Trading algorithmique sur cryptomonnaies

Valentin Petoin et Pierre Buteaux

07/04/2021

ING4 FIN Gr2

INTRODUCTION

Le trading algorithmique connaît aujourd'hui une forte adoption et représente maintenant la majorité des transactions financières conduites sur les marchés internationaux.

Ce projet consiste en la conception d'un bot qui se connectera à une plateforme de trading où il assurera le pilotage d'un portefeuille de crypto-monnaies, exécutant des ordres d'achat et de vente.

Qu'est ce qu'un crypto trading bot ?

Un bot est tout simplement un programme automatisé qui fonctionne sur internet et effectue des tâches répétitives plus efficacement que les humains. En effet, certaines estimations suggèrent qu'environ la moitié du trafic internet est constituée de bots qui interagissent avec les pages web et les utilisateurs, recherchent du contenu et effectuent des tâches automatisées.

Les robots de trading de crypto-monnaies fonctionnent selon le même principe.
Ce sont des programmes qui exécutent des fonctions à l'aide d'une intelligence artificielle en fonction de paramètres préétablis. Plus de transactions ou d'opportunités manquées, en exécutant un ensemble d'algorithmes, vous pouvez automatiquement acheter, vendre ou conserver des actifs de manière opportune, efficace et automatisée, de jour comme de nuit, de n'importe où dans le monde.

Les bots cryptographiques se présentent sous de nombreuses formes, mais ils appartiennent tous à quatre grandes catégories : les bots de trend-trading, les bots d'arbitrage, les bots coin-lending et les bots market-maker.

Comme leur nom l'indique, les bots trend-trading tentent de réaliser des gains en analysant la dynamique d'un actif dans une direction particulière. Discerner les tendances peut s'avérer utile lors de la mise en place automatique de “take-profit” ou de” stop-loss” afin de capturer des bénéfices ou d'éviter des pertes.

Les bots d'arbitrage sont utilisés pour identifier les inefficacités et les différences de prix entre les marchés.

Pour ceux qui souhaitent prêter des crypto-monnaies à des taux d'intérêt avantageux avec des risques limités, les bots de coin-lending automatisent le processus, atténuant la

volatilité des taux d'intérêt et des remboursements de prêts par les emprunteurs.

Quant aux bots de tenue de marché (market-maker), ils réalisent des bénéfices sur la différence entre le prix de vente (ask) et le prix d'achat (bid), que l'on appelle le spread.

Comment les robots de trading fonctionnent-ils?

En communiquant directement avec les bourses de crypto-monnaies et en passant des ordres automatiquement en fonction de nos propres conditions prédéfinies, les bots de trading de crypto-monnaies offrent une vitesse et une efficacité exceptionnelles, moins d'erreurs et un trading non affecté par l'émotion et des comportements biaisés propre à l'humain. Pour négocier sur une bourse, nous devons autoriser un robot de trading à accéder à notre compte via des clés API (Application Program Interface), et l'accès peut être accordé ou retiré à tout moment.

Les robots de trading fonctionnent en trois étapes essentielles : le générateur de signaux, l'allocation des risques et l'exécution.

Le générateur de signaux fait essentiellement le travail du trader, en faisant des prédictions et en identifiant les transactions possibles sur la base des données du marché et des indicateurs d'analyse technique.

Comme l'expression l'indique, l'allocation des risques consiste pour le robot à répartir les risques en fonction d'un ensemble spécifique de paramètres et de règles fixés par le trader, ce qui inclut généralement la manière et l'ampleur de l'allocation du capital lors des transactions.

C'est le moment de l'exécution. L'exécution est l'étape au cours de laquelle les crypto-monnaie sont effectivement achetées et vendues sur la base des signaux générés par le système de trading préconfiguré. À ce stade, les signaux seront convertis en demandes de clés API que la bourse de crypto-monnaies peut comprendre et traiter.

HYPOTHÈSE

Dans ce projet, nous nous focaliserons uniquement sur la stratégie de deep learning qu'utilisait notre de bot. Nous présenterons une approche simple, pédagogique et illustrée de l'implémentation du réseau neuronal utilisé.

C'est pourquoi nous ne nous attarderons pas sur l'infrastructure du projet, qui permet au

bot de communiquer avec une plateforme de change (le connecteur) et sur le moteur capable de le faire tourner ainsi que d'exécuter des stratégies.

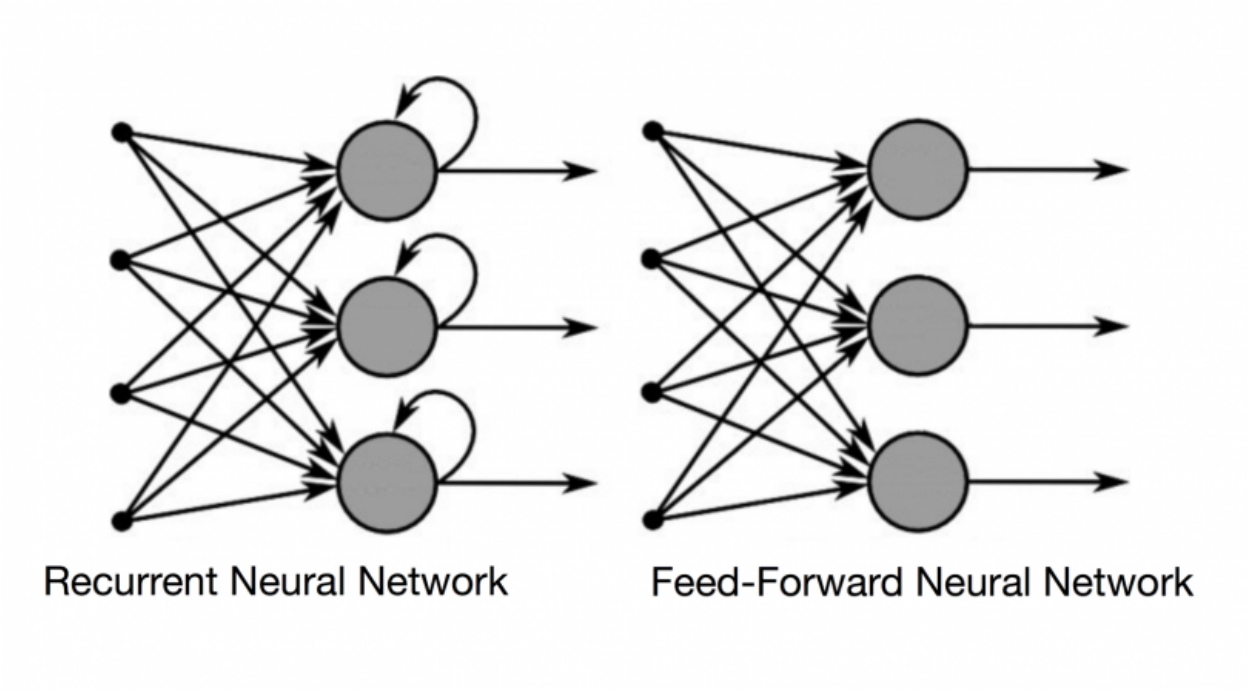
STRATÉGIE

Afin de mettre en place notre stratégie, nous utiliserons un réseau neuronal récurrent (RNN) afin de guider nos ordres de vente ou d'achats. Avant de rentrer en détail dans le code, faisons un tour d'horizon du fonctionnement des RNN.

Les RNN sont un type de réseau neuronal puissant et robuste, et font partie des algorithmes les plus prometteurs en usage car c'est le seul à disposer d'une mémoire interne.

Comme de nombreux autres algorithmes d'apprentissage profond, les réseaux de neurones récurrents sont relativement anciens. Ils ont été créés dans les années 1980, mais ce n'est qu'au cours des dernières années que nous avons vu leur véritable potentiel. L'augmentation de la puissance de calcul, les quantités massives de données avec lesquelles nous devons désormais travailler et l'invention de la mémoire à long terme (LSTM) dans les années 1990 ont réellement mis les RNN au premier plan.

Grâce à leur mémoire interne, les RNN peuvent se souvenir d'éléments importants concernant les données qu'ils ont reçues, ce qui leur permet d'être très précis dans la prédiction de ce qui va suivre. Ils sont très utilisés pour le traitement des données séquentielles telles que les séries temporelles, la parole, le texte, les données financières, l'audio, la vidéo, la météo et bien plus encore. Les réseaux neuronaux récurrents peuvent acquérir une compréhension beaucoup plus profonde d'une séquence et de son contexte que les autres algorithmes.



Dans un FFNN, l'information ne se déplace que dans un sens : de la couche d'entrée à la couche de sortie, en passant par les couches cachées. L'information se déplace directement dans le réseau et ne touche jamais deux fois un nœud.

Les Feed forward neural network n'ont aucune mémoire des données qu'ils reçoivent et sont incapables de prédire ce qui va se passer. Comme un réseau feed-forward ne prend en compte que l'entrée actuelle, il n'a aucune notion de l'ordre dans le temps. Il ne peut tout simplement pas se souvenir de ce qui s'est passé dans le passé, sauf de sa formation.

Dans un RNN, les informations circulent dans une boucle. Lorsqu'il prend une décision, il tient compte de l'entrée actuelle et de ce qu'il a appris des entrées qu'il a reçues précédemment.

Par conséquent, un RNN a deux entrées : le présent et le passé récent. C'est important car la séquence de données contient des informations cruciales sur ce qui va suivre.

Un réseau neuronal à action directe attribue, comme tous les autres algorithmes d'apprentissage profond, une matrice de poids à ses entrées, puis produit la sortie. Les RNN appliquent des poids à l'entrée actuelle et également à l'entrée précédente. En outre, un réseau neuronal récurrent modifiera également les poids pour les deux par descente de gradient et par rétropropagation dans le temps (BPTT).

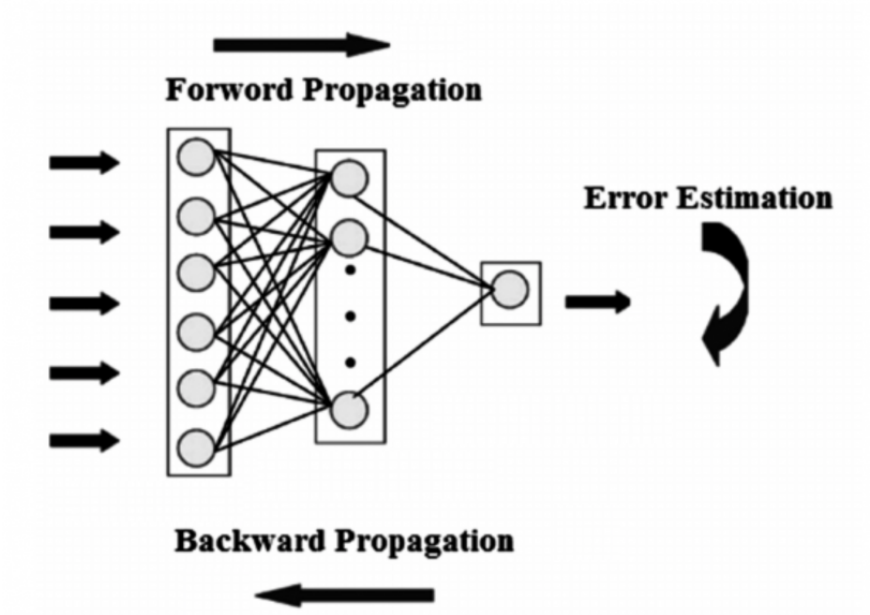
Dans les réseaux neuronaux, nous faisons essentiellement de la propagation en avant

pour obtenir la sortie d'un modèle et vérifier si cette sortie est correcte ou incorrecte, pour obtenir l'erreur. La rétropropagation n'est rien d'autre que le fait de revenir en arrière dans notre réseau neuronal pour trouver les dérivées partielles de l'erreur par rapport aux poids, ce qui nous permet de soustraire cette valeur des poids.

Ces dérivées sont ensuite utilisées par la descente de gradient, un algorithme capable de minimiser itérativement une fonction donnée. Il ajuste ensuite les poids vers le haut ou vers le bas, en fonction de ce qui diminue l'erreur. C'est de cette manière qu'un réseau neuronal apprend au cours du processus de formation.

Ainsi, avec la rétropropagation, nous essayons essentiellement d'ajuster les poids de notre modèle pendant la formation.

L'image ci-dessous illustre le concept de propagation directe et de rétropropagation dans un réseau neuronal à action directe :

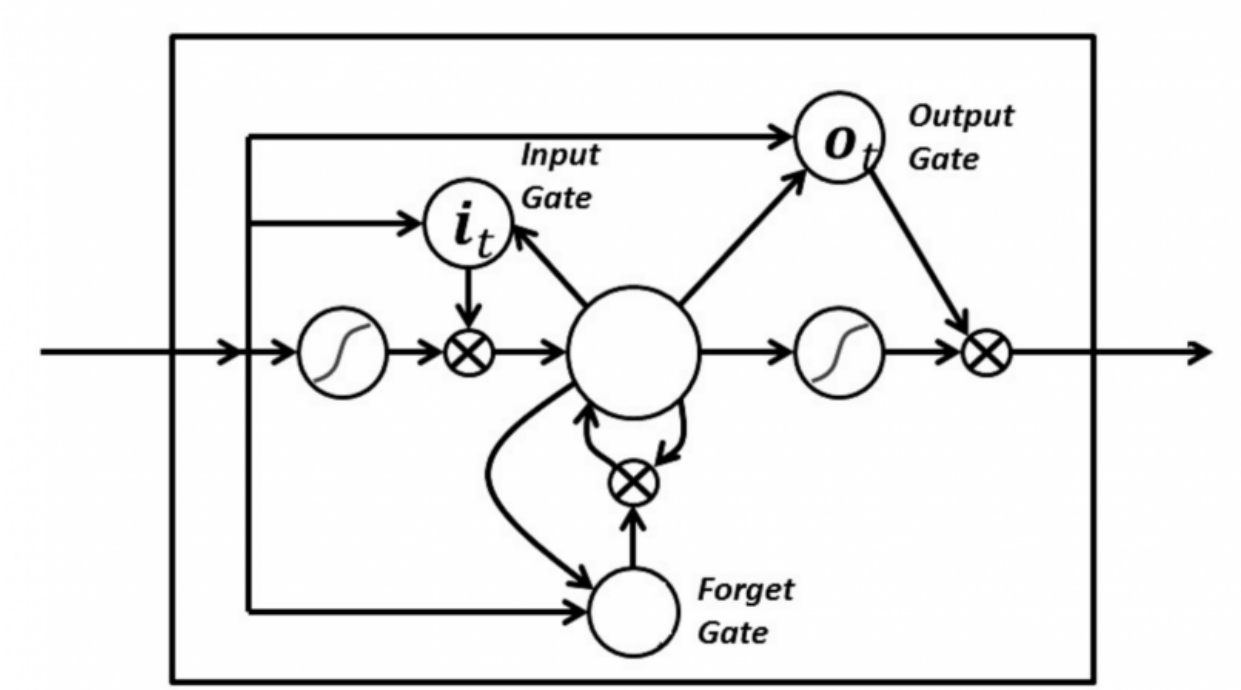


Réseau neuronal LSTM

Notre Bot utilisera un RNN LSTM (Long short term memory) via Tensorflow et Keras.

Tensorflow est une bibliothèque open source de machine learning crée par Google, elle permet de développer et d'éxécuter des application Machine learning et Deep Learning. Keras est l'API de haut niveau de TensorFlow permettant de créer et d'entraîner des modèles de deep learning. Elle est utilisée dans le cadre du prototypage rapide, de la recherche de pointe et du passage en production.

Les réseaux de mémoire à long terme sont une extension des réseaux neuronaux récurrents, qui étendent essentiellement la mémoire.
Les LSTM permettent aux RNN de se souvenir des entrées sur une longue période de temps. Cela est dû au fait que les LSTM contiennent des informations dans une mémoire, un peu comme la mémoire d'un ordinateur. Le LSTM peut lire, écrire et supprimer des informations de sa mémoire.
Cette mémoire peut être considérée comme une cellule à portes, ce qui signifie que la cellule décide de stocker ou de supprimer des informations (c'est-à-dire d'ouvrir ou non les portes), en fonction de l'importance qu'elle accorde à ces informations. L'attribution de l'importance se fait par le biais de poids, qui sont également appris par l'algorithme. Cela signifie simplement qu'il apprend au fil du temps quelles informations sont importantes et lesquelles ne le sont pas.Dans un LSTM, il y a trois portes : entrée, oubli et sortie. Ces portes déterminent s'il faut ou non laisser entrer une nouvelle entrée (porte d'entrée), supprimer l'information parce qu'elle n'est pas importante (porte d'oubli) ou la laisser avoir un impact sur la sortie au moment présent (porte de sortie). Voici une illustration d'un RNN avec ses trois portes :



Implémentation

Les données que nous utiliserons sont les données Open, High, Low, Close, Volume pour Bitcoin, Ethereum, Litecoin et Bitcoin Cash.
Pour nos besoins ici, nous allons seulement nous concentrer sur les colonnes Close et Volume. La colonne Close mesure le prix final à la fin de chaque intervalle, dans ce cas, il s'agit d'intervalles d'une minute.
La colonne Volume indique le volume de l'actif négocié dans chaque intervalle, dans ce cas, par minute également.

Avant de commencer à entrainer notre modèle il est nécessaire de travailler sur notre set de data (balance, scale et normalisation).

Afin de lire le set de data en csv nous utilisons la librairie pandas.

Ensuite, nous devons créer une target. Pour ce faire, nous devons savoir quel prix nous essayons de prévoir. Nous devons également savoir quelle durée nous voulons prédire. Nous choisirons le Litecoin pour l'instant. Savoir à quelle distance nous voulons prédire dépend probablement aussi de la longueur de nos séquences. Si la longueur de notre

séquence est de 3 (minutes), nous ne pouvons probablement pas prédire facilement 10 minutes. Si la longueur de notre séquence est de 300, 10 ne sera peut-être pas aussi difficile. Nous optons ici pour une longueur de séquence de 60, et une prédiction future de 3. Nous pourrions également faire de la prédiction une question de régression, en utilisant une activation linéaire avec la couche de sortie, mais, à la place, nous optons simplement pour une classification binaire.

Si le prix monte dans 3 minutes, alors c'est un achat. S'il baisse en 3 minutes, ce n'est pas un achat/vente.

```
######## ici on def notre projection #######
SEQ_LEN = 60  # la longueur de la séquence précédente à collecter pour le RNN
FUTURE_PERIOD_PREDICT = 3  # à quelle distance dans le futur essayons-nous de
prédire ?
RATIO_TO_PREDICT = "LTC-USD"
```

Nous séparons ensuite nos données de validation/hors échantillon. D'habitude il est commun de mélanger les données avec shuffle puis de les découper. Cependant, dans notre cas cela n'a pas de sens. Le problème avec cette méthode est que les données sont intrinsèquement séquentielles, donc prendre des séquences qui ne viennent pas dans le futur est probablement une erreur. En effet, dans notre cas, des séquences espacées d'une minute seront presque identiques. Il y a de fortes chances que l'objectif soit également le même (achat ou vente). De ce fait, toute suradaptation est susceptible de se répercuter sur l'ensemble de validation. Au lieu de cela, nous voulons trancher notre validation pendant qu'elle est encore en ordre. Nous prenons donc les 5 derniers % des données.

```
###### ici on décpoupe nos data en deux, 95% pour l'entrainement et les 5%
derniers pour la validation #######
times = sorted(main_df.index.values)
last_5pct = sorted(main_df.index.values)[-int(0.05*len(times))]  # on prend les 5%
validation_main_df = main_df[(main_df.index >= last_5pct)]  # index
main_df = main_df[(main_df.index < last_5pct)]  # maintenant le main_df est toutes
les données jusqu'au dernier 5%.
```

Ensuite, nous devons équilibrer et normaliser ces données. Nous devons également prendre nos données et en faire des séquences.

```
def preprocess_df(df):
    df = df.drop("future", 1)

    for col in df.columns:  # on itère sur nos colonnes
    ######## on normalise nos datas (une mise à l'échelle des cours des monnaies
    entre elle) ########
        if col != "target":  #on normalize tout sauf la target
            df[col] = df[col].pct_change()  # La variation en pourcentage
            "normalise" les différentes monnaies (chaque crypto-monnaie a des
            valeurs très différentes), on s'intèresse à leur variation relative.
            df.dropna(inplace=True)  # sup les nans du au pct_change
            df[col] = preprocessing.scale(df[col].values)  # scale entre 0 et 1.
```

```
sequential_data = []  #list contenant les séquences
prev_days = deque(maxlen=SEQ_LEN)  # Séquences actuelles. On utilise deque qui
garde la longueur max en sortant les vieilles valeurs quand les nouvelles
rentres
######## on séquence nos datas en plusieurs lot de 60 secondes ########
for i in df.values:  # iterate over the values
    prev_days.append([n for n in i[:-1]])
    if len(prev_days) == SEQ_LEN:  # on vérifie qu'on a 60 seq
        sequential_data.append([np.array(prev_days), i[-1]])

random.shuffle(sequential_data)  # on mélange le seq data
```

Nous avons normalisé et mis à l'échelle nos données. Ensuite, nous voulons les équilibrer. Toujours dans notre fonction preprocess_df, nous pouvons équilibrer en faisant:

```
random.shuffle(sequential_data)  # on mélange le seq data

buys = []  # liste de toutes nos séquences d'achats et nos targets
sells = []  # liste de toutes nos séquences de vente et nos targets

for seq, target in sequential_data:  # parcourt notre lot de datas
    if target == 0:  # si c'est une vente
        sells.append([seq, target])  # on ajoute à la liste des ventes
    elif target == 1:  # sinon c'est un ordre d'achat
        buys.append([seq, target])  # on ajoute à la liste des achats
# on mélange nos listes #
random.shuffle(buys)
random.shuffle(sells)

lower = min(len(buys), len(sells))  # on récupère la plus courte?
#on s'assure que les deux listes sont seulement jusqu'à la longueur la plus courte#
#pour avoir des listes de même taille#
buys = buys[:lower]
sells = sells[:lower]

sequential_data = buys+sells  # additionne nos listes
random.shuffle(sequential_data)  # on mélange pour que le modèle ne soit pas embrouillé
avec une classe puis une autre

X = []
y = []

for seq, target in sequential_data:  # parcourt notre nouvelle sequence de datas
    X.append(seq)  # X = sequences
    y.append(target)  # y = targets  (achat ou vente)

return np.array(X), y
```

Une fois cette fonction terminée, nous pouvons maintenant prétraiter nos données avec :

```
train_x, train_y = preprocess_df(main_df) # nos datas d'entrainement
validation_x, validation_y = preprocess_df(validation_main_df) #nos datas de test
```

De plus, nous affichons quelques tests pour vérifier le bon fonctionnement de notre fonction preprocess_df.

Nous vérifions la cohérence de taille entre notre tableau d'entraînement et de test. (Nous nous attendons à avoir un tableau d'entraînement nettement plus grand, 95% des données sont pour le training). Et nous vérifions que les tableaux retournés par

preprocess_df sont de même taille.

```
print(f"train data: {len(train_x)} validation: {len(validation_x)}")
print(f"Dont buys: {train_y.count(0)}, buys: {train_y.count(1)}")
print(f"VALIDATION Dont buys: {validation_y.count(0)}, buys: {validation_y.count(1)}")
```

Avant d'implémenter les dernières étapes de notre modèle, nous définissons la taille de nos échantillons (batch), le nombre de total du parcours de nos données (epoch), le nombre de couches de neurones, ainsi que le nombre de nœuds/neurones.

Dans notre simulation nous ferons 10 epoch et 64 batchs. (nous nous sommes limité à cette taille pour avoir un temps de compilation raisonnable par la suite). Et notre réseau neuronal est constitué de 3 couches, et 32 nœuds (par couches).

En outre, nous allons importer les libraires de tensorflow adapté à notre modèle. Ici nous prenons un modèle Sequential,car c'est le plus approprié pour une pile simple de couches où chaque couche a exactement un tenseur d'entrée et un tenseur de sortie.

Pour nos couches de neurones nous importons les librairies : **Dense** (fonction d'activation pour une couche de neurone), **Dropout** (pour éviter l'overfitting lors de phases d'apprentissage), **LSTM** (Long short term memory), **BatchNormalization** (permet la normalisation des sortie de chaque couche de neurones).

```
import tensorflow as tf
from tensorflow.keras.models import Sequential #type de modèle adapté à notre série
from tensorflow.keras.layers import Dense, Dropout, LSTM, BatchNormalization
```

Après avoir défini nos couches de neurones, nous choisissons l'optimisation d'Adam pour optimiser nos sorties. C'est une méthode de descente de gradient stochastique qui repose sur l'estimation adaptative des moments de premier et de second ordre.

```
#### definition de notre réseau de neurones ####
model = Sequential() # création du modele
# création 1ere couche de neuronnes #
# 32 noeuds de type : Long Short-Term Memory
model.add(LSTM(32, input_shape=(train_x.shape[1:]), return_sequences=True))
model.add(Dropout(0.2)) #La couche Dropout place aléatoirement les unités d'entrée à 0 avec une
fréquence de taux à chaque étape de la formation, ce qui permet d'éviter un overfitting.Dropout
s'applique que lorsque l'apprentissage est défini sur True.
model.add(BatchNormalization())#La normalisation des lots applique une transformation qui maintient
la sortie moyenne proche de 0 et l'écart type de la sortie proche de 1.

model.add(LSTM(32, return_sequences=True))
model.add(Dropout(0.1))
model.add(BatchNormalization())

model.add(LSTM(32))
model.add(Dropout(0.2))
model.add(BatchNormalization())

model.add(Dense(32, activation='relu')) #fonction d'activation par éléments passée comme argument
d'activation, ici "rectified linear"
model.add(Dropout(0.2))

model.add(Dense(2, activation='softmax')) #dense layer pour l'output, 2 car choix binaire, et
activation de la couche de sortie à softmax

####descente de gradient stochastique####
opt = tf.keras.optimizers.Adam(lr=0.001, decay=1e-6)
```

Une fois tous les paramètres de notre modèle neuronal définis, nous n’avons plus qu'à compiler et entraîner notre réseau de neurones.

```
####Compilation du model ####
model.compile(
    loss='sparse_categorical_crossentropy',
    optimizer=opt,
    metrics=['accuracy']
)
####training de notre modele####
# définition des variables avec les données
train_x = np.asarray(train_x)
train_y = np.asarray(train_y)
validation_x = np.asarray(validation_x)
validation_y = np.asarray(validation_y)
#lancement de l'entrainement
history = model.fit(
    train_x, train_y,
    batch_size=BATCH_SIZE,
    epochs=EPOCHS,
    validation_data=(validation_x, validation_y),
    callbacks=[tensorboard, checkpoint],
)
```

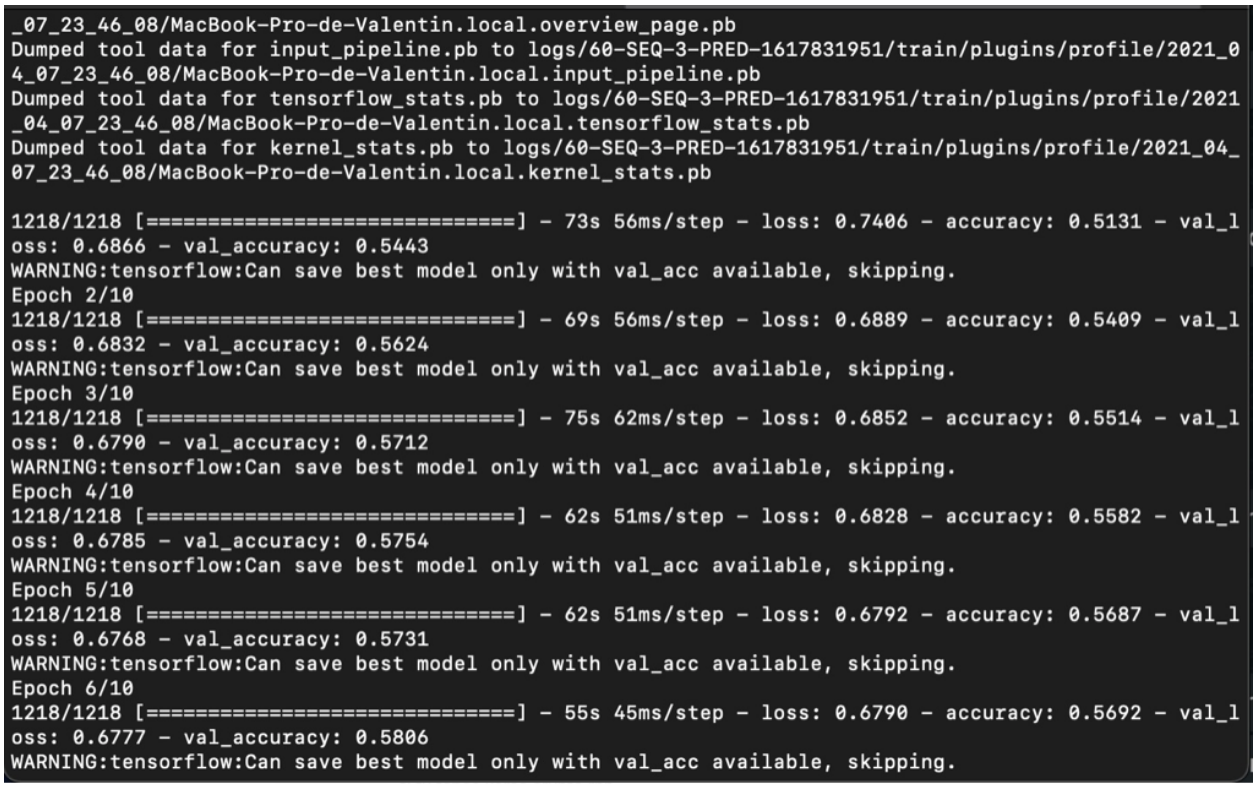
Nous vous invitons à regarder l'implémentation complète et fonctionnelle de réseau neuronal pouvant servir de base fonctionnelle pour un bot de trading.

Nous avons aussi joint une banque de données au format .csv permettant l'entraînement du RNN de notre bot.

RÉSULTATS

Interprétation des résultats (qualités des prédictions, marge d'erreur):

Voici les résultat de notre implémentation :



```
_07_23_46_08/MacBook-Pro-de-Valentin.local.overview_page.pb
Dumped tool data for input_pipeline.pb to logs/60-SEQ-3-PRED-1617831951/train/plugins/profile/2021_0
4_07_23_46_08/MacBook-Pro-de-Valentin.local.input_pipeline.pb
Dumped tool data for tensorflow_stats.pb to logs/60-SEQ-3-PRED-1617831951/train/plugins/profile/2021
_04_07_23_46_08/MacBook-Pro-de-Valentin.local.tensorflow_stats.pb
Dumped tool data for kernel_stats.pb to logs/60-SEQ-3-PRED-1617831951/train/plugins/profile/2021_04_
07_23_46_08/MacBook-Pro-de-Valentin.local.kernel_stats.pb

1218/1218 [==============================] - 73s 56ms/step - loss: 0.7406 - accuracy: 0.5131 - val_l
oss: 0.6866 - val_accuracy: 0.5443
WARNING:tensorflow:Can save best model only with val_acc available, skipping.
Epoch 2/10
1218/1218 [==============================] - 69s 56ms/step - loss: 0.6889 - accuracy: 0.5409 - val_l
oss: 0.6832 - val_accuracy: 0.5624
WARNING:tensorflow:Can save best model only with val_acc available, skipping.
Epoch 3/10
1218/1218 [==============================] - 75s 62ms/step - loss: 0.6852 - accuracy: 0.5514 - val_l
oss: 0.6790 - val_accuracy: 0.5712
WARNING:tensorflow:Can save best model only with val_acc available, skipping.
Epoch 4/10
1218/1218 [==============================] - 62s 51ms/step - loss: 0.6828 - accuracy: 0.5582 - val_l
oss: 0.6785 - val_accuracy: 0.5754
WARNING:tensorflow:Can save best model only with val_acc available, skipping.
Epoch 5/10
1218/1218 [==============================] - 62s 51ms/step - loss: 0.6792 - accuracy: 0.5687 - val_l
oss: 0.6768 - val_accuracy: 0.5731
WARNING:tensorflow:Can save best model only with val_acc available, skipping.
Epoch 6/10
1218/1218 [==============================] - 55s 45ms/step - loss: 0.6790 - accuracy: 0.5692 - val_l
oss: 0.6777 - val_accuracy: 0.5806
WARNING:tensorflow:Can save best model only with val_acc available, skipping.
```

Premièrement nous avons constaté que deux approches sont possibles dans l'installation de la librairie tensorflow, l'un en GPU et l'autre en CPU. La différence majeure entre ces deux versions demeure dans la performance nettement supérieure de la GPU par rapport à la CPU.

De plus, nous constatons une réduction du temps de parcours au fils des epochs, dû à l'entraînement du réseau neuronal (de 73 à 55s en 6 epoch). La perte est aussi en diminution (de 0.74 à 0.67) au fil des entraînements.

Cependant la précision de nos prédictions restent autour de 0.5 et augmentent faiblement (de 0.51 à 0.56). Nous pensons qu'un plus grand nombre

d'entraînement/epoch serait nécessaire pour atteindre une précision significativement meilleure.

CONCLUSION

Ce projet nous a permis de mettre en pratique et d'approfondir nos connaissances sur les réseaux neuronaux récurrents. Nous avions été initié aux fonctionnements des réseaux neuronaux tel que les réseaux FFNN lors du premier semestre, c'est pourquoi nous avons décidé d'implémenter un bot utilisant un réseau neuronal similaire pour notre algorithme de trading sur cryptomonnaie.

De plus, à travers ce projet nous avons découvert et nous sommes familiarisé avec la bibliothèque tensor flow permettant une application plus simple des principes du machine learning dans notre projet.

En définitive, ce projet nous a permis d'avoir une approche concrète des applications du machine learning dans le domaine de la finance.

RÉFÉRENCES

https://www.datacamp.com/community/tutorials/tensorboard-tutorial

https://www.tensorflow.org/guide/keras/rnn

https://www.kaggle.com/fedewole/algorithmic-trading-with-keras-using-lstm

https://www.kdnuggets.com/2018/11/keras-long-short-term-memory-lstm-model-predict-stock-prices.html