learning_rate)
```

Remarque: *“the learning rate is perhaps the most important hyperparameter. If you have time to tune only one hyperparameter, tune the learning rate.”*

Optimizer :

```
# Optimizer.
print('TF Optimization operations')
optimizer = tf.train.AdamOptimizer(learning_rate)
gradients, v = zip(*optimizer.compute_gradients(loss))
gradients, _ = tf.clip_by_global_norm(gradients, 5.0)
optimizer = optimizer.apply_gradients(zip(gradients, v))

print('\tAll done')
```

Après exécution :

```
Defining training Loss
Learning rate decay operations
TF Optimization operations
    All done
```

Calculs liés à la prédiction :

Ici, nous allons définir les opérations TensorFlow liées à la prédiction.

Tout d'abord, on définit un espace réservé pour l'alimentation de l'entrée (sample_inputs).

Puis, comme pour l'étape d'apprentissage, vous définissez des variables d'état pour la prédiction (sample_c et sample_h).

Enfin, nous calculons la prédiction avec la fonction tf.nn.dynamic_rnn, puis nous envoyons la sortie via la couche de régression (w et b).

Nous devons également définir l'opération reset_sample_state, qui réinitialise l'état de la cellule et l'état masqué. Et exécuter cette opération au début, chaque fois que nous effectuons une séquence de prédictions.

Le code est le suivant:

```
### Prediction Related Calculations

print('Defining prediction related TF functions')

sample_inputs = tf.placeholder(tf.float32, shape=[1,D])

# Maintaining LSTM state for prediction stage
sample_c, sample_h, initial_sample_state = [],[],[]
for li in range(n_layers):
  sample_c.append(tf.Variable(tf.zeros([1, num_nodes[li]]), trainable=False))
  sample_h.append(tf.Variable(tf.zeros([1, num_nodes[li]]), trainable=False))
  initial_sample_state.append(tf.contrib.rnn.LSTMStateTuple(sample_c[li],sample_h[li]))

reset_sample_states = tf.group(*[tf.assign(sample_c[li],tf.zeros([1, num_nodes[li]])) for li in range(n_layers)
                                 *[tf.assign(sample_h[li],tf.zeros([1, num_nodes[li]])) for li in range(n_layers)

sample_outputs, sample_state = tf.nn.dynamic_rnn(multi_cell, tf.expand_dims(sample_inputs,0),
                                    initial_state=tuple(initial_sample_state),
                                    time_major = True,
                                    dtype=tf.float32)

with tf.control_dependencies([tf.assign(sample_c[li],sample_state[li][0]) for li in range(n_layers)]+
                             [tf.assign(sample_h[li],sample_state[li][1]) for li in range(n_layers)]):
  sample_prediction = tf.nn.xw_plus_b(tf.reshape(sample_outputs,[1,-1]), w, b)

print('\tAll done')
```

Après exécution :

```
Defining prediction related TF functions

    All done
```

Running the LSTM :

Ici, nous allons entraîner et prédire les mouvements du « Stock price » pour plusieurs époques et voir si les prévisions s'améliorent ou empirent au fil du temps. On va suivre la procédure suivante :

1- Définir un ensemble de tests de point de départ (test_points_seq) sur la série temporelle pour évaluer le modèle.

2- Et puis, pour chaque époque on va :

- Pour la longueur de séquence complète du training data :
  . Dérouler un ensemble de lots de num_unrollings
  . Entraînez le réseau neuronal avec les lots déroulés
- Calculer the average training loss.
- Pour chaque point de départ du test set :
  . Mettre à jour l'état LSTM en itérant à travers le précédent num_unrollings les points de données trouvés avant le test point.
  . Faire des prédictions pour n_predict_once étapes en continu, en utilisant le prédiction précédente comme entrée actuelle.
  . Calculer la perte MSE entre les n_predict_once point prédits et les prix réels des actions à ces horodatages.

Le code est le suivant:

```
epochs = 30
valid_summary = 1 # Interval you make test predictions

n_predict_once = 50 # Number of steps you continously predict for

train_seq_length = train_data.size # Full length of the training data

train_mse_ot = [] # Accumulate Train losses
test_mse_ot = [] # Accumulate Test loss
predictions_over_time = [] # Accumulate predictions

session = tf.InteractiveSession()

tf.global_variables_initializer().run()

# Used for decaying learning rate
loss_nondecrease_count = 0
loss_nondecrease_threshold = 2 # If the test error hasn't increased in this many steps, decrease learning rate

print('Initialized')
average_loss = 0

# Define data generator
data_gen = DataGeneratorSeq(train_data,batch_size,num_unrollings)

x_axis_seq = []
```

```
# Points you start your test predictions from
test_points_seq = np.arange(11000,12000,50).tolist()

for ep in range(epochs):

    # ======================== Training ====================================
    for step in range(train_seq_length//batch_size):

        u_data, u_labels = data_gen.unroll_batches()

        feed_dict = {}
        for ui,(dat,lbl) in enumerate(zip(u_data,u_labels)):
            feed_dict[train_inputs[ui]] = dat.reshape(-1,1)
            feed_dict[train_outputs[ui]] = lbl.reshape(-1,1)

        feed_dict.update({tf_learning_rate: 0.0001, tf_min_learning_rate:0.000001})

        _, l = session.run([optimizer, loss], feed_dict=feed_dict)

        average_loss += l
    # ============================ Validation ==============================
    if (ep+1) % valid_summary == 0:

      average_loss = average_loss/(valid_summary*(train_seq_length//batch_size))

      # The average loss
      if (ep+1)%valid_summary==0:
        print('Average loss at step %d: %f' % (ep+1, average_loss))

      train_mse_ot.append(average_loss)

      average_loss = 0 # reset loss

      predictions_seq = []

      mse_test_loss_seq = []
```

```
# ==================== Updating State and Making Predicitons ========================
for w_i in test_points_seq:
  mse_test_loss = 0.0
  our_predictions = []

  if (ep+1)-valid_summary==0:
    # Only calculate x_axis values in the first validation epoch
    x_axis=[]

  # Feed in the recent past behavior of stock prices
  # to make predictions from that point onwards
  for tr_i in range(w_i-num_unrollings+1,w_i-1):
    current_price = all_mid_data[tr_i]
    feed_dict[sample_inputs] = np.array(current_price).reshape(1,1)
    _ = session.run(sample_prediction,feed_dict=feed_dict)

  feed_dict = {}

  current_price = all_mid_data[w_i-1]

  feed_dict[sample_inputs] = np.array(current_price).reshape(1,1)

  # Make predictions for this many steps
  # Each prediction uses previous prediciton as it's current input
  for pred_i in range(n_predict_once):

    pred = session.run(sample_prediction,feed_dict=feed_dict)

    our_predictions.append(np.asscalar(pred))

    feed_dict[sample_inputs] = np.asarray(pred).reshape(-1,1)

    if (ep+1)-valid_summary==0:
      # Only calculate x_axis values in the first validation epoch
      x_axis.append(w_i+pred_i)
```

```
    mse_test_loss += 0.5*(pred-all_mid_data[w_i+pred_i])**2

  session.run(reset_sample_states)

  predictions_seq.append(np.array(our_predictions))

  mse_test_loss /= n_predict_once
  mse_test_loss_seq.append(mse_test_loss)

  if (ep+1)-valid_summary==0:
    x_axis_seq.append(x_axis)

current_test_mse = np.mean(mse_test_loss_seq)

# Learning rate decay logic
if len(test_mse_ot)>0 and current_test_mse > min(test_mse_ot):
    loss_nondecrease_count += 1
else:
    loss_nondecrease_count = 0

if loss_nondecrease_count > loss_nondecrease_threshold :
      session.run(inc_gstep)
      loss_nondecrease_count = 0
      print('\tDecreasing learning rate by 0.5')

test_mse_ot.append(current_test_mse)
print('\tTest MSE: %.5f'%np.mean(mse_test_loss_seq))
predictions_over_time.append(predictions_seq)
print('\tFinished Predictions')
```

Après exécution :

```
Initialized
Average loss at step 1: 1.703350
    Test MSE: 0.00318
    Finished Predictions
  ...
  ...
  ...
Average loss at step 30: 0.033753
    Test MSE: 0.00243
    Finished Predictions
```

On remarque qu'après chaque étape, the average loss se diminue,

Ainsi que la perte MSE .

Et donc la prédiction devient plus précise au fil du temps .

Visualisation des prédictions :

Après l'exécution du LSTM, il est le temps de visualiser les prédictions, et voir s'ils s'améliorent ou s'aggravent au fil du temps.

Après tous calculs fait pour la sortie LSTM et alimentation à la couche de régression pour obtenir la prédiction finale, et d'après le calcule de la prédiction en se basant sur les opérations TensorFlow liées à la prédiction, nous avons pu conclure que pour une meilleure prédiction la valeur « epoch » c'est-à-dire nombre d'itération est « 28 ».

Le code est le suivant:

```
#=============== Visualiser les prédictions ====================
best_prediction_epoch = 28
# replace this with the epoch that you got the best results when running the plotting code

plt.figure(figsize = (18,18))
plt.subplot(2,1,1)
plt.plot(range(df.shape[0]),all_mid_data,color='b')
```

Ensuite, puisque le but de cette partie est de visualiser le changement de prédiction au fil du temps, nous commençons par tracer à la fois l'ancienne valeur de prédiction avec la variable alpha inférieure, cette dernière étant la valeur de l'étape d'apprentissage> 0, et aussi avec la nouvelle prédiction avec une valeur alpha plus élevée, nous avons donc combiné la boucle for, pour afficher le test de prédiction à tout moment, comme indiqué dans le code s'il est ci-dessous.

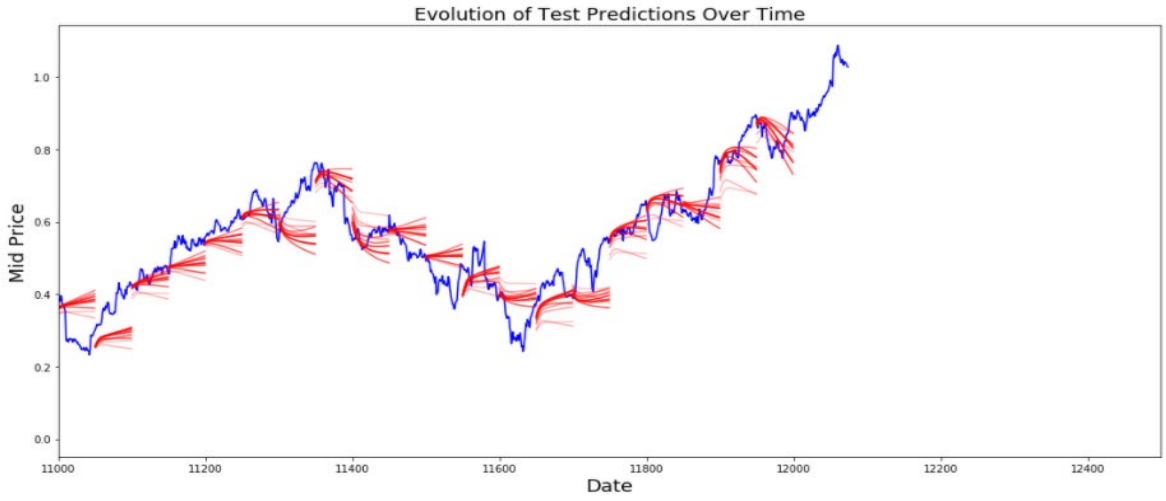
```
# Plotting how the predictions change over time
# Plot older predictions with low alpha and newer predictions with high alpha
start_alpha = 0.25
alpha  = np.arange(start_alpha,1.1,(1.0-start_alpha)/len(predictions_over_time[::3]))
for p_i,p in enumerate(predictions_over_time[::3]):
    for xval,yval in zip(x_axis_seq,p):
        plt.plot(xval,yval,color='r',alpha=alpha[p_i])

plt.title('Evolution of Test Predictions Over Time',fontsize=18)
plt.xlabel('Date',fontsize=18)
plt.ylabel('Mid Price',fontsize=18)
plt.xlim(11000,12500)
```

Afin d'exécuter le code si dessus, notre but est d'afficher la meilleure prédiction après le test effectuer. Le code est le suivant :

```
# Predicting the best test prediction you got
plt.plot(range(df.shape[0]),all_mid_data,color='b')
for xval,yval in zip(x_axis_seq,predictions_over_time[best_prediction_epoch]):
    plt.plot(xval,yval,color='r')

plt.title('Best Test Predictions Over Time',fontsize=18)
plt.xlabel('Date',fontsize=18)
plt.ylabel('Mid Price',fontsize=18)
plt.xlim(11000,12500)
plt.show()
```

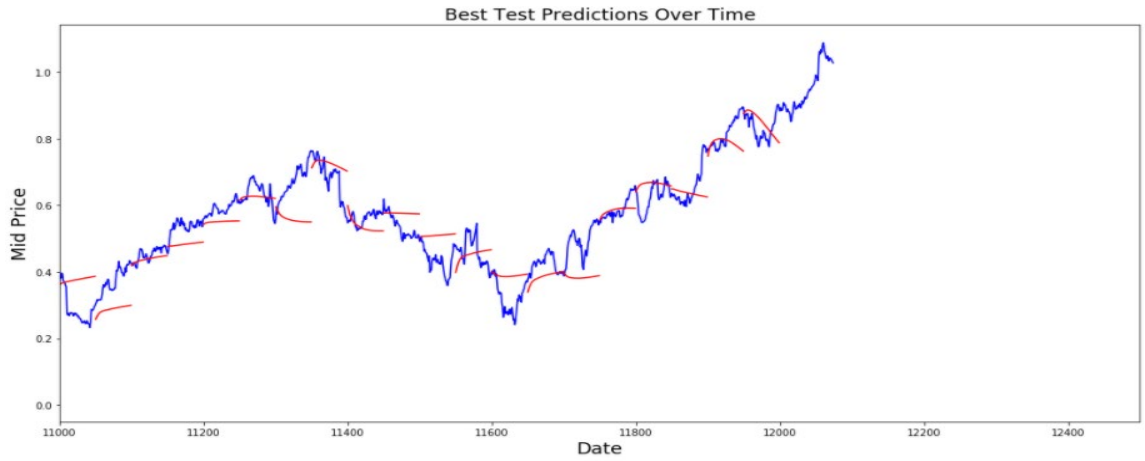
Par cette partie du code, nous arrivons à afficher les divers changements du prix, par la variation de la valeur alpha et la meilleure valeur « epoch » de prédiction. Le programme s'exécute étant de fois jusqu'au arriver au but général c'est « best test prediction ».

Après l'exécution, nous recevons :



La courbe en bleu représenté notre informations, c'est-à-dire les données concernant le « stock prices », et les courbes en rouge représentent les prédictions à chaque valeur de alpha, pour arriver à la fin à une bonne amélioration, donc nous avons conclus que ce modèle arrive à apprendre et à modéliser le changement du prix au fil du temps.

L'autre figure est la suivante :



Nous pouvons voir comment la perte de MSE est en baisse avec la quantité de formation. Notre remarque, que cela c'est un bon signe que le modèle apprend quelque chose d'utile.

D'autre part, pour quantifier nos résultats, nous avons comparé la perte de MSE du réseau à la perte d'ESH reçue lors de la moyenne standard (0,004).

Nous pouvons conclure que le LSTM fait mieux que la moyenne standard. Aussi que la moyenne standard (mais pas parfaite) a suivi les mouvements de prix des actions vrai raisonnablement.

D'après la lecture de ces deux figures, nous réussissons à avoir un modèle LSTM qui modélise correctement stock prices et qui prédit si le prix augmentera ou diminuera au fil du temps.

Conclusion :

Dans ce projet, nous avons réussi à comprendre les cellules à mémoire interne : les LSTM et comprendre le mécanisme des portes de contrôle. Ensuite, nous avons assimilé l'utilisation des LSTM en couches et comprendre l'intérêt des LSTM bidirectionnels et multidimensionnels, et leur fonctionnement.

D’après la présentation d’une version améliorée des réseaux de neurones récurrents simples : nous avons conclu que les LSTM, sont capables de modéliser des dépendances à très long terme.

Nous avons appris aussi que les LSTM reposent sur un mécanisme de mémoire interne piloté par des portes de contrôle. Tous les éléments étant différentiables, ces réseaux s'apprennent par une rétropropagation à travers le temps classique. Et par l’application de la régression, nous avons arrivé à comprendre combien il peut être difficile de périphérique d’un modèle qui est capable de prédire correctement les mouvements du cours des actions.

L’acheminement du projet c’est de commencer avec une motivation pour expliquer pourquoi le besoin de modéliser les prix des actions. Ceci a été suivi d’une explication et d’un code pour télécharger des données. Ensuite, nous avons examiné deux techniques de moyenne qui vous permettent de faire des prédictions un pas dans l’avenir. Dont nous avons remarqué que ces méthodes sont futiles lorsque le besoin de prédire plus d’un pas dans l’avenir.

Par la suite, une discussion de la façon dont vous pouvez utiliser les MST pour faire des prédictions à plusieurs étapes de l’avenir. Enfin, la visualisation des résultats et vu que notre modèle (mais pas parfait) est assez bon pour prédire correctement les mouvements du cours des actions.

Ce projet présente vraiment une expérience enrichissante, nous avons bien appris que les LSTM sont capables de modéliser des dépendances à très long terme.

Référence :

- (Tutorial) LSTM in Python: Stock Market Predictions - DataCamp
- Huge Stock Market Dataset | Kaggle