Challenge Data : Prédiction des volumes de transactions boursières

Rapport de projet

THOMAS MESNARD [1]
Département d'informatique
École Normale Supérieure de Paris
thomas.mesnard@ens.fr

Résumé

Des milliards de dollars sont échangés chaque jour dans les marchés financiers américains. Ces échanges ne se déroulent pas de manière aléatoire et il est possible déceler des schémas complexes de corrélation durant ces échanges. Le but du projet est de découvrir et quantifier ces corrélations pour prévoir au mieux l'évolution des échanges.

Challenge proposé par Capital Fund Management
Cours : *"Représentations Parcimonieuses en Ondelettes"*, par Stéphane Mallat
Pour le Master 2 Mathématiques, Vision et Apprentissage (MVA) de l'ENS de Cachan.

1. Les implémentations réalisées durant ce projet sont disponibles en libre accès à cette adresse https://github.com/thomasmesnard/CFM, sous licence MIT.

Table des matières

1 Description du projet

1.1 Les données

1.1.1 Vue d'ensemble

Chaque exemple contient : un identifiant qui lui est propre, la date des transactions, l'identifiant du produit considéré et pour finir une série temporelle de valeurs. Cette série correspond aux volumes échangés chaque demie heure entre 9h et 14h. Nous devons par la suite prédire le volume de transactions qui sera échangé entre 14h et 16h. L'ensemble d'entraînement est composé de 618 556 exemples pouvant correspondre à 352 types de produits boursiers.

Pour entraîner nos modèles et pouvoir les tester, nous allons découper notre ensemble de données en deux sous-parties : une partie pour l'entraînement et une partie pour la validation. L'ensemble de validation représente 10% de l'ensemble des données. Précisons ici que les données de validation ne seront bien sur pas utilisées lors de l'entraînement.

1.1.2 Problèmes rencontrés

Grande variabilité des moyennes La première chose frappante en observant les données est la grande variabilité des ordres de grandeur que peuvent prendre les séries temporelles. En effet, en fonction du produit boursier, les volumes échangés peuvent varier de 8 ordres de grandeurs. Dans la perspective de l'utilisation de réseaux de neurones artificiels cela peut être très problématique. Bien que très performants, ces réseaux ne peuvent pas traiter naturellement une telle gamme de valeurs. Un deuxième problème dû à cette large gamme de valeur est lié à la fonction de coût dont nous parlerons plus amplement dans la section 1.2 .

Valeurs manquantes Dans la majorité des cas seul un petit nombre de valeur manque, mais cela peut varier largement d'un exemple à un autre, et dans certain cas la série temporelle est presque entièrement manquante. Cela pose un problème évident que nous devrons régler en développant une technique adaptée pour ces cas particuliers.

1.1.3 Techniques de prétraitement

Normalisation Comme nous l'avons évoqué dans la section 1.1.2, nous devons trouver une solution pour gérer la grande variabilité des ordres de grandeur des séries temporelles. Pour cela, nous avons utilisé deux techniques pour normaliser nos données :

— Pour chaque exemple, normalisation de la série temporelle sans prendre en compte le type de produit dont il est question.
— Normalisation de la série temporelle en fonction du type de produit considéré. On normalise alors la série à l'aide de la moyenne et de l'écart type de l'ensemble des exemples d'entraînements pour ce produit et non plus seulement avec la moyenne et l'écart type de cet exemple en particulier.

Après de nombreux tests, nous avons observé que la première approche, bien plus simple, fournit de meilleurs résultats pour tous les modèles implémentés. Cela sous-entend qu'il existe une certaine variabilité dans l'ordre de grandeur des séries temporelles y compris pour un même type de produit.

Techniques de remplacement des valeurs manquantes Dans le cas de données manquantes dans les séries temporelles, plusieurs cas se présentent :

— S'il existe une valeur avant et après la valeur manquante, nous remplaçons la valeur manquante par la moyenne de la valeur précédente et suivante.
— Dans le cas où au moins un des voisins directs n'est pas présent, on remplace la valeur manquante par la moyenne de la valeur la plus proche avant et de la valeur la plus proche après.
— Dans le cas où toutes les valeurs sont manquantes, on prend la valeur moyenne prise dans les séries temporelles pour le produit en question dans l'ensemble des autres exemples.

Dans le cas des réseaux de neurones, il est important de fournir également un vecteur composé de 1 et 0 indiquant l'emplacement des valeurs manquantes, le cas échéant.

Ordre aléatoire d'apparition des exemples Pour assurer une distribution homogène des exemples dans nos données d'entraînement, nous présentons les exemples dans un ordre aléatoire.

1.2 Fonction de coût

La fonction de coût proposée par CFM est Mean Absolute Percentage Error. Si on note y la valeur réelle attendue et $\hat{y}$ la valeur prédite par notre modèle, le coût est défini par :

$$coût(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} \frac{|y-\hat{y}|}{y}$$

Cette fonction de coût pose un problème évident, problème qui est exacerbé par la large gamme de valeur que peuvent prendre les séries temporelles. Dans le cas où le modèle prédit 10 000 à la place de la valeur 1, l'erreur va être proche de 10 000. Les données sont très majoritairement composées de grosses valeurs et de quelques petites valeurs qui risquent fortement de faire exploser le coût. Ceci semble vraiment problématique et va être un frein certain à l'entraînement des modèles.

1.3 Implémentation des modèles

L'implémentation des différents modèles de réseaux de neurones artificiels a été réalisés à l'aide de Theano, Blocks et Fuel qui sont des frameworks Python développés à l'Université de Montréal par l'équipe de Yoshua Bengio. L'implémentation du modèle utilisant des forêt d'arbres décisionnels a été réalisée grâce au framework Python Scikit-learn. L'ensemble des implémentations sont disponibles à cette adresse : `https://github.com/thomasmesnard/CFM`. L'ensemble des entraînements des réseaux de neurones a été réalisé sur des processeurs graphiques (GPU) afin de paralléliser les calculs et donc d'accélérer grandement l'entraînement de ces réseaux.

2 Forêt d'arbres décisionnels

La première méthode que nous avons implémenté est une méthode par forêt d'arbres décisionnels. Les arbres de décisions peuvent non seulement répondre à des problèmes de classification mais ils peuvent également répondre à des problèmes de régression.

Revenons sur le principe des forêts d'arbres décisionnels. Si nous avons p modalités d'une variable, elles peuvent être décomposées en $p-1$ variables binaires. On peut donc ramener des variables quantitatives à des variables binaires. Pour repasser rapidement sur le mécanisme de cette méthode, prenons un exemple. Supposons que les variables sont à valeurs dans $[0,1]^2$. On peut par exemple chercher à prédire si le point en question est de classe rouge ou de classe bleue en fonction des deux variables quantitatives comprises dans [0, 1]. Pour cela, il est possible de construire des partitions de plus en plus fines de l'ensemble $[0,1]^2$. L'arbre de décision associé à cette découpe correspond à l'ordre des partitions ainsi réalisées. La figure suivante illustre ce mécanisme :

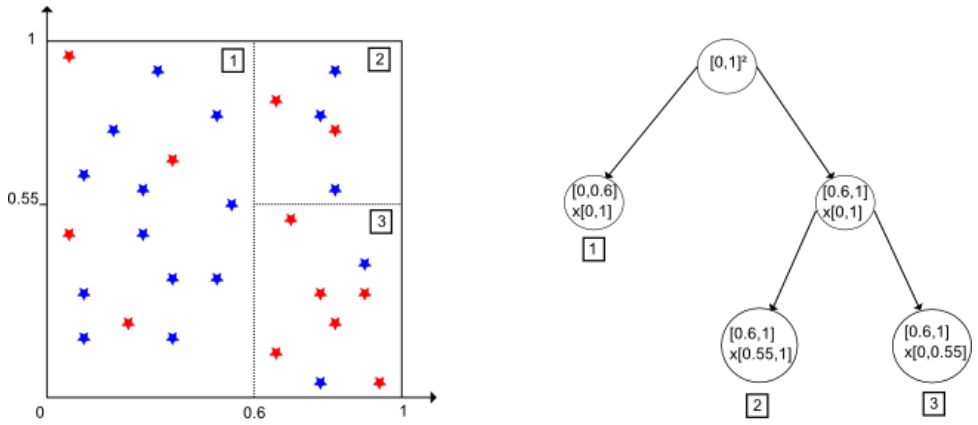

FIGURE 1 – Correspondance entre partition de l'espace des variables et arbre décisionnel. Crédits à Erwan Scornet, 2011

	1	**2**	**3**
Nombre d'arbres	10	40	200
Erreur sur la validation	**0.81**	**0.75**	**0.70**

Avec une forêt de 200 arbres, nous sommes parvenus à une erreur minimale de 70% MAPE.

3 Apprentissage profond

3.1 Généralités

L'apprentissage profond est un domaine en plein expansion qui ne cesse d'améliorer l'état de l'art dans de nombreuses tâches classiques en apprentissage automatique. Avant de parler de perceptron multicouche ou encore de réseau de neurones récurrents, nous allons rapidement parler ici des techniques propres à l'entraînement de ces réseaux aussi bien au niveau des méthodes de régularisation que des méthodes de descente de gradient. L'ensemble des modèles ont été entraînés par rétropropagation du gradient.

Différents algorithmes de descente de gradient stochastique peuvent être appliqués lors de l'entraînement des réseaux profonds. Un bon choix peut permettre d'accélérer considérablement la phase d'entraînement. Parmi les algorithmes classiques on peut citer la SGD (*stochastic gradient descent*) ou encore RMS-Prop, AdaGrad, AdaDelta, ... qui sont autant de variantes de la SGD.

Une astuce souvent utilisée pour entraîner les réseaux est l'ajout d'un *momentum* en complément de l'algorithme de descente de gradient. On ajoute alors une fraction du précédent pas de descente de gradient dans le pas actuel. Cela permet une convergence plus rapide.

Nous avons également utiliser un *gradient clipping*. Cette technique consiste à ajouter une borne supérieure aux valeurs que peuvent prendre le gradient. Cela évite en cas d'explosion du gradient d'appliquer aux paramètres un énorme pas dans le sens opposé au gradient et détruisant ainsi l'ensemble des petits pas réalisés auparavant. Cette technique est particulièrement intéressante dans notre cas car comme nous l'avons souligné précédemment la fonction de coût peut parfois poser problème et cela limitera les dommages causés par une explosion momentanée du coût (ce qui arrive en pratique relativement souvent).

Il est également nécessaire de choisir un bon learning rate. Un bon learning rate permettra certes un rapide entraînement du réseau mais devra rester suffisamment petit pour pouvoir converger vers un minimum global et ne pas passer de vallée de stabilité en vallée de stabilité. Face aux problèmes liés à la fonction de coût, nous avons utilisé des learning rates assez petits.

Nous avons également utilisé une méthode qui consiste à diminuer la valeur du learning rate à chaque itération de l'algorithme. On veut ainsi pouvoir garder un learning rate important en début d'entraînement pour aller rapidement dans une bonne vallée de stabilité mais au fur et à mesure, diminuer ce learning rate pour pouvoir réaliser des pas plus précis et pouvoir ainsi converger vers une meilleure solution.

De plus, deux techniques de régularisation ont été utilisées.

— L'utilisation de bruit blanc sur les poids synaptiques permet de rendre le réseau plus résistant à de petites fluctuations dans les entrées.

— Une technique plus récente est l'utilisation de *dropout*. À chaque instant, on rend inactif une proportion aléatoire des neurones de notre réseau. Cela permet de rendre l'apprentissage redondant dans le réseau et le rend ainsi plus robuste. Cela pousse le réseau à avoir une compréhension intrinsèque et robuste des données.

Pour finir, plusieurs fonctions d'activation peuvent être utilisées dans un tel réseau : tangente hyperbolique, sigmoïde, rectifier ($x \mapsto \max(0, x)$). Par exemple, il a été montré que les

fonctions rectifier étaient particulièrement efficaces dans le cas des structures très profondes et que les fonctions tangentes hyperboliques étaient adaptées aux réseaux de neurones récurrents.

3.2 Perceptron multicouche

Le perceptron multicouche est un des modèles les plus simples utilisé dans le domaine de l'apprentissage profond. Il consiste en une succession de couches de neurones reliées entre eux par des poids synaptiques. Chaque neurone somme l'ensemble de ses entrées pondérées par les poids synaptiques associés puis applique une fonction d'activation. La dernière couche correspond à la sortie du réseau. Voici le schéma récapitulatif d'un tel réseau :

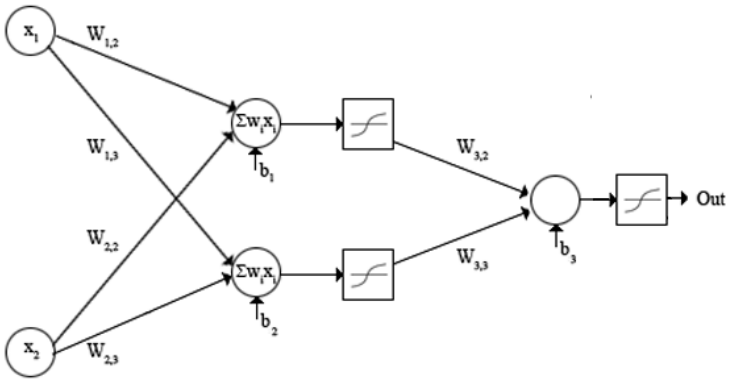

FIGURE 2 – Perceptron multicouche (crédits à `http://matlabgeeks.com`)

Nous avons fourni à notre réseau un vecteur d'entrée contenant pour chaque exemple :

— L'identifiant du produit en question transformé en un vecteur de 1 et de 0, bien plus facilement assimilable par le réseau.
— L'ensemble des volumes échangés à chaque demie heure.
— Un vecteur de 0 et 1 indiquant si la valeur étant manquante initialement ou non.
— La moyenne des volumes échangés entre 9h et 14h.
— L'écart type des volumes échangés entre 9h e 14h.

Nous avons réalisé de nombreuses recherches d'hyperparamètres permettant de diminuer l'erreur sur l'ensemble de validation. L'ensemble des résultats collectés se trouve en Annexe 1. Voici un tableau montrant deux modèles testés dont nous allons discuter :

Caractéristiques	**1**	**2**
Nombre de couche cachée	3	2
Nombre de neurones par couche	50	200
Fonction d'activation	Tanh	Rectifier
Taille des batchs	400	10
Bruit blanc sur les poids	0	0.01
Dropout	0	0.5
Proportion train/valid	0.9	0.9
Algorithme de descente de gradient	RMS-Prop	RMS-Prop
Learning rate	10^{-5}	10^{-5}
Momentum	0.9	0.9
Step clipping	2	2
Décroissance du learning rate	1	1
Erreur sur la validation	**0.31**	**0.50**

Le modèle 1 correspond au perceptron multicouche le plus performant que nous avons trouvé. Voici l'erreur MAPE en fonction de l'itération pour la partie d'entraînement et la partie de validation durant l'entraînement du modèle :

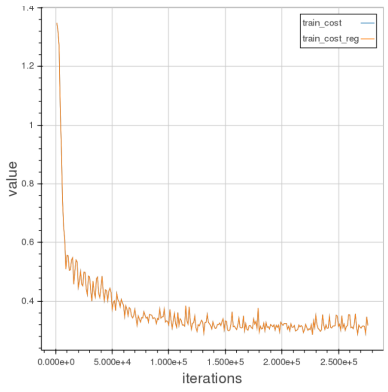

FIGURE 3 – Évolution du coût au cours de l'entraînement sur l'ensemble d'entraînement

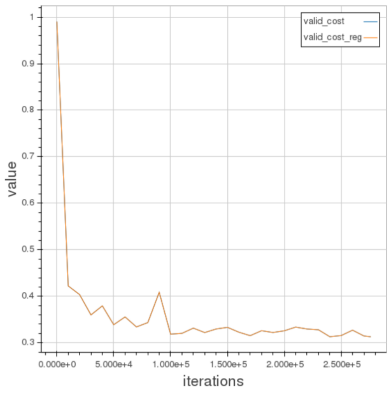

FIGURE 4 – Évolution du coût au cours de l'entraînement sur l'ensemble de validation

Nous pouvons pu remarquer lors de la recherche d'hyperparamètres que :
— Le réseau doit être d'une profondeur contenue entre 2 et 3 couches cachées.
— La fonction tangente hyperbolique permet d'obtenir de meilleurs résultats que la fonction rectifier.
— Un nombre faible de neurones par couche (50) permet d'obtenir des résultats optimaux.
— Un dropout trop important rend l'entraînement difficile et le système ne parvient plus à converger.
— Un faible learning rate est important pour permettre au réseau de converger.
— Une grande taille de batch permet une forte accélération de l'entraînement et une meilleure convergence.

3.3 Réseaux de neurones récurrents

Dans cette section nous allons nous intéresser à des réseaux de neurones récurrents. À la différence d'un perceptron multicouche, l'état d'un neurone dépend de son propre état au

temps précédent, en plus d'être déterminé par l'état des neurones dans les couches précédentes. Ces modèles sont particulièrement adaptés pour traiter des séries temporelles de données, ce qui est notre cas. Voici un schéma représentant un modèle simple de RNN :

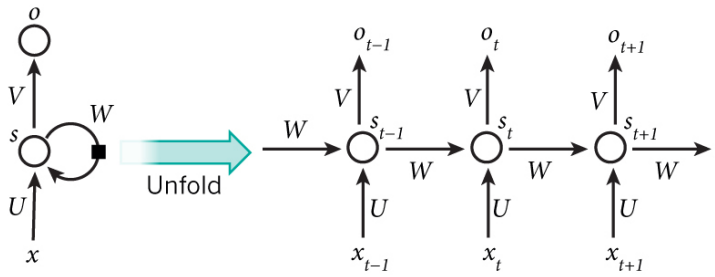

FIGURE 5 – Réseau de neurones récurrents (crédits à `http://www.wildml.com`)

Dans notre cas, au lieu d'utiliser comme unité de base un neurone artificiel classique nous avons utilisé des cellules plus complexes appelée Long short-term memory (LSTM). Cette cellule inventée par Jürgen Schmidhuber est dotée d'une mémoire interne. Un mécanisme de multiplications matricielles permet d'utiliser l'histoire de la cellule pour moduler sa sortie.

Ces réseaux sont bien plus complexes à entraîner qu'un perceptron multicouche et nécessitent plus de temps. Nous avons réalisé également des recherches d'hyperparamètres. Voici un tableau récapitulatif :

Caractéristiques	**1**	**2**
Nombre de couche cachée	1	1
Nombre de neurones par couche	64	64
Fonction d'activation	Tanh	Tanh
Taille des batchs	100	100
Bruit blanc sur les poids	0	0.01
Dropout	0	0.5
Algorithme de descente de gradient	RMS-Prop	RMS-Prop
Learning rate	10^{-5}	10^{-5}
Momentum	0.9	0.9
Step clipping	1	1
Décroissance du learning rate	2	2
Erreur sur la validation	0.45	0.38

Nous avons pu remarquer que :

— Les réseaux de neurones récurrents sont difficiles à entraîner.
— La fonction d'activation tangente hyperbolique est bien plus performante que la fonction sigmoïde ou rectifier.
— Un léger bruit blanc sur les poids permet de meilleurs résultats sur le test.
— Un réseau peu profond et avec peu de neurones par couche est suffisant pour capturer la complexité des données.

Malgré tous nos efforts, ces modèles récurrents n'ont pu obtenir d'aussi bons résultats que les modèles à perceptron multicouche, ce qui est dû à la difficulté à entraîner ces réseaux. Une recherche plus poussée d'hyperparamètres permettrait peut être de tirer partie de toutes les capacités de ces réseaux récurrents.

4 Comparaison des modèles

Modèles	**Random Forest**	**MLP**	**RNN**
Nombre d'arbres	200		
Nombre de couche cachée		3	1
Nombre de neurones par couche		50	64
Fonction d'activation		Tanh	Tanh
Taille des batchs		400	100
Bruit blanc sur les poids		0	0.01
Dropout			0.5
Algorithme de descente de gradient		RMS-Prop	RMS-Prop
Learning rate		10^{-5}	10^{-5}
Momentum		0.9	0.9
Step clipping		2	2
Décroissance du learning rate		1	1
Erreur sur la validation	**0.7**	**0.31**	**0.38**

5 Conclusion

Le meilleur coût obtenu est de **30.21%**. Ce score nous a permis de nous classer **9ème** sur 27 participants.

Bien que simple, la méthode par forêt d'arbres décisionnels produit des résultats honorables. Les méthodes d'apprentissage profond donnent de très bons résultats. Nous pensons que ces mêmes architectures seraient capables de faire mieux. Une recherche encore plus poussée d'hyperparamètres devrait permettre d'améliorer le score de ces modèles.