Projet de Master

Romain Mencattini

15 février 2018

Table des matières

1 Introduction

Ajouter une introduction qui donne envie. Attirer le lecteur. À la fin de l'introduction, faire un plan détaillant grossièrement chaque parties.

Le but de ce projet est d'utiliser des techniques de *machine learning* dans le cadre de la finance. Plus précisément, nous allons reprendre des techniques algorithmiques pour créer un programme de *trading* de taux de changes.

Nous allons dans un premier temps faire un état de l'art. Ce dernier sera composé de plusieurs parties.

La première traitera la marché des taux de changes afin d'obtenir les connaissances pour comprendre les enjeux et les buts recherchés. Il s'agira d'une introduction d'éléments de bases mais néanmoins essentiels à la compréhension des algorithmes.

La deuxième partie plus mathématique abordera l'aspect théorique de plusieurs algorithmes clefs. Soit :

- Les réseaux de neurones.
- Les arbres de décision.
- Les algorithmes *SVM*.
- *Logistic Regression*.
- *Naive Bayes*.
- Descente du Gradient.

Ensuite, nous verrons l'application de la théorie à notre cas concret, le marché des change ; leurs problèmes, limitations et solutions rencontrés ainsi que les résultats concernant les performances des programmes.

Pour conclure, nous justifierons le choix de l'algorithme ainsi que les éléments clefs du *machine learning*.

Après l'état de l'art, nous implémenterons l'algorithme ou une analyse plus poussée de sa partie mathématique sera effectuée, mais également de son application au domaine financier. Nous analyserons également les problèmes et solutions rencontrés.

Une fois l'implémentation terminée, plusieurs *benchmark* seront effectués afin d'estimer les améliorations les plus pertinentes et la performance générale de l'algorithme.

Finalement, nous analyserons les résultats et déduirons des conclusions. De plus une analyse de la pertinence de l'algorithme et des optimisations sera réalisée et nous proposerons des pistes d'améliorations.

2 État de l'art

2.1 Introduction

Avant la démocratisation de l'informatique, les opérations financières étaient réalisées par des humains. Ce système pouvait avoir des inconvénients :

- L'émotionnel influençait les transactions. En effet, ces dernières étant effectuées par des humains, il y avait un risque non négligeable que l'état de la personne agisse sur sa décision.
- Un problème sous-jacent était de maintenir une discipline de *trading*. Afin de minimiser les pertes et de maximiser les gains, il fallait se tenir à un plan afin de ne pas se laisser influencer par des paramètres extérieurs. Cela pouvait être très difficile.
- Le *backtesting*[1] était impossible. Tester la qualification ainsi que la qualité de *trading* d'une personne était compliquée. De même pour un *trading plan*.

Ces éléments ont, en partie, favorisé l'émergence et l'utilisation d'algorithmes dans la finance. En 2014 aux États-Unis, 84% des transactions étaient accomplies par des algorithmes [27]. Ce qui représente environ 100'000 réalisations, ou *ticks*, par secondes [27]. L'informatique a engendré une automatisation très importante des transactions. Durant l'évolution de l'outil informatique, le monde de la finance en a suivi les améliorations afin de perfectionner leurs algorithmes. L'automatisation mise à part, on retrouve des méthodes d'optimisations poussées ainsi que les récentes découvertes de *data mining* et de *machine learning*, abrégé *ML*. Des propositions de plus en plus pointues dans les deux domaines voient le jour. L'algorithme qui sera au cœur de ce projet en fait partie. Il s'agit d'un réseau de neurones avec plusieurs couches prenant en compte des paramètres particuliers à la finance.

Afin d'approcher aux mieux ces notions, nous allons discuter des éléments nécessaires à leur compréhension. Nous allons en premier lieu traiter le domaine financier ainsi que ces outils. Puis nous parlerons de plusieurs méthodes de *ML*. Il est compliqué d'être exhaustif tant les méthodes sont nombreuses et disposent de variations complexes. Les algorithmes choisis présenterons un large panel de techniques et constitueront une introduction complète au domaine. Ces derniers comprendront les réseaux de neurones, les arbres de décision, le *Support Vector Machine* [36], les classifieurs *Naive Bayes* ainsi que la descente du gradient [31] et sa version dite stochastique [26].

Finalement, nous lierons les deux domaines en montrant comment adapter les modèles mathématiques de *ML* pour les utiliser comme techniques de *trading*, en évaluant leur performances.

1. Le *Backtesting* est le processus qui consiste à tester une stratégie de *trading* sur des données historiques afin de s'assurer de sa viabilité avant de risquer du capital. [10]

2.2 Finance

2.2.1 *FOREX*

Avant de parler du *FOREX*, il convient d'en donner une définition [14] :

"*Forex (FX) is the market in which currencies are traded. The forex market is the largest, most liquid market in the world [...]. It includes all of the currencies in the world. There is no central marketplace for currency exchange ; trade is conducted over the counter. The forex market is open 24 hours a day, five days a week, except for holidays, and currencies are traded worldwide among the major financial centers of London, New York, Tokyo, Zürich, Frankfurt, Hong Kong, Singapore, Paris and Sydney.[...] any person, firm or country may participate in this market.*"

Ce marché permet de faire des transactions sur toutes les monnaies. Il n'y a pas de place centrale, les échanges pouvant être conduits par les pays. Précision importante, chaque personne peut participer au *FOREX*.

Afin d'appréhender le fonctionnement du *FOREX*, il est important de mentionner certaines décisions historiques. Ces dernières ayant façonné le marché des devises actuel.

Jusqu'à la première guerre mondiale, le système en vigueur se basait sur l'or, que l'on nommait l'étalon-or [1]. S'en suit une période d'instabilité notamment dûe aux pertes occasionnées par la guerre, un après-guerre compliqué, la crise boursière de 1929 et la seconde guerre mondiale.

C'est au sortir de cette dernière, que la nécessité de "*mettre en place une organisation monétaire mondiale et de favoriser la reconstruction et le développement économique des pays touchés par la guerre*" [34], est apparue. Le but était également "*d'aplanir les conflits économiques, reconnaissant par là les problèmes engendrés par les disparités économiques*" [29].

Plusieurs idées furent proposées, mais ce fût celle de Harry Dexter White qui fût mise en place. Cette dernière prévoyait entre autre :

- le choix du Dollar américain comme étalon, avec rattachement à l'or [2].
- Création de la Banque internationale pour la reconstruction et le développement (BIRD) qui deviendra la banque mondiale.
- Le Fond monétaire international (FMI).
- Création de l'Organisation mondiale du commerce [3].

On remarque que ces institutions sont toujours en activités, cela démontre l'importance de ces accords pour le système financier actuel.

Le marché *FOREX* porte sur les devises. La valeur d'une devise ne peut être exprimée qu'en fonction d'une autre. Par exemple 1 franc suisse vaut 1.05 euro. [4] La transaction

1. Source : [29].
2. Suspension de l'équivalence or pour le dollar américain en août 1971 puis abandon définitif en mars 1973 [34].
3. Ne verra le jour qu'en 1995 faute d'accord [34].
4. Taux fictif utilisé pour l'exemple.

porte donc sur deux monnaies comme CHF/EUR. On va vendre des francs suisses pour acheter des euros ou l'inverse. Le nom du marché vient d'ailleurs de ces échanges. On échange une monnaie contre une autre, c'est un *FOreign EXchange*, ou *FOREX*.

Il y a deux variations possibles :

— La monnaie peut subir une dépréciation.

— La monnaie peut subir une appréciation.

Lorsque le prix d'une devise augmente par rapport à une monnaie étrangère, on parle d'appréciation. Ainsi dans le cas contraire, on parlera d'une dépréciation.

La mondialisation a facilité ce marché. En effet, toutes devises étant accessibles depuis n'importe où, il devient donc possible d'avoir des marchés avec des devises plus exotiques. Il devient par exemple possible de vendre une monnaie avec un faible taux d'intérêt puis utiliser les fonds générés par la vente pour investir dans une monnaie avec un meilleur taux d'intérêt. Le but étant de capturer le différentiel d'intérêt [12].

Les principaux acteurs financiers sont [8] :

— Les banques commerciales. Elles peuvent pratiquer des interventions directes car elles gèrent des dépôts et veulent opérer des transactions sur ces derniers. Il leur est également possible de réaliser le rôle d'intermédiaire financier.

— Les entreprises. Ces dernières vont pratiquer des transactions directes, si elles disposent d'un accès aux marchés sinon via des intermédiaires.

— Les institutions financières non-bancaires. On peut citer les fonds de pensions, les sociétés d'assurances ou les *hedge funds*. Ce sont surtout dans un but de spéculation, d'arbitrage ou de couverture de risque qu'elles agissent.

— Les banques centrales. Il peut y avoir des interventions directes, dans le but de modifier l'appréciation de la monnaie.

— Les ménages. Surtout dans une optique de voyage, d'achat ou de spéculation.

Henry Bourguinat a énoncé "*la règle des trois unités*" qui correspondent aux unités de temps, de lieu et d'opérations et d'acteurs. Le *FOREX* répond à ces trois unités [19] :

— Ce marché fonctionne 24h/24 et les transactions s'effectuent presque en continue.

— Il fonctionne à l'échelle mondiale tout en étant décentralisé. De part l'évolution des technologies, l'information circule aisément malgré son statut.

— L'uniformité des procédés ainsi que des produits est présente. Les acteurs malgré nationalité sont de même nature.

Il existe principalement deux horizons temporels : le *spot* et le *forward*.

Le premier est également appelé "Le marché au comptant". Lorsque deux acteurs se mettent d'accord sur une transaction, cette dernière se réalise immédiatement [1].

Le second peut être nommé "Le marché à terme". L'accord est passé à un temps T mais la transaction effective ne se réalise que dans le futur. Il également possible que la

1. Valable en théorie, dans la réalité cela peut prendre du temps [8]

transaction ne se réalise pas. Ce futur, ou maturité, peut être de plusieurs dizaines de jours, voir des années, soit $T + X$ [1].

Il y a des opérations réalisable sur le marché à terme :

— Les *swaps*. Ils consistent à vendre une monnaie au comptant puis à la racheter à terme [2].

— Les *futures/forwards*. La différence entre ces deux tient surtout à leur standardisation et leur mise en place. Cependant le principe reste le même : on réalise une opération (d'achat ou de vente) qui ne s'effectuera qu'à maturité.

— Les *options*. Cela représente un contrat vendu par un parti (*the option writer*) à un autre parti (*the option holder*). Ce contrat offre le droit, et non l'obligation contrairement aux *futures/forwards*, d'acheter (*call*) ou de vendre (*put*). Ici encore, il faut attendre la maturité [3].

Les options sont très versatiles. Elles peuvent être utilisées afin de spéculer ou de diminuer le risque. Voici les différents types possibles :

— **long call** → on achète le droit d'acheter le sous-jacent à un certain prix.

— **short call** → on vend le droit d'acheter le sous-jacent à un certain prix.

— **long put** → on achète le droit de vendre le sous-jacent à un certain prix.

— **short put** → on vend le droit de vendre le sous-jacent à un certain prix.

Le *bid* est le prix maximum qu'un acheteur est d'accord de payer pour un sous-jacent. De la même manière, le *ask* est le prix minimum qu'un vendeur accepte pour vendre un sous-jacent [11].

La différence entre le *bid* et le *ask*, appelée le *spread*, représente la liquidité d'un actif. Il est également utilisé comme marge par les *broker* [33] et autres plateformes.

2.3 Cadre théorique des algorithmes de *Machine Learning*

2.3.1 Introduction

T. Mitchell a donné une définition formelle [21] :

"*A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E*"

Cette citation signifie que si nous disposons d'une tâche T à accomplir. T peut consister à trier des images ou à reconnaître des motifs. La mesure de la réussite de T est nommée P. C'est-à-dire la qualité du résultat pour la tâche donnée, T. Si le programme améliore P pour la tâche T grâce à l'expérience E. Il s'agit d'un programme de *machine learning*. L'expérience E peut être vue comme une phase d'entraînement ou comme le fait de retenir les réponses après avoir accompli la tâche.

1. Où T est le moment présent, et X une durée de temps.
2. Soit à $T + X$
3. Cela est vrai pour les options dites européennes [13]. Dans le cas des options américaines [9], le droit peut s'exercer à n'importe quel moment, offrant ainsi une plus grande flexibilité.

Il existe deux catégories d'apprentissage :

— L'apprentissage supervisé.

— L'apprentissage non-supervisé.

Dans le cas du premier, on fournit au programme, un ensemble d'entraînement [1], qui contient des réalisations ainsi que le résultat de la classification. Le programme va donc pouvoir utiliser ce savoir afin d'améliorer sa performance P. Nous disposons donc de nombreux couples (x_i, y_i) et le but est de trouver une fonction $f \in F$ telle que : $f(x) = y$.

Pour l'apprentissage non-supervisé, on fournit des données, mais sans le résultat voulu. C'est uniquement après avoir décidé d'une valeur qu'on va signifier au programme si cette dernière est correcte. On ne lui donnera jamais la valeur attendue. Il va donc utiliser uniquement les résultats précédents pour améliorer son P.

Par exemple, on désire reconnaître un certain type de voiture à partir d'images. Dans le cas de l'apprentissage supervisé, nous allons fournir au programme un ensemble d'entraînement qui contient de nombreuses photos de voitures, ainsi que la marque des dites voitures. L'algorithme va donc travailler avec ces données.

Par contre dans le cas de l'apprentissage non-supervisé, le programme ne pourra utiliser que les photos, et après avoir retourné le résultat, nous lui dirons si c'est juste ou faux. Il mémorisera le résultat optimisera en conséquence ses réponses.

Concernant, l'ensemble d'entraînement, il y a des points à prendre en compte afin de minimiser les risques de sur-apprentissage [2], et de maximiser la qualité de nos données. Pour ce faire il faut :

— Représenter la population générale. Donc si le but est du traitement de la langue, il faut que la propension et la répartition des mots soient les mêmes que ceux de la langue.

— Contenir des membres de chacune des classes. Pour reconnaître des chiffres, il est important de disposer de chacun des chiffres dans l'ensemble d'entraînement.

— Contenir de grandes variations ainsi que du bruit. Afin d'éviter le sur-apprentissage, il faut de nombreux exemples différents, voir très différents, les uns des autres ainsi que du bruit [3].

Il est important de saisir comment fonctionne les algorithmes de *machine learning*. Le but est d'utiliser des données, souvent de très hautes dimensionnalités [4], dans des équations dont on pourra faire varier les paramètres afin de classifier au mieux. Le cœur des algorithmes de *machine learning* consiste à optimiser les dits paramètres.

1. Ou d'expérience, E
2. Le sur-apprentissage consiste à apprendre par cœur la tâche, plutôt que d'apprendre les principes pour réaliser la tâche.
3. Comme des faux exemples.
4. Cela signifie qu'elles sont représentées par un grand nombre d'attributs.

2.3.2 *Logistic Regression*

Une régression en statistique consiste à analyser la relation entre une variable par rapport à un ensemble d'autres [35]. De manière plus simple, l'objectif est d'interpoler au mieux les données à partir des éléments connus ; il s'agît d'une analyse prédictive. Afin d'atteindre cet objectif, plusieurs techniques sont possibles comme l'utilisation de l'analyse numérique avec des outils comme les splines [1] ou une approche plus statistique avec des variables aléatoires et des probabilités conditionnelles. C'est de cette dernière dont il est question.

On veut estimer la probabilité conditionnelle, en se basant sur des variables et en utilisant une distribution logistique cumulative [1] :

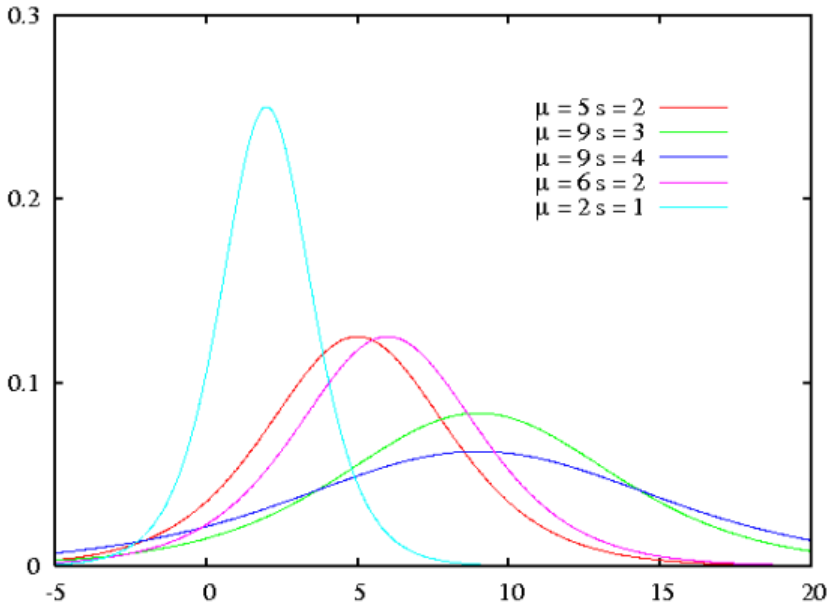

FIGURE 1 – Exemple de distribution logistique.

D'un point de vue mathématique, la densité de probabilité est donnée par :

$$f(x;\mu,s) = \frac{e^{-\frac{x-\mu}{s}}}{s\left(1+e^{-\frac{x-\mu}{s}}\right)^2} = \frac{1}{s\left(e^{\frac{x-\mu}{2s}}+e^{-\frac{x-\mu}{2s}}\right)^2}$$

Les probabilités cumulées par :

$$F(x;\mu,s) = \frac{1}{1+e^{-\frac{x-\mu}{s}}} = \frac{1}{2} + \frac{1}{2}\ \tanh\left(\frac{x-\mu}{2s}\right)$$

Cette dernière a une forme semblable à une distribution Gaussienne, mais avec des

1. Source : https://upload.wikimedia.org/wikipedia/commons/e/e9/Logisticpdfunction.png

queues épaisses et donc une *kurtosis* plus élevée. La *kurtosis* étant définie comme suit :

$$Kurt(X) = E\left[\left(\frac{X-\mu}{\sigma}\right)\right]$$

Où μ est la moyenne mathématique et σ la déviation standard.

Le but est de modéliser : $P(Y = 1|X = x)$ comme fonction de x. Nous voulons donc savoir quelle est la probabilité que la classe[1] Y vaille 1 sachant que X vaut x.

Le modèle de régression est le suivant [18] :

$$log(\tfrac{p(x)}{1-p(x)}) = \beta_0 + x \cdot \beta$$

, où β_0 représente l'ordonnée à l'origine de la régression linéaire , β est le coefficient de régression, de même dimension que x, et x la donnée dont on veut obtenir la classe.

En résolvant cette équation pour p, cela donne [18] :

$$p(x|y) = \tfrac{1}{1+e^{-(\beta_0+x\cdot\beta)}}$$

Dans le cadre de l'article : *A Machine Learning Approach to Automated Trading* [18], l'auteur a implémenté deux variations de cet algorithme :

— *Logistic regression with a ridge penalty* :

$$\sum_{i=1}^{N}(y_i - \sum_j \beta_j x_{ij})^2 + \lambda \sum_j \beta_j^2$$

, où y_i est la classe de notre observation, β_j sont les coefficients de la régression logistique originale [18] et x_{ij} sont les j éléments de l'observation i.
L'objectif est de minimiser le carré de la différence entre la classe, ou résultat, de y_i et le résultat calculé : $\sum_j \beta_j x_{ij}$. Donc en fonction de l'observation x_i et des coefficient de β. En ajoutant une pénalité, sur β, on va tenter d'éviter le sur-apprentissage.

— *Lasso logistic regression/ Lasso regularization* :

$$\sum_{i=1}^{N}(y_i - \sum_j \beta_j x_{ij})^2 + \lambda \sum_j |\beta|$$

Le fonctionnement est similaire au précédent, seul change la pénalité.

Comme mentionné plus haut, l'optimisation porte sur les paramètres de l'équation. Dans le cas de la régression logistique, il s'agît du β. Il conviendra donc de trouver la valeur optimale pour cette variable afin de maximiser ou minimiser les équations ci-dessus. De manière similaire pour les autres techniques de *ML*, les équations et les paramètres vont changer, mais le but sera toujours d'optimiser ces éléments.

1. Dans ce cas, la classe désigne une réalisation de la variable aléatoire Y.

2.3.3 Les arbres de décision

Un arbre de décision est un arbre, dont chaque nœud représente un test sur un attribut. Les branches qui suivent directement le nœud sont les valeurs possibles de l'attribut. Les feuilles de l'arbre, quant à elles, sont la classification d'élément donné en entrée.

Il est important de disposer des attributs avant de commencer la construction de l'arbre. Lors de l'implémentation, nous pouvons représenter l'arbre comme une suite de *if-then-else* afin d'améliorer la lisibilité. Dans ce cas très précis, disposer d'un langage permettant le *pattern matching* est fort utile.

Cet algorithme a tendance à très facilement sur-apprendre, il convient donc de bien choisir la manière de construire l'arbre ainsi que l'ensemble d'entraînement pour minimiser cet effet.

Voici un exemple d'arbre de décision [1] :

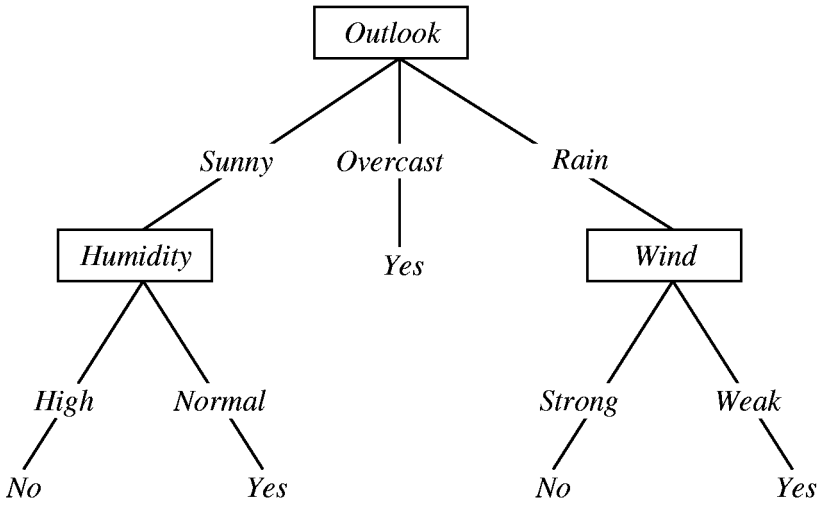

FIGURE 2 – Exemple d'arbre de décision : Permet de décider si nous pouvons aller jouer au tennis ou non.

Afin de construire l'arbre à partir de l'ensemble d'entraînement, il existe plusieurs algorithmes. Un des plus connus est le *ID3* [20]. Il s'agit d'une méthode de type *greedy*.

À chaque itération, il faut :

— effectuer un test statistique [2] afin de trouver l'attribut le plus discriminant.
— utiliser cet attribut comme nœud.
— retourner au premier point, tant que l'ensemble des attributs n'est pas vide.

Nous allons construire l'arbre de manière *top-down* en utilisant ,à chaque pas, le meilleur attribut, selon notre test statistique. La construction de l'arbre constitue la phase d'en-

1. http://cloudmark.github.io/images/kotlin/ID3.png
2. Il vise à vérifier la quantité d'information gagnée pour la classification [20].

traînement. Si de nouvelles données viennent s'ajouter, il devient nécessaire de construire un nouvel arbre, soit une nouvelle phase d'entraînement.

La phase de classification consiste à parcourir l'arbre en appliquant les prédicats à l'instance que nous désirons classifier. Une fois arrivé a une feuille, nous avons un résultat.

2.3.4 *Naive Bayes*

À l'instar de la régression logistique (voir 2.3.2), il s'agit d'un classifieur probabiliste. Ce dernier se base sur le théorème de Bayes [1] :

$$P(Y|X) = \frac{P(X|Y)}{P(X)}P(Y)$$

À partir de cela, nous obtenons l'équation de *machine learning* suivante [18] :

$$V_{NB} = arg\ max_{v_j \in V} P(v_j) \prod_i P(a_i|v_j)$$

Où V_{NB} est la classe obtenue, $P(v_j)$ la probabilité à *priori* donc sans informations et $\prod_i P(a_i|v_j)$ la probabilité de vraisemblance. Il convient donc de trouver la classe qui maximise ce calcul.

Afin d'avoir une certaine sécurité dans les résultats, il est possible d'ajouter un seuil. Les réponses du classifieur étant comprise entre 0 et 1, le seuil permettra de décider si la réponse sera prise en compte.

Par exemple, avec un seuil de 0.6 si $V_{NB} = 0.58$, alors la réponse n'est pas validé et le classifieur ne renvoie aucun résultat. Si $V_{NB} = 0.7$ alors la réponse est jugée sûre et la classe de V_{NB} est retournée.

La *ROC*[2] *Curve analysis* permet d'améliorer le classifieur. En effet cette dernière peut détecter les *true positive rate* par rapport aux *false positive rate* pour différents seuils de classification de l'algorithme *Naive Bayes* [18]. À partir de cela, nous pouvons déterminer le meilleur seuil de sortie et donc perfectionner notre algorithme. De plus cette courbe peut aider à comparer des classifieurs entre eux en comparant la surface sous la courbe [18].

La méthode utilisée est la suivante [18] : il faut faire s'intersecter la pente S avec la courbe ROC et ainsi obtenir une valeur optimale pour le seuil. Cette pente S est définie comme suit [18] :

$$S = \frac{Cost(P|N) - Cost(N|N)}{Cost(N|P) - Cost(P|P)} \cdot \frac{N}{P}$$

Sachant que $Cost(P|N)$ est le coût pour avoir mal classé une classe négative comme positive. P est la somme des vrais positifs et des faux négatifs. N, quant à lui, vaut la somme des vrais négatifs ainsi que des faux positifs.

Voici une illustration [3] :

1. https://brilliant.org/wiki/bayes-theorem/
2. Receiver Operating Characteristic
3. Source : http://www.prolekare.cz/dbpic/jp_5403_f_20-x1000_1600

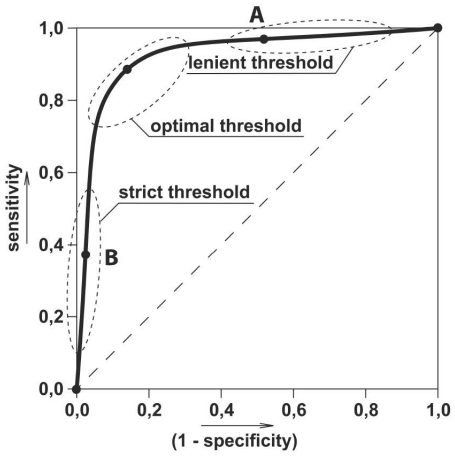

FIGURE 3 – Exemple de *ROC curve* : L'axe des x correspond au taux de faux positifs et l'axe des y au taux de vrais positifs. On veut donc tendre vers $y = 1$ et $x = 0$. Lors de l'optimisation par la pente S, le but sera d'obtenir une intersection avec la *ROC curve* dans la zone *optimal threshold* afin d'avoir le meilleur seuil possible.

2.3.5 *SVM*

Le but de l'algorithme *SVM*[1] est de séparer les données grâce à un hyper-plan. Cela permet de différencier les classes des observations suivantes en déterminant s'ils se trouvent d'un côté ou l'autre du plan. Ce dernier n'est pas unique[2] :

1. *Support Vector Machine*
2. Source : https://computersciencesource.files.wordpress.com/2010/01/svmafter.png

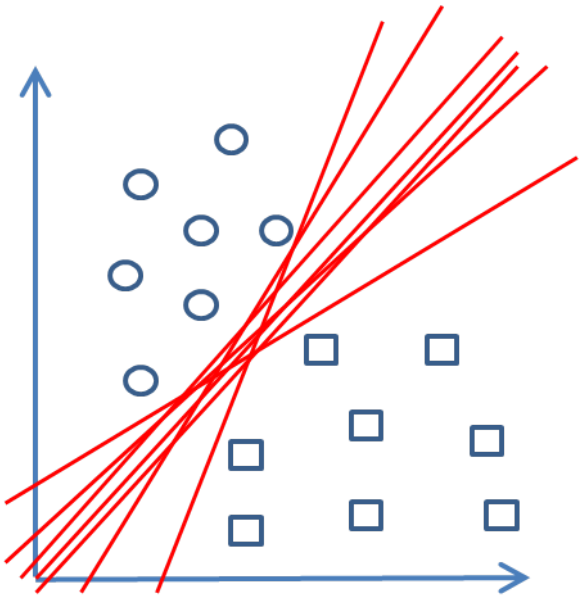

FIGURE 4 – Exemple d'hyper-plans séparant les données. Si nous voulons classifier un nouvel élément, il suffit de calculer s'il se trouve à gauche ou à droite de l'hyper-plan. Dans le premier cas, il s'agira, pour notre algorithme, d'un rond et dans l'autre d'un carré

Notre fonction est :

$$f(x) = (w \cdot x) + b$$

Où x est le vecteur [1], w est le vecteur normal à l'hyperplan et b l'ordonnée à l'origine. Le but est de maximiser la distance entre les points les plus proches de l'hyper-plan, tout en pénalisant les points mal classés. Il n'est pas toujours possible de séparer les données de dimensions n, il conviendra donc d'augmenter la dimension afin d'obtenir une dimension $m > n$ plus discriminante. La fonction $\phi(x)$ est utilisée dans ce but. Un exemple de fonction est :

$$\phi : R^2 \to R^3 : (x, y) \to (x, y, z) := (x^2, \sqrt{2}xy, y^2)$$

Il est possible d'imager cette opération comme cela [2] :

1. i.e notre instance.
2. https://www.dtreg.com/uploaded/pageimg/SvmDimensionMap.jpg

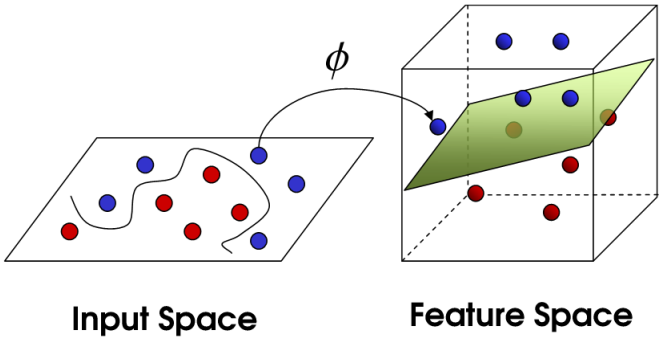

FIGURE 5 – Exemple d'utilisation de la fonction ϕ pour passer d'un espace R^2 à R^3 afin de faciliter la séparation.

En terme d'équation, nous voulons minimiser w, dans l'équation de l'hyper-plan : $(w \cdot x) + b$. Il faut également prendre en compte $y_i \in \{-1, +1\}$:

$$(w \cdot x) + b = \begin{cases} \geq +1 & \text{si } y_i = +1 \\ \leq -1 & \text{si } y_i = -1 \end{cases}$$

Ce qui donne [28] :

$$y_i(w \cdot x_i + b) \geq 1$$

Si malgré l'augmentation de la dimension, les données ne sont pas séparables, il faut tenter de minimiser le nombre d'éléments mal placés. Pour ce faire [28] :

$$y_i(w \cdot x_i + b) \geq 1 - \xi_i \text{ avec } \xi_i > 0$$

Afin de diminuer l'erreur et optimiser au mieux notre classifieur.

2.3.6 Réseaux de neurones

Tout comme les algorithmes génétiques s'inspirent de la sélection naturelle [32] dans un but d'optimisation, les réseaux de neurones se basent sur un modèle formel de neurones [1] afin de copier la capacité d'apprentissage des êtres vivants.

Il s'agit d'opérer à partir de données en entrée, des *inputs*, une ou plusieurs multiplications matricielles en utilisant des vecteurs de poids, des *weights*. L'optimisation s'applique sur les *weights*, afin de maximiser la classification.

Un réseau de neurones peut avoir plusieurs couches, *layers*. Dans ce cas, la première couche est appelée *input layer*, la dernière *output layer* et toutes celles entre ces deux sont les *hidden layers*. De plus, les neurones peuvent être pleinement connectés avec ceux de la

1. Neurones formels : http://www.peoi.org/Courses/Coursesfr/neural/neural3.html

couche suivante, *feed-forward* ; ce qui signifie que les neurones de la couche $n-1$ influencent ceux de la couche n. Il est également possible que les éléments de n agissent sur les neurones de $n-1$, ce phénomène est appelé *feedback networks*.

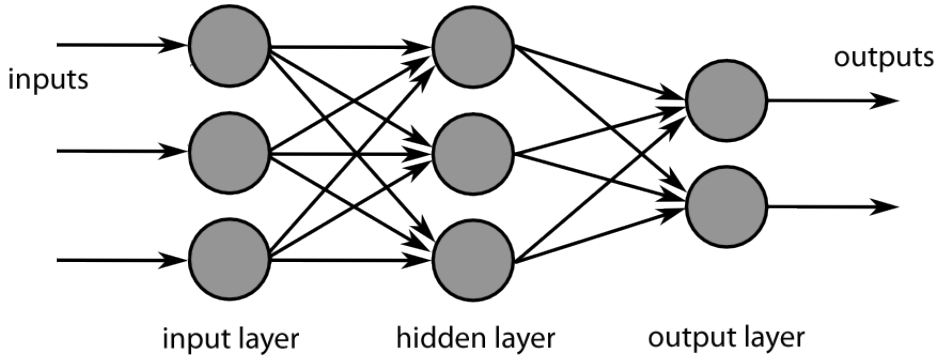

FIGURE 6 – Exemple d'un réseau de neurones avec plusieurs couches. Illustre également le *feed-forward* : l'*ouput* de l'*input layer* est propagé dans chaque neurones de l'*hidden layer*. Même chose pour les deux dernières couches. [1]

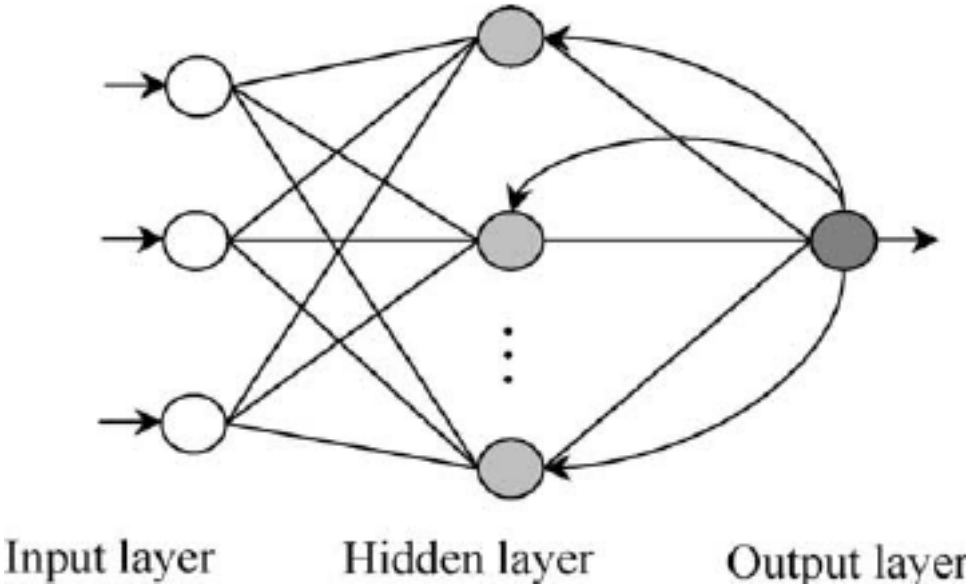

FIGURE 7 – Exemple d'un réseau de neurones avec plusieurs couches. Dans ce cas là, les couches font du *feed-forward* mais de plus, le *feedback* est utilisé afin d'influencer les couches précédant l'*output layer*. [2]

1. Source : http://web.utk.edu/~wfeng1/spark/_images/fnn.png
2. Source :https://jcrisch.files.wordpress.com/2015/04/reseau_de_neurones.png

Concernant la valeur de sortie, elle peut être simple, comme le résultat d'une classification binaire, *i.e.* 0 ou 1 en sortie. Mais elle peut également être d'une dimensionnalité plus élevée comme un vecteur, citons l'exemple d'un point dans un espace R^2.

Afin de borner les valeurs en sortie, la plupart des réseaux utilisent une fonction. Nous pouvons citer :

— La fonction sigmoïde : $S(t) = \frac{1}{1+e^{-t}}$.

— La fonction tangente hyperbolique : $f(x) = tanh(x)$.

Bien souvent, l'utilisation d'un réseau de neurones à une couche est suffisante. Cela est valable pour les fonctions continues, dans le cas de fonctions discontinues, il est intéressant de passer à un réseau disposant de plusieurs couches. Attention toutefois, si le nombre de neurones est trop important, l'algorithme va avoir tendance à sur-apprendre, et à l'inverse, à sous-apprendre si le nombre est trop faible. Il est donc important de bien doser cette quantité afin d'éviter ces problèmes.

Un exemple concret de réseau de neurones est celui présenté dans l'article qui se trouve au cœur du projet [4]. Avant de montrer les équations, en voici un schéma [4] :

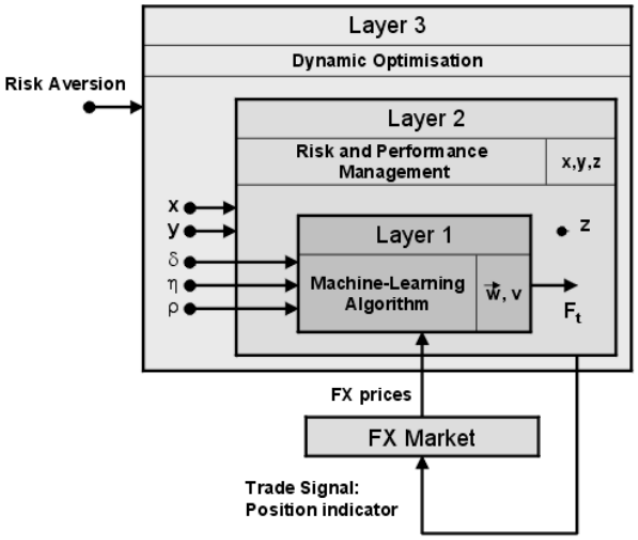

FIGURE 8 – Représentation sous forme de schéma du réseau de neurones. Avec w le vecteur de poids, v le seuil, δ le coût de *trading*, η un paramètre d'adaptation, ρ le taux d'apprentissage, x le seuil d'arrêt lors de pertes, y le seuil de *trading*, z la condition d'arrêt automatique lors de pertes critiques et l'aversion au risque notée v.

Il s'agit de trois couches qui doivent optimiser chacune une série de paramètres. La première va maximiser le vecteur de poids w ainsi que v le seuil. À partir de ces paramètres,

nous pouvons utiliser la formule suivante :

$$F_t = sign(\sum_{i=0}^{M} w_{i,t} r_{t-i} + w_{M+1,t} F_{t-1} + v_t)$$

où $r := p_t - p_{t-1}$ est le rendement d'une position. F_t nous permet de calculer la position à prendre au temps t en tenant compte de l'historique des prix.

L'optimisation de w se fait par un algorithme de descente du gradient (voir 2.3.7).

La deuxième couche va travailler à partir des paramètres x, y, z. Il est possible que le marché soit irrationnel durant une longue durée, dans ce cas la psychologie peut pousser à garder une position en espérant un changement. C'est ce genre de comportement qu'un algorithme permet d'éviter. Pour ce faire, nous définissons un excédant des pertes et nous veillons qu'il soit toujours à une distance x du meilleur prix atteint ; afin d'éviter de tenir trop longtemps une position défavorable.

Il est également intéressant de définir un seuil. Si la réponse du réseau est supérieure à ce seuil, nous la prenons en compte et dans le cas contraire, nous ne faisons rien. Cela permet d'évaluer la "force" du signal renvoyé par la première couche. Ce seuil est y.

La dernière variable utilisée dans cette couche est z. Il y a un consensus dans la communauté de *trading* concernant le fait que les algorithmes fonctionnent bien durant un temps puis cesse d'être profitable. À ce moment, il convient d'arrêter le programme et d'y apporter des modifications [1] avant de le relancer. La tâche du paramètre z est de donner un seuil pour les pertes du profit cumulé qui, lorsqu'il est dépassé, lance une procédure d'arrêt. Contrairement à x, y qui seront optimisés par la troisième couche, le paramètre z est fixé au début du programme et ne change plus.

La couche d'optimisation dynamique est la troisième couche. Cette dernière va, à chaque itération, optimiser les paramètres suivants : x, y, η, δ, ρ.

Afin de conduire à bien l'optimisation, il fallu définir une fonction de coût [4] :

$$\Sigma := \frac{\sum_{i=0}^{N} (R_i)^2 I(R_i < 0)}{\sum_{i=0}^{N} (R_i)^2 I(R_i > 0)}$$

$$U(\overline{R}, \Sigma, \upsilon) := a \cdot (1 - \upsilon) \cdot \overline{R} - \upsilon \cdot \Sigma$$

,où $R_i := W_i - Wi - 1$ est le rendement au temps i avec W_i le profit cumulé, $\overline{R} := \frac{W_N}{N}$ est le profit moyen avec N qui est le nombre d'intervalle, a une constante et υ l'aversion au risque. Ces équations semblent complexes, néanmoins Σ représente les *returns* négatifs sur les positifs.

Ces équations ont été construites de manière à posséder ces propriétés [4] :

— Une stratégie négative implique un risque très élevé afin d'éviter de soudaines et importantes pertes.

1. Au niveau des équations, des données ou des paramètres.

— La mesure Σ pénalise uniquement les stratégies négatives et pas les stratégies positives.

La fonction à optimiser est donc :

$$\max_{\delta,\eta,\rho,x,y} U(\overline{R}; \Sigma : \delta, \eta, \rho, x, y; \upsilon)$$

Car x, y, η, δ, ρ interviennent tous dans le calcul de Σ. Ces derniers peuvent être optimisés de la manière voulue.
Les réseaux de neurones peuvent être optimisés de plusieurs manières. L'une d'entre elle est la descente du gradient.

2.3.7 Descente du gradient

L'algorithme de descente du gradient fonctionne sur des fonctions réelles différentiables sur un espace tel que $\mathbb{R}^n$. Il est itératif et fonctionne en améliorant l'itération précédente, jusqu'à atteindre une condition d'arrêt.

Avant d'en expliquer la teneur mathématique, voici un exemple de l'algorithme dans $\mathbb{R}^2$:

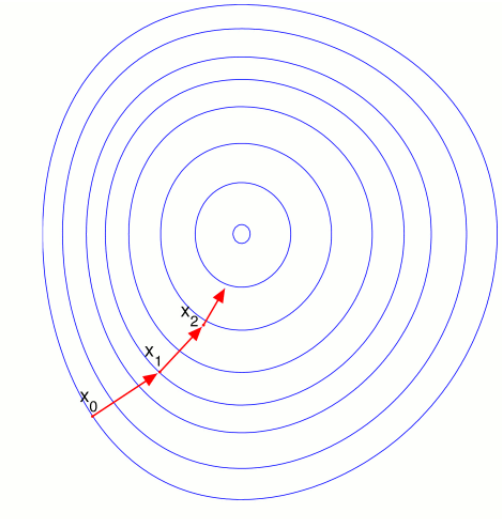

FIGURE 9 – Exemple de l'algorithme de descente du gradient en deux dimensions. À chaque itération, il faut prendre la direction opposée au gradient [1], cela permet d'arriver à un nouveau point. En itérant, on se rapproche de l'optimum.

Afin de comprendre, les divers algorithmes, il est important d'avoir les connaissances mathématiques sur ce sujet. L'algorithme se définit comme suit [2] :

1. Dans cet exemple, il est possible de dire, qu'il faut prendre la normal de la courbe de niveau.
2. Inspiré de [31].

Soit un point initial $x_0 \in \mathbb{R}$. Soit $\epsilon > 0$ un seuil de tolérance. L'algorithme définit une suite d'itération $x_1, x_2, .. \in \mathbb{R}^n$, jusqu'à ce qu'un test d'arrêt soit satisfait. Pour passer de x_i à x_{i+1}, il faut :

- Calculer $\bigtriangledown f(x_k)$
- Si $|| \bigtriangledown f(x_k)|| \leq \epsilon$ alors arrêt.
- Sinon il faut calculer α_k par recherche linéaire [1] sur f en x_k. Cette recherche se fait dans la direction opposée au gradient, soit $- \bigtriangledown f(x_k)$. Une fois α_k calculé, il faut mettre à jour le point itéré :

$$x_{k+1} = x_k - \alpha_k \bigtriangledown f(x_k)$$

La preuve que la recherche dans la direction opposée au gradient induit une décroissance est la suivante. Si la dérivée est non nulle [2] au point x, $f'(x) \neq 0$. Soit

$$d = - \bigtriangledown f(x)$$

Puisque :

$$f'(x) \cdot d = \bigtriangledown f(x) \cdot - \bigtriangledown f(x) = -|| \bigtriangledown f(x)||^2 < 0$$

L'égalité est strictement plus petite car la dérivée est non nulle par hypothèse. Cela implique que :

$$f(x - \alpha \bigtriangledown f(x)) < f(x), \forall \alpha > 0$$

Nous avons donc que pour chaque itération, la valeur obtenue va décroître jusqu'à attendre un maximum, local ou global, ou le seuil ϵ.

La descente du gradient d'un point de vue mathématique peut également être utilisée conjointement à un réseau de neurones. En effet dans cet article [4], les poids w de la première couche, sont optimisés suivant cette formule :

$$w_{i,t} = w_{i-1,t} + \rho \bigtriangleup w_{i,t}$$

Où t correspond au temps, $i - 1$ à l'itération courante et ρ le taux d'apprentissage.

De part la convergence de w_i vers son optimum, cela permet, une fois cette valeur obtenue, de l'injecter comme étant les poids d'un réseau de neurones.

Il existe deux types d'algorithme de descente du gradient :

- Le *Batch Algorithm* a pour but de minimiser la fonction de coût qui aura été définie au préalable en utilisant toutes les données. Ce dernier peut prendre énormément de temps, de par son côté itératif, si l'ensemble d'entraînement croît.

1. La recherche linéaire consiste à choisir une direction de descente afin de minimiser une fonction donnée jusqu'à atteindre l'optimum ou un seuil fixé [30].
2. i.e : Si nous ne sommes pas déjà sur un maximum

— Le *Online Algorithm* va également minimiser une fonction de coût. Contrairement au *Batch Algorithm*, il prendra les exemples un à un. L'optimisation ne se fera que pour l'exemple courant puis recommencera sur une autre donnée. Cette manière de procéder s'applique aisément sur les gros volumes de données.

Provenant de cet article [26], voici des équations pouvant être minimisées dans le cadre d'un programme de *machine learning*. Soit (x_i, y_i), $i = 1..N$, un ensemble de données. Notre fonction de coût est $l(y, y')$. Cela représente le coût de prédire y' quand la réponse est y. Moyennons cela sur tous les exemples :

$$E_N(f) = \frac{1}{N}\sum_{i=1}^{n} l(f_w(x_i), y_i)$$

Où f_w est une fonction pondérée par un vecteur de poids w que l'on cherche à optimiser. Pour optimiser les poids, nous allons utiliser la descente du gradient :

$$w_{k+1} = w_k - \alpha_k \bigtriangledown f_{w_k}(x) \rightarrow w_{k+1} = w_k - \alpha_k \sum_{i=1}^{n} \bigtriangledown Q((x_i, y_i), w_k)$$

Où $Q(x, y) = l(f_w(x), y)$.

Comme mentionné plus haut, la technique de *Batch Algorithm* devient lente lorsque le nombre de données augmente. Pour pallier ce problème, il existe une technique : *stochastic gradient descent*. C'est une méthode d'approximation statistique de la descente du gradient.

Il convient de ne prendre qu'un seul couple au lieu de l'ensemble complet d'entraînement. La somme disparaît donc dans l'équation à minimiser :

$$w_{k+1} = w_k - \alpha_k \bigtriangledown Q((x_i, y_i), w_k)$$

C'est donc une technique de *Online Algorithm*. L'algorithme peut traiter des données à la volée car il n'a pas besoin de se souvenir des exemples précédant.

Il est possible pour pénaliser la complexité de w de rajouter un élément. Cela permet de borner quelque peu les valeurs des poids :

$$w_{k+1} = w_k - \alpha_k \bigtriangledown Q((x_i, y_i), w_k) + \sigma P(w_k)$$

Où $\sigma > 0$ est un hyper-paramètre et P peut valoir :

— **L1 norm** = $P(w) := \sum_{i=1}^{n} |w_i|$

— **L2 norm** = $P(w) := \frac{1}{2}\sum_{i=1}^{n} w_i^2$

— **Elastic Net** = $P(w) := \rho\frac{1}{2}\sum_{i=1}^{n} w_i^2 + (1-\rho)\sum_{i=1}^{n} |w_i|$

2.3.8 *Majority vote*

Le *majority vote* n'est pas strictement un algorithme de *ML*. Il s'agît plutôt d'améliorer les résultats en modifiant la manière d'utiliser certains des algorithmes pré-mentionnés. Il est donc possible d'employer le *majority vote* avec la régression logistique, les réseaux de neurones, etc.

Le principe est le suivant. Soit E notre ensemble d'entraînement, soit $M = \{f_1, f_2, ...\}$ notre ensemble de méthodes f_i de *machine learning*. Notons $M_E = \{f_{1_E}, f_{2_E}, ...\}$ notre ensemble de méthodes f_{i_E} entraînées.

$$f_{i_E}(x) = \begin{cases} 1 & \text{si la classe calculée de } x \text{ est 1.} \\ 0 & \text{si la classe calculée de } x \text{ est 0.} \\ -1 & \text{si aucune classe n'est trouvée ou si le seuil est insuffisant.} \end{cases}$$

L'algorithme se comporte comme suit pour une observation x :

$\forall f_{i_E} \in M_E$, il faut calculer $f_{i_E}(x)$ et garder la classe résultante dans notre liste de résultat R. Si $\exists\ classe_i \in R$, telle que

$$\sum(classe_i \in R) > \frac{|M_E|}{2}$$

Alors cela signifie que la $classe_i$ est notre résultat, dans le cas contraire, l'algorithme ne donne aucune réponse.

L'algorithme va donc, à partir d'un ensemble de méthodes et d'une observation, calculer les classes. Si une de ces dernières est représentée de manière majoritaire [1], le programme retournera ce résultat. Si la majorité n'est pas atteinte, aucune classe n'est jugée valable et donc aucune réponse ne sera rendue.

Cette technique réduit la composante individuelle des méthodes de *ML* ainsi que le risque d'erreur. Si une observation est mal classifiée, cela signifie que plus de la moitié des algorithmes ont fait une erreur. Dans ce cas, il convient de changer les attributs utilisés ou l'ensemble d'entraînement car l'erreur ne proviendra vraisemblablement pas d'un problème d'implémentation ou de la faiblesse de classification d'un algorithme particulier.

2.3.9 *Random Subset*

À l'instar du *majority vote*, le *random subset* est une technique pour employer des méthodes de *ML*.

1. *i.e.* 50% des voix + 1 voix

Le principe est le suivant :

Il faut prendre N différents ensembles de données [1] du domaine concerné. Ils permettront d'entraîner un algorithme sur chacun d'entre eux [2]. Nous disposerons donc de N instances de l'algorithme de *ML* initial, mais avec un entraînement différent pour chacun, puis pour chaque observation, nous allons utiliser ces N instances afin d'obtenir un résultat de classification.

Comme pour le *majority vote*, si une même classe est représentée une majorité de fois, le *random subset* retournera ce résultat, et aucun le cas contraire.

Il est nécessaire d'avoir un grand jeu de données afin de fournir suffisamment d'échantillons aux instances pour qu'elles apprennent correctement. Une fois ce problème résolu, cette technique nous donne la possibilité d'utiliser au mieux l'ensemble d'entraînement. En le fractionnant, cela permet d'entraîner les algorithmes avec des données différentes et donc d'augmenter l'horizon de connaissance du programme.

De plus, le système de majorité promeut une qualité et une confiance accrue dans les résultats obtenus.

2.4 *Machine Learning* dans le cadre de la finance

2.4.1 Introduction

Avant de parler des performances des algorithmes mentionnés, il convient de préciser que ces derniers proviennent de plusieurs articles différents. Ils ont donc été testés avec des paramètres ayant des valeurs disparates ainsi que sur des données distinctes. Il est donc très compliqué de comparer les résultats des algorithmes entre deux articles différents et d'affirmer qu'une méthode est meilleure qu'une autre.

Pour un même article, il sera possible de comparer les performances cependant dans le cas contraire, ces résultats serviront plutôt à illustrer la qualité intrinsèque des procédés. Avec un résultat de 80%, nous pourrons estimer que la performance est bonne, et l'inverse pour une valeur de 20%.

Pour tous les algorithmes qui ont été mentionnés dans la section précédente, nous allons expliquer les changements appliqués à ces derniers en vue de les étendre au domaine financier. De plus nous étudierons les résultats obtenus et analyserons quelles améliorations apportent un gain significatif dans la classification.

La séparation sera quelque peu différente. Certains articles cumulant plusieurs algorithmes, afin d'éviter les répétitions d'explications, il convient de les séparer par article afin de regrouper les améliorations.

De plus, tous les algorithmes ne sont pas forcément évalués dans les articles. Il est donc possible que certains comme les réseaux de neurones n'aient pas de valeurs.

1. Il est possible de découper notre ensemble initial afin d'atteindre ce critère.
2. Comme *Naive Bayes* ou SVM.

2.4.2 *A Machine Learning Approach to Automated Trading*

2.4.2.1 Introduction

Cet article montre un exemple d'application sur le marché des actions [18] [1]. Après avoir essayé deux approches :

— L'approche individuelle.
— L'approche par secteur.

La première partait de l'hypothèse que l'historique du prix d'une action contenait des motifs permettant la prédiction du prix futur. Pour prédire P_N l'auteur utilisait les N précédant prix, soit : $[P_{N-1}, P_{N-2}, ..., P_1]$. Il s'agissait d'un modèle simple qui ne prenait pas en compte les actions des entreprises concurrentes. Après les premiers résultats, l'auteur a conclu que cette approche était trop simple pour être utilisée, car les valeurs étaient significativement moins bonnes que celle de la seconde approche [18].

La seconde approche, quant à elle, repose sur la supposition que le prix d'une action dépend des autres actions, souvent concurrentes. Il a donc pris en compte l'historique des prix du sous-jacent évalué, mais également celui de ses concurrents en termes de marché. Les données possédées par l'auteur portent sur divers secteurs. Notamment celui de l'*utility*, de l'*energy* et de l'*information technology*. Le procédé est le suivant, pour déterminer si une action précise A_1 va augmenter ou diminuer, il va analyser le prix de l'action durant les N jours précédant mais également celui des M autres actions du secteur donné.

$$f(A_{(1,t_1)}, \ldots, A_{(1,t_N)}, A_{(2,t_1)}, \ldots, A_{(2,t_N)}, \ldots, A_{(M,t_1)}, \ldots, A_{(M,t_N)}) = A_{(1,t_0)}$$

Un autre point qu'il convient d'aborder porte sur l'ensemble d'entraînement. Afin d'entraîner et d'évaluer son programme, il a fallu partager le *set* de données.

Dans le cas contraire, il aurait été impossible de tester les algorithmes entraînés. Ou bien, ils auraient été évalués sur les mêmes données que leur entraînement. Ce qui n'est pas possible.

Le choix a donc été fait de partager l'ensemble de données en deux sous-parties. Une première contenant 80% des données qui sera dédiée à l'entraînement et une seconde avec 20% pour l'évaluation.

L'auteur a utilisé le S&P [2] pour sélectionner des actions. Les données du secteur *utility* en contiennent 29, le secteur energy 39 et le secteur *technology* 61. Tous les *ticks* entre le 02.01.2014 et le 01.02.2016 sont présents dans les données. La taille de l'historique est de quatre jours afin que les vecteurs d'attributs ne soient pas trop grand.

Le but est donc à partir des quatre premiers jours de la semaine, de prédire si l'action sera en hausse ou non le cinquième jour.

1. Ou *Stock market.*
2. Le *Standard & Poor's* est un indice calculé à partir de 500 grandes sociétés capitalisées dans les bourses américaines.

Les trois métriques utilisées sont :

— Le *True Positive Rate* est définit comme suit :

$$TPR = \frac{TP}{TP + FN}$$

Où TP sont les positif détectés positifs et FN les négatifs détectés positif.

— Le *True Negative Rate* est définit comme :

$$TNR = \frac{TN}{TN + FP}$$

— Le *True Rate* :

$$TR = \frac{TP + TN}{TP + TN + FP + FN}$$

2.4.2.2 Résultats

Cet article a pour avantage de tester de nombreux algorithmes et techniques. En analysant les résultats, des pistes peuvent être mises en évidences. Nous pourrons également déterminer un ordre de grandeur concernant la précision pour un problème donné.

Les résultats proviennent de l'article suivant [18] :

	TPR	*TNR*	*TR*
Utility	0.5595	0.4507	0.5235
Energy	0.4653	0.5369	0.5047
Information Technology	0.5244	0.5031	0.5102

TABLE 1 – Tableau de résultats pour l'algorithme *Lasso Logistic Regression* (voir 2.3.2).

	TPR	*TNR*	*TR*
Utility	0.5699	0.4624	0.5179
Energy	0.4524	0.5320	0.5042
Information Technology	0.5075	0.54966	0.5052

TABLE 2 – Tableau de résultats pour l'algorithme *Decision Tree* (voir 2.3.3).

	TPR	*TNR*	*TR*
Utility	0.5949	0.4957	0.5495
Energy	0.4797	0.5812	0.5193
Information Technology	0.5115	0.5048	0.5091

TABLE 3 – Tableau de résultats pour l'algorithme *Naive Bayes* (voir 2.3.4).

	TPR	*TNR*	*TR*
Utility	0.5804	0.5081	0.5562
Energy	0.4818	0.6049	0.5201
Information Technology	0.5142	0.5040	0.5149

TABLE 4 – Tableau de résultats pour l'algorithme *SVM* (voir 2.3.5).

À ce stade, nous remarquons que les performances de la régression logistique et des arbres de décisions sont du même ordre de grandeur. Ces derniers étant battus par les algorithmes *Naive Bayes* et *SVM*.

Un autre point important concernant le fait que les algorithmes détectent mieux les *TPR* que les *TNR*[1]. Cela est dû au cours des actions des différents secteur globalement en hausse dans les données prises en compte. Du coup, de par le manque de données à la baisse dans l'ensemble d'entraînement, l'apprentissage est limité. Concernant le secteur de l'énergie, vu qu'il était en baisse, l'inverse se produit.

Afin d'améliorer cet aspect l'auteur a utilisé le *majority vote* avec les quatre algorithmes[2] :

	TPR	*TNR*	*TR*
Utility	0.5807	0.4921	0.5573
Energy	0.4753	0.6003	0.5192
Information Technology	0.5133	0.5055	0.5233

TABLE 5 – Tableau de résultats pour le *majority vote* (voir 2.3.8).

Les résultats sont similaires à ceux de *Naive Bayes* et de *SVM*, cela n'est donc pas très concluant comme amélioration. Il est possible que les deux algorithmes ayant les meilleurs résultats sont souvent d'accord sur la classe, cassant ainsi l'intérêt du vote.

L'auteur va utiliser l'algorithme *Naive Bayes* afin de l'améliorer. En se concentrant sur un seul algorithme, il lui est plus facile d'en voir les effets[3].

	TPR	*TNR*	*TR*
Utility	0.5949	0.4957	0.5495
Energy	0.4797	0.5812	0.5193
Information Technology	0.5115	0.5048	0.5091

TABLE 6 – Tableau de résultats pour l'algorithme *Naive Bayes* (voir 2.3.4) avec une optimisation par *ROC curve analysis* (voir 2.3.4).

1. Exception faite du secteur énergie.
2. Source des résultats : [18]
3. Source des résultats : [18]

Le fait d'utiliser la *ROC curve analysis* afin d'obtenir le seuil optimal permet principalement d'augmenter la performance dans la détection des négatifs. Il est important de mentionner que cela n'influence que peu la détection des positifs car ils sont bien représentés dans l'ensemble d'entraînement et donc mieux reconnus. L'amélioration est donc intéressante car sans augmenter le risque de sur-apprentissage, elle augmente la qualité de classification.

	TPR	*TNR*	*TR*
Utility	0.6021	0.5187	0.5779
Energy	0.4574	0.5871	0.5108
Information Technology	0.5348	0.5317	0.5382

TABLE 7 – Tableau de résultats pour le *Random Subset* (voir 2.3.9) avec l'algorithme *Naive Bayes* (voir 2.3.4) optimisé par *ROC curve analysis* (voir 2.3.4).

L'amélioration est assez importante par rapport à l'algorithme *Naive Bayes Roc curve analysis*. Le *Random Subset* obtient 58% et 54% pour l'*utility* et l'information technology, pour seulement 55% et 51% au *Naive Bayes ROC curve analysis*. Même s'il y a une légère perte pour le secteur *energy* d'environ 1%. L'ensemble procure de meilleurs résultats.

2.4.3 *Online Machine Learning Algorithms For Currency Exchange Prediction*

2.4.3.1 Introduction

Un autre exemple d'article utilisant les techniques de *machine learning*. Ce dernier ayant principalement exploré la technique de la descente du gradient (voir 2.3.7) et plus précisément de sa version probabiliste ou *stochastic gradient descent* [26].

Trois variations ont été implémentées et testées :

— La descente du gradient stochastique simple ou *Plain Stochastic Gradient Descent*[1].
— La descente du gradient stochastique avec choix aléatoire ou *Stochastic Gradient Descent with random sample picking*[2].
— La descente du gradient stochastique avec choix aléatoire et "départ chaud" ou *Stochastic Gradient Descent with random sample picking and warm start*[3].

Le *random sample picking* est une technique qui consiste à choisir aléatoirement un élément d'un ensemble de manière non uniforme. Dans ce cadre précis, il est intéressant d'accorder plus de valeurs à une donnée récente qu'à une plus ancienne. On considère que les éléments proches de nous d'un point de vue temporel, ont une plus grande influence.

La meilleure manière d'appliquer cela est d'utiliser une sélection aléatoire exponentielle.

1. Noté : *plainSGD*.
2. Noté : *approxSGD*.
3. Noté : *approxWarmSGD*.

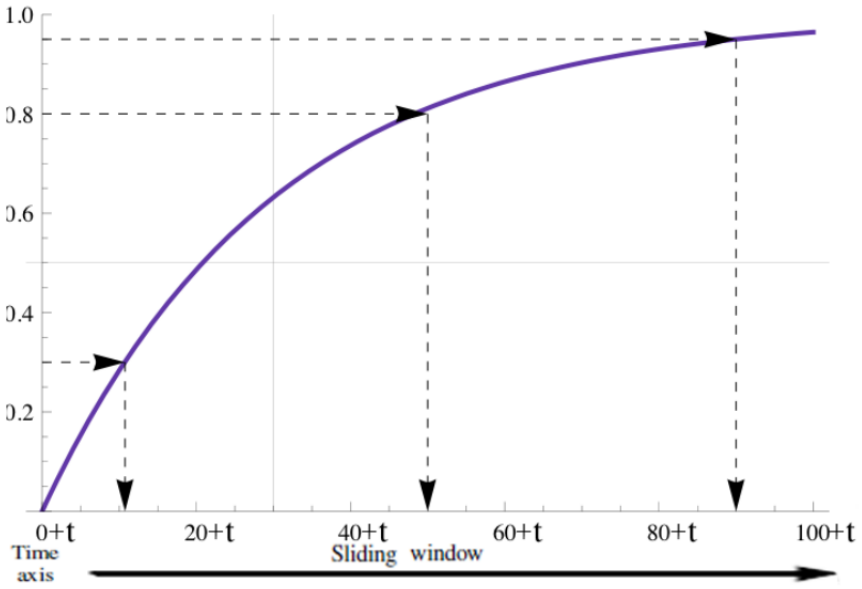

FIGURE 10 – Exemple de distribution exponentielle [1]. Cela va donc permettre de choisir les points les plus proches avec une probabilité plus grande.

L'autre point important est le *warm start*. C'est une pré-optimisation. Il s'agît d'initialiser les paramètres du gradient d'une certaine manière afin d'augmenter la vitesse et les chances de convergence. Pour ce faire, la meilleure option trouvée dans l'article consiste à utiliser les valeurs obtenues lors de la précédente itération [26]. Par conséquent, l'algorithme convergera plus vite pour un coût plus faible en termes d'erreur de calcul.

La métrique d'évaluation est différente des notions de TPR, TNR et TR. Dans cet article, le but est de prédire le prix d'un cours, comme EUR/USD à partir de plusieurs autres, comme EUR/CAD, EUR/AUD, EUR/GBP. Il est donc extrêmement compliqué de calculer avec plusieurs décimales le résultat exact, impliquant donc des taux nuls pour chacune des métriques. Il fallu donc trouver une méthode basée sur l'erreur relative :

$$relative_{error} = \frac{\Delta x}{x}$$

Cela quantifie la différence entre le résultat calculé et le résultat réel tout en le pondérant.

2.4.3.2 Résultats

Tout comme l'article précédent, l'analyse des résultats et des algorithmes utilisés peuvent conduire à des pistes de recherches. Le but est de calculer un taux de change à partir de

1. Source : [26]

données. Pour obtenir ces résultats précis, l'auteur a pris le *Singapore Hedge Fund FOREX Data Set* ainsi que le *Capital K FOREX Data Set*. Les dates ainsi que la fréquence ne sont pas précisées dans l'article.

L'algorithme retourne deux valeurs : le *bid* et le *ask* à partir des données fournies. Ci-dessous, nous allons voir la précision de ces résultats. Les tableaux représentent la comparaison entre les valeurs calculées et les valeurs réelles, soit l'erreur relative.

Données	*plainSGD*	*approxSGD*	*approxWarmSGD*
	Bid/Ask	*Bid/Ask*	*Bid/Ask*
Singapoore with SW-1[1]	0.1366274551%	0.133887583%	0.2294997927%
Singapoore with SW/2[2]	0.1366274551%	0.09281447603%	0.1551851906%

TABLE 8 – Tableau de résultats[3] pour les différentes versions de *SGD* sur le *FOREX* de Singapore. Les différences entre le *bid* et le *ask* étant minimes, l'auteur ne les a pas consignées.

Les erreurs relatives sont très faibles. Les estimations sont donc très proches des valeurs réelles. De plus, on remarque que le changement de la taille de la SW peut avoir une grande influence. Sur le *plainSGD* cela ne change rien, on peut donc en conclure que les $N/2-1$ données supplémentaires ne sont pas significatives dans le calcul. Pour le *approxSGD* et le *approxWarmSGD*, le taux d'erreur diminue. Nous pouvons donc dire que la fenêtre était trop grande et, pire encore, ajoutait du bruit rendant le calcul moins précis.

D'un point de vue calculatoire c'est très intéressant car il est possible de diminuer la taille de la fenêtre et donc gagner en vitesse tout en gardant, voir en augmentant, la qualité des résultats.

Données	*plainSGD*	*approxSGD*	*approxWarmSGD*
	Bid/Ask	*Bid/Ask*	*Bid/Ask*
Capital K SW-1	0.01167%/0.0117%	0.0116%/0.0117%	1.502e-03%/1.554e-03%
Capital K SW/2	0.01167%/0.0117%	0.0314%/0.0313%	1.701e-03%/1.751e-03%

TABLE 9 – Tableau de résultats[4] pour les différentes versions de *SGD*.

Nous constatons qu'il n'y a pas de différence pour le *plainSGD* entre les deux tailles de fenêtres. On remarque cependant que pour les deux autres variantes, la diminution de la taille les pénalise. Cependant, l'ordre de grandeur des erreurs demeure faible. Il est donc important de savoir si nous voulons un chiffre précis ou obtenir le résultat de manière rapide. L'objectif aiguillera le choix pour l'une ou l'autre des fenêtres.

1. $SW-1$ signifie une fenêtre de choix de taille $N-1$.
2. $SW/2$ signifie une fenêtre de choix de taille $N/2$.
3. Source des résultats : [26].
4. Source des résultats : [26]

À noter, que le *approxWarmSGD* obtient de meilleures valeurs que ces concurrents et subit beaucoup moins les effets de la diminution de SW que le *approxSGD*.

2.5 Conclusion

Afin d'élaborer des algorithmes de *ML*, il est important de saisir les considérations théoriques. À partir de ces connaissances, le choix d'un algorithme est plus aisé. En effet, si vous voulez apprendre une fonction continue, mieux vaut utiliser des techniques de descente de gradient et dans le cas de données faiblement différentiables, l'utilisation d'un programme *SVM* avec un *kernel trick* est recommandée.

Cela est également valable pour sa mise en place. Les optimisations mathématiques possibles sont nombreuses et complexes, comme nous l'avons vu dans la partie (2.4), il conviendra donc de les comprendre afin de mieux cerner les contraintes et les gains lors de l'optimisation.

Ces éléments peuvent être vus comme une boîte à outils algorithmiques, le choix et l'utilisation des différents outils reviendra à la personne qui programmera. Ce sera cette dernière qui devra construire l'algorithme le plus adapté avec les données, les contraintes et les techniques à sa disposition.

Les considérations mathématiques ne sont pas les seuls éléments importants. Appliquer un algorithme sans chercher à comprendre les cas concrets peut mener à des programmes peu efficaces.

Dans chacun des articles, les auteurs avaient une connaissance et une compréhension du monde de la finance suffisante, pour améliorer les techniques en dehors du carde mathématique. L'approche par secteur [18] ou l'utilisation de méta-paramètres propre au domaine financier [4] en sont des exemples concrets. Chacun de ces éléments a sensiblement amélioré les performances des algorithmes auxquels ils ont été appliqués.

Une technique de *machine learning* n'est qu'une solution à un problème mathématique précis. L'ajout d'éléments, comme ceux cités auparavant, améliore la quantité d'informations disponible et précise l'équation mathématique à optimiser.

Il est important de lier ces deux parties pour obtenir des résultats optimaux.

Les valeurs peuvent être encourageantes même s'il convient de relativiser les résultats. Ces derniers ayant pu être obtenus sur des ensembles de données "faciles". Ce mot qualifie des périodes avec peu de changements ou peu de variations, ce qui facilite grandement la classification. Néanmoins, les taux d'erreurs sont faibles et la classification efficace. Cela démontre que les bons algorithmes appliqués avec une bonne connaissance du domaine fournissent des résultats satisfaisants.

Le choix d'implémenter l'algorithme de réseau de neurones provient des justifications suivantes :

— Les *Neural Nets* peuvent, avec assez de ressources [1], approximer n'importe quel fonction. De plus ils sont capables d'opérer des séparations non linéaires sans recourir au *Kernel trick*.
— Il est possible d'éviter les problèmes de sur-apprentissage, en modifiant les méta-paramètres. Cela nous permet de profiter de n'importe quel ensemble d'entraînement sans craindre que ce dernier pose problème.

Les algorithmes d'arbres de décision sont des classifieurs linéaires. Ce qui signifie qu'ils sont limités lorsque les données ne sont pas linéaires, réduisant de fait, la qualité de classification. On retrouve ce même problème pour la régression logistique. De plus les arbres de décision (voir 2.3.3) ont de gros risques de sur-apprentissage même avec un ensemble d'entraînement adapté, sans compter qu'ils gèrent mal les exemples en continus. En effet, lors d'un entraînement *online* [2] chacun des nouveaux éléments peut introduire des exceptions. Ce qui implique de reconstruire l'arbre, ce cas pouvant être très lourd en termes de calculs, il convient donc de l'éviter.

Le classifieur *Naive Bayes* aurait pu être une méthode de classification efficace [3]car ce dernier peut classifier les données non linéaires, ne présente pas de problème de sur-apprentissage et est adapté pour un apprentissage *online*. Il suffit de rajouter le dernier exemple dans l'équation pour mettre à jour l'apprentissage. La raison qui nous a poussé à choisir les réseaux de neurones plutôt que l'algorithme de *Naive Bayes* est que ce dernier a des performances limitées sur un grand ensemble de données. Si l'on prend des petits jeux de données, *Naive Bayes* est très efficace [4] [6]. Cependant si l'ensemble grandit, ces performances plafonnent, ce qui n'est pas le cas du *Neural Nets* dont les résultats augmentent à mesure que le nombre de données disponibles croît.

Au vu des raisons susmentionnées, un algorithme de réseau de neurones nous semblait un choix raisonnable tant pour les spécifications mathématiques que pour la manière de l'utiliser. Car le *FOREX* nous fournit suffisamment de données pour avoir d'excellent résultats, de plus, nous ne sommes pas limités par une classification linéaire. Le cas échéant, si les réseaux de neurones ne fournissent pas de résultats suffisants, il sera possible de réorienter notre choix sur une autre méthode. Cela nous laisse des pistes de recherches tant en cas de succès que d'échec.

1. Cela comprend le temps et les données.
2. Quand les exemples arrivent de manière continue.
3. i.e : Deviner si le cours va monter, descendre ou stagner.
4. Dans ce cadre précis, il est même meilleur que tous les autres algorithmes vus dans ce projet.

3 Projet

3.1 Introduction

Le but de ce projet est d'implémenter l'algorithme présenté dans l'article *"An automated fx trading system using adaptive reinforcement learning"* [4]. Notre question de recherche consiste à déterminer et à mesurer si les propositions informatiques fonctionnent ou non et, le cas échéant, de comprendre les raisons de ces résultats puis proposer des améliorations pour le faire fonctionner ou augmenter ces performances.

L'algorithme n'est pas totalement décrit car l'article est un *proof of concept* et n'a pas vocation à être publié comme une nouvelle méthode de *trading*. Il s'agît plutôt d'une piste éventuelle de recherches. Il faudra donc interpréter les éléments et interpoler ceux manquants. Une fois le programme créé, nous le testerons sur des données réelles afin d'estimer s'il est utilisable ou non d'après des critères de *back-testing*. Toutes les notions économiques, mathématiques et scientifiques nécessaires à la compréhension sont décrites dans l'état de l'art précédant cette partie. Néanmoins, les équations importantes seront introduites ou exposées à nouveaux dans un soucis de rigueur scientifique et pour mieux comprendre les implications.

Cet algorithme est composé de trois parties. La première qui constitue le cœur du *machine learning* présent, se base sur un réseau de neurones à une couche construite de manière à utiliser la *back-propagation* et le *feed-forward*. L'optimisation des poids se fait via une descente du gradient simple à partir de fonctions de coût utilisant au mieux les données économiques à dispositions. La deuxième couche utilise des notions financiers afin de déterminer si le signal obtenu par la couche une doit être utilisé ou non et comment limiter les risques et pertes. La dernière partie va permettre de réduire l'influence humaine en optimisant les métas-paramètres. Ces derniers sont des paramètres qui conditionnent le comportement du réseau de neurones, de la descente du gradient et de la deuxième couche. Comme le taux d'apprentissage ρ, le *stop loss* x, etc.[1] Ils ne varient pas à l'intérieur des dîtes parties et sont modifiés dans la troisième couche. Cela permet au programme d'avoir plus de flexibilité et d'adaptabilité par rapport au temps et aux nouveaux éléments rencontrés.

À cause du caractère empirique du projet, nous avons dû chercher et expérimenter afin de comprendre, d'implémenter, d'interpoler et de corriger les bouts d'algorithmes présentés dans le *proof of concept* [4]. Nous allons donc découper l'article de manière à pouvoir expliquer l'idée mais également les problèmes et nos améliorations. Dans un premier temps, nous exposerons chacune des trois couches avec une partie présentant le concept mathématique et une autre pour la manière dont nous l'avons implémentée. Ensuite, nous parlerons des problèmes rencontrés ainsi que les remédiations ou les pistes de solutions que nous avons trouvées. Puis, nous présenterons les résultats, ainsi que certaines métriques afin d'expliquer les performances et le comportement de l'algorithme. Pour finir, nous conclurons notre rapport en synthétisant les idées, les problèmes et les résultats et en proposant des améliorations ou des pistes pour des recherches futures.

1. Tous les métas-paramètres seront introduits et expliqués dans la section suivante.

3.2 Algorithme

Dans cette section, pour chacune des parties de l'algorithme, nous allons dans un premier temps expliquer son fonctionnement théorique et ses bases mathématiques, puis exposer la manière dont nous avons implémenter ces éléments. Une fois l'algorithme détaillé, nous présenterons les problèmes rencontrés ainsi que les remédiations proposées. Après nous passerons à l'analyse des résultats pour les différentes variantes afin de saisir les causes et les conséquences de nos choix.

3.2.1 Algorithme *Layer 1*

Le rôle du *Layer 1* est de fournir un signal. Dans l'article, ce signal peut avoir comme ensemble de valeurs : $\{-1, +1\}$. Le calcul du dit signal suit la formule suivante [4] :

$$F_t = sign(\sum_{i=0}^{M} w_{i,t} r_{t-i} + w_{M+1,t} F_{t-1} + v_t)$$

où r_t est le *return* au temps t obtenu par $r_t = p_t - p_{t-1}$, $w_{i,t}$ est le ième poids de l'itération t, v_t est un seuil et F_{t-1} le résultat de l'itération $t-1$.

Cette formule mathématique implémente un réseau de neurones. Il s'agît d'une multiplication vectorielle entre les poids (w) et les *returns*(r_i). La notation des poids, soit $w_{i,t}$, prend en compte l'itération courrante. En effet, $w_{i,t}$ n'est pas forcément équivalent à $w_{i,t+1}$. Les poids ont pour but de donner une pondération à chacun des éléments afin de fournir un résultat le plus "juste" possible, suivant certains critères[1]. Dans notre cas, le réseau de neurones dispose d'une seule couche mais cette couche est récurrente. Cela signifie que l'*output*[2] à l'itération $t-1$ est utilisé dans le calcul du signal de t. Cette construction permet d'augmenter la quantité d'information disponible par le réseau afin d'améliorer les résultats. Un seuil, v_t, est ajouté dans la formule afin de lisser les résultats et d'éviter des variations trop fréquentes.

Afin d'obtenir de bons résultats, il convient de trouver les valeurs de w_t et v_t qui vont maximiser une fonction de coût. L'optimisation choisie est une descente du gradient (2.3.7). Soit $\widehat{S}(t)$ définit comme :

$$\widehat{S}(t) := \frac{A_t}{B_t}$$

où $A_t := A_{t-1} + \eta(R_t - A_{t-1})$ et $B_t := B_{t-1} + \eta(R_t^2 - B_{t-1})$. De manière intuitive, A_t représente l'espérance et B_t la volatilité. Le but est donc d'obtenir une grand espérance avec peu de risque. La valeur théorique maximale de la fonction $\widehat{S}(t)$ doit tendre vers l'infini.

La fonction de coût est définie comme suit :

$$D_t := \frac{d\widehat{S}(t)}{d\eta}|_{\eta=0} = \frac{B_{t-1}\Delta A_t - \frac{1}{2}A_{t-1}\Delta B_t}{(B_{t-1} - A_{t-1}^2)^{\frac{3}{2}}}$$

1. Nous développerons ces critères plus loin.
2. Notre signal $\in \{-1, +1\}$

où $\Delta A_t := (R_t - A_{t-1})$ et $\Delta B_t := (R_t^2 - B_{t-1})$.Il faut appliquer un développement de Taylor à $\widehat{S}(t)$ en $\eta = 0$

Il reste encore à définir R_t, qui quantifie le gain au temps t :

$$R_t := F_{t-1} r_t - \delta |F_t - F_{t-1}|$$

où δ est le coût de transaction. La première partie $F_{t-1} r_t$ estime le gain en fonction de la position et du *return*. Plusieurs cas de figures se présentent :

— Si $sign(F_{t-1}) = sign(r_t)$, alors la multiplication de ces deux éléments donnera un nombre supérieur ou égal à 0.

— Dans le cas contraire, le résultat sera négatif.

Si nous devinons correctement la direction du cours, alors nous faisons un profit et dans le cas contraire une perte.

La deuxième partie $\delta |F_t - F_{t-1}|$ ajoute les coûts de transactions dans le calcul du profit. Cela découle du fait que prendre une position a un coût. Les différents cas sont :

— Si $F_t = F_{t-1}$ alors la différence est nulle et de même pour le coût de transaction.

— Dans le cas contraire, cela signifie que nous avons dans un premier temps fermé une position, puis ouvert une autre. Il y a deux transactions effectuées et donc 2δ.

Le profit n'est pas uniquement dépendant du signal et du *return* mais également du signal de l'itération précédente.

Le profit cumulé est définit comme la somme des profits individuels :

$$P_T = \sum_{t=0}^{T} R_t$$

Maintenant que nous avons défini toutes les équations, intéressons nous à l'optimisation des poids. La formule de mise à jour, nous donne :

$$w_{i,t} = w_{i,t-1} + \rho \Delta w_{i,t}$$

Où $\Delta w_{i,t} = \frac{dD_t}{dw_i}$. Donc les poids de l'itération t sont définis en fonction de $t-1$ et de la dérivé D_t par rapport aux poids w_i. Cette dérivée peut s'estimer à partir de :

$$\frac{dD_t}{dw_i} \approx \frac{dD_t}{dR_t} \left(\frac{dR_t}{dF_t} \frac{dF_t}{dw_{i,t}} + \frac{dR_t}{dF_{t-1}} \frac{dF_{t-1}}{dw_{i,t-1}} \right)$$

Avec $\frac{dF_t}{dw_{i,t}}$ qui peut également être estimé comme :

$$\frac{dF_t}{dw_{i,t}} \approx \frac{\delta F_t}{\delta w_{i,t}} + \frac{\delta F_t}{\delta F_{t-1}} \frac{\delta F_{t-1}}{\delta w_{i,t-1}}$$

Ces deux équations sont approximées sur la base d'une optimisation [22]. Cette dernière consiste à ne prendre en compte que le dernier retour R_t. Cela permet de construire un système très rapide et réactif permettant ainsi d'optimiser le programme en continue, au fil des nouvelles données.

On remarque que toutes les formules sont définies de manières récursives. Cela découle de la structure du réseau de neurones. En effet, comme ce dernier est récurrent, et comme l'itération $t-1$ influence celle de t, une définition récursive est plus naturelle qu'une définition itérative qui ferait intervenir des accumulateurs.

Nous avons expliqué plus tôt, qu'il existe deux versions de la descente du gradient (2.3.7). Une version normale, ainsi qu'une version stochastique. Les deux sont équivalentes, tant au niveau des résultats que des problèmes rencontrés [22]. Elles le sont également d'un point de vue mathématique [22]. L'optimisation peut s'effectuer en version stochastique, avec à chaque itération le calcul de $\Delta w_{i,t}$ comme vu précédemment Ou bien en utilisant tous les exemples et avec une somme :

$$\Delta w_{i,t} = \sum_{t=1}^{T} \frac{dD_t}{dR_t}\left(\frac{dR_t}{dF_t}\frac{dF_t}{dw_{i,t}} + \frac{dR_t}{dF_{t-1}}\frac{dF_{t-1}}{dw_{i,t-1}}\right)$$

Puis dans tous les cas, il faut mettre à jour par $w_{i,t} = w_{i,t-1} + \rho\Delta w_{i,t}$.

Dans les deux cas, nous pouvons appliquer une boucle d'optimisation, notée *epochs* dans la littérature. Cette dernière permet d'appliquer plusieurs fois la descente du gradient afin d'améliorer les résultats. Si la descente du gradient est notée $\widehat{g}$ et w nos poids cela revient à faire :

$$\widehat{g}(\widehat{g}(...\widehat{g}(w)))$$

Mathématiquement la descente va tendre vers un optimum et en chaînant les dérivations cela permet d'accélérer la convergence.

3.2.2 Implémentation *Layer 1*

De part le caractère récursif des équations, il faut définir les valeurs pour $t = 0$. Il s'agît d'un choix arbitraire, mais qui se base néanmoins sur des éléments concrets. Nous avons :

- $A_0 = 0$. Le profit moyen au temps 0 est nul car aucune transaction n'a été effectuée.
- $B_0 = 0$. La déviation standard est nulle également pour les mêmes raisons que A_0.
- $w_{i,0} \in [0, 1]$. Les poids sont initialisés de manières aléatoire et uniformément répartis sur l'ensemble $[0, 1]$. Il n'y a aucune raison de penser qu'un départ est, à priori, meilleur qu'un autre.
- $F_0 = 0$. Le signal de l'itération 0 est nul car on peut, sans information complémentaire, savoir s'il s'agît plutôt d'un signal *long* ou *short*.
- $\delta, \rho, \eta, x, y \in [0, 1]$. Les métas-paramètres sont tirés aléatoirement dans l'intervalle $[0, 1]$, car l'article ne donne aucune indication sur les valeurs optimales. On suppose donc que l'optimisation annule le caractère aléatoire et donne des résultats similaires malgré des valeurs initiales différentes.

Nous avons fait le choix de représenter le vecteur w et les *returns* comme un `Array<Double>`. À partir de là, l'implémentation du signal consiste à itérer sur les deux vecteurs, à multiplier les éléments correspondant et à sommer le tout. Nous y ajouterons le seuil v_t ainsi

que le résultat de l'itération précédente soit $w_{i,M+1}F_{t-1}$. Voici un pseudo code résumant le calcul :

```
acc = 0
for(i=0; i < returns.length; i++){
   acc += w[i] * returns[i]
}
return acc + vt + w[w.length - 1] * oldFt
```

Afin de produire un code décomposable, nous avons créé des objets avec un objectif métier clair. Un objet `Weights` est donc instancié, représentant les poids. Des méthodes y sont attachées comme celle de mise à jour des poids. Il dispose de plusieurs attributs :

- A_{t-1} afin d'utiliser le résultat de l'itération précédente dans le calcul de A_t, noté `Weights.oldAt`.
- B_{t-1} pour les mêmes raisons que A_{t-1}, `Weights.oldBt`.
- w dont nous avons déjà parlé plus haut, `Weights.coefficients`.
- $\frac{dF_{t-1}}{dw_{i,t-1}}$ pour le calcul de $\frac{dF_t}{dw_{i,t}}$, `Weights.oldFt`.

Le code implémente directement les équations que nous avons mentionnées plus haut. Nous calculons donc $\Delta w_{i.t}$ puis nous mettons à jour les valeurs des poids. La représentation de w et des *returns* ne change pas. Concernant les autres éléments, A_t,B_t,δ, ρ, η sont de type `Double`. Il y a cependant des éléments qui ne sont pas totalement défini, il convient donc de faire un choix. Ces trois équations posant problèmes sont :

- $\frac{dR_t}{F_t} = -\delta\frac{F_t - F_{t-1}}{|F_t - F_{t-1}|}$ qui n'est pas défini si $F_t = F_{t-1}$.
- $\frac{dR_t}{F_{t-1}} = r_t + \delta\frac{F_t - F_{t-1}}{|F_t - F_{t-1}|}$ non défini pour $F_t = F_{t-1}$.
- $\frac{dD_t}{dR_t} = \frac{B_{t-1} - A_{t-1}r_t}{|B_{t-1} - A_{t-1}^2|^{\frac{3}{2}}}$ qui est non défini pour $A_{t-1}^2 >= B_{t-1}$.

Pour résoudre cela, nous avons posé que si $F_t = F_{t-1}$, alors :

$$\frac{F_t - F_{t-1}}{|F_t - F_{t-1}|} = 0$$

Et si $A_{t-1}^2 >= B_{t-1}$, alors il faut que la valeur de $\frac{dD_t}{dR_t}$ soit très basse pour pénaliser cela[1].

La descente du gradient est très sensible au *learning rate*[2]. C'est un des paramètres critiques en vue d'un algorithme efficient [25]. Dans certains cas, les poids peuvent diverger et tendre vers l'infini. C'est un phénomène qui peut arriver dans des domaines et des projets différents [25]. Une solution possible pour pallier ce problème consiste à appliquer un facteur $f < 1$ afin d'empêcher les poids de croître trop rapidement et de tendre vers l'infini. La technique du gradient consiste à :

$$w_{t+1} = w_t + \rho dw_t$$

1. La valeur numérique est à l'appréciation du programme. Cela peut être `-100` ou `Double.MIN_VALUE`
2. ρ dans notre cas.

À cause l'addition, w peut devenir arbitrairement grand [1]. Pour limiter cela, si les poids dépassent un certain seuil, nous multiplions les composantes problématiques de w_{t+1} par f, ce qui aura pour conséquence de réduire leur valeur.

3.2.3 Algorithme *Layer 2*

Nous l'avons vu avec la partie sur le *layer 1*, ce dernier nous donne un signal. Ce signal ne possède que deux valeurs $\{+1, -1\}$. Le *layer 2* va utiliser ce signal si certaines conditions sont requises. Les deux conditions sont :

— Le seuil, ou *threshold*.
— Le *stop trailing loss*.

Le premier sert à vérifier la puissance du signal. En effet, la calcul du signal, avant application du signe, renvoie un nombre réel. Cela peut être -0.0005 comme $+1000$. Le but du seuil est de filtrer les signaux peu puissants. On suppose que si un signal est proche de 0, il ne faut apporter que peu de confiance à la direction initiée par le signal. Notre y représente le seuil, il y a deux cas :

$$|F_t| >= y \rightarrow \text{alors le signal est utilisé}$$

$$|F_t| < y \rightarrow \text{alors le signal n'est pas utilisé}$$

Il convient d'apporter une petite précision. En annonçant que le signal n'est pas utilisé, cela peut signifier deux choses :

— Nous gardons la position qui a été ouverte au temps $t-1$.
— Nous ne prenons aucune position.

Cela dépend de la manière dont l'algorithme est construit. L'article ne fournit aucune précision quant à la technique utilisée, c'est donc une choix dépendant de l'implémentation. Ne prendre aucune position revient à dire qu'à chaque position prise, nous la fermons, et donc il est possible de ne pas se positionner. Au contraire, si nous ne fermons la position que lorsqu'un signal contraire intervient, il est juste de garder la position au temps $t-1$.

Comme tout les métas-paramètres, il est important de trouver une bonne valeur pour y. En effet, si le seuil est trop bas, ce dernier n'aura aucun effet et les résultats seront similaires à un algorithme sans seuil. Par contre, si y a une valeur trop élevée, alors cela va bloquer trop de signaux qui auraient pu conduire à des gains. La valeur optimale va dépendre des *returns* rencontrés, il n'est donc pas possible d'optimiser à priori.

Le *stop trailing loss* est une sécurité. Cela permet d'éviter de dépasser un certain pourcentage de perte et donc de protéger les gains. C'est un mécanisme financier qui a été ajouté afin de compléter la partie informatique.

Lorsqu'on prend une position, il faut mémoriser le profit cumulé. Tant que la position ne change pas, nous allons vérifier ce profit cumulé. Deux cas de figure se présentent :

— S'il augmente. Cela signifie que nous avons fait des gains et qu'il faut remplacer la valeur mémorisé par le nouveau profit cumulé.

1. Dans le pire des cas, la croissance est exponentielle en doublant à chaque itération.

— S'il diminue. Nous avons subi des pertes, il faut donc vérifier que ces pertes ne dépassent pas un certain pourcentage.

Si les pertes dépassent effectivement le pourcentage, alors nous fermons la position et attendons de recevoir une position différente de celle ayant entraînée les pertes. Dans le cas contraire, nous ne faisons rien.

Voici l'effet que peut avoir un *stop trailing loss* sur une courbe de profit :

FIGURE 11 – Exemple de l'effet d'un *trailing stop loss*.

Nous remarquons dans l'exemple qu'il n'y a pas de grande perte et qu'une perte est souvent suivie d'une période plate. L'application d'un *stop loss* peut diminuer les pertes et surtout il empêche des comportements humains problématiques. En effet, il arrive qu'une personne prenne une position, que cette dernière commence à lui faire perdre de l'argent, mais qu'il la garde ouverte en espérant que les choses vont changer et s'améliorer. Le *stop loss* automatise et empêche ce biais cognitif. Néanmoins cela peut causer d'autres soucis. Lorsqu'une grande baisse survient, le cour descend, déclenchant le *stop loss*, ce qui va revenir à fermer la position et si de nombreux algorithmes réagissent comme cela, le cour va continuer de descendre à cause de la vente simultanée de nombreuses positions créant ainsi un cercle vicieux.

Dans le cadre de notre algorithme, le pourcentage de perte à ne pas dépasser est quantifié par la formule suivante :

$$loss < x * 0.0001$$

Les pertes doivent être plus faibles que x points de bases. Il faut optimiser x, car si sa valeur est trop grande, il n'aura aucun effet sur le profit cumulé. Dans le cas contraire, il deviendra presque impossible de faire du profit, car à la moindre perte, le système va bloquer une position qui aurait pu engendrer des gains à terme, de plus cela va encourager le système à changer dès que le *stop loss* est déclenché entraînant ainsi une augmentation des coûts de transactions.

3.2.4 Implémentation *Layer 2*

L'implémentation du seuil est simple à mettre en place. Le *layer 1* nous renvoie un `Double`, et il suffit de comparer la valeur avec notre y. Afin de gérer les métas-paramètres ensemble, nous avons créé un objet `Parameters` qui contient δ, ρ, η, x, y. Le pseudo-code du seuil ressemble à :

```
ft = computeFt()
if (math.abs(ft) < param.y) {
 /* traitement du signal */
 }
```

Nous avons mentionné le fait qu'il ait plusieurs manières de traiter le signal lorsqu'il est trop faible. Nous avons fait le choix de retourner :

```
ft = computeFt()
if (math.abs(ft) < param.y) {
 ft = 0.0
 }
```

De cette manière, en accord avec les équations, cela signifie que nous fermons la position courante. Cela reste cohérent car nous avons toujours un coût de transaction :

$$\delta|F_t - F_{t-1}| = \delta(1 - 0)$$

Et nous perdons rien car :

$$F_{t-1}r_t - \delta|F_t - F_{t-1}| = 0 - \delta|F_t - 0|$$

Concernant le *stop loss*, nous avons créé un objet `Position`, qui va mémoriser la dernière position ainsi que le profit cumulé. À partir de cela, voici à quoi ressemble l'implémentation :

```
    ft = computeFt()
// if we change the position we need to update the max pnl and the
    position
    if (ft != position.lastPosition) {
        position.maxPnl = position.currentPnl
        position.lastPosition = ft
        return ft
    } else {
        // else we need to check the diff
        val diff = position.maxPnl - position.currentPnl
        if (diff <= 0.0) {
            position.maxPnl = position.currentPnl
        } else if (diff > 0.0 && diff > params.x * 0.001) {
            res = Math.signum(0.0)
        }
    }
```

Dans la première partie, nous vérifions que la position est la même. Si ce n'est pas le cas, nous avons juste besoin de mettre à jour la nouvelle position et le profit cumulé maximum. Sinon, il convient de vérifier.

Si le profit cumulé a augmenté, nous mettons à jours. Et dans le cas contraire, nous vérifions et appliquons le seuil avec $x * 10^{-4}$. À l'instar, du seuil, nous avons choisi de renvoyer un signal neutre, pour les mêmes raisons mathématiques qu'énoncées plus haut.

3.2.5 Algorithme *Layer 3*

Cette dernière couche a pour but d'optimiser les métas-paramètres présents dans les couches inférieures. Cela empêche de devoir recourir à une intervention humaine pour trouver les meilleures valeurs possibles. C'est une solution assez courante dans le domaine du *machine learning* [5]. Cette méthode est souvent opposée à celle de la *grid search* [39] :

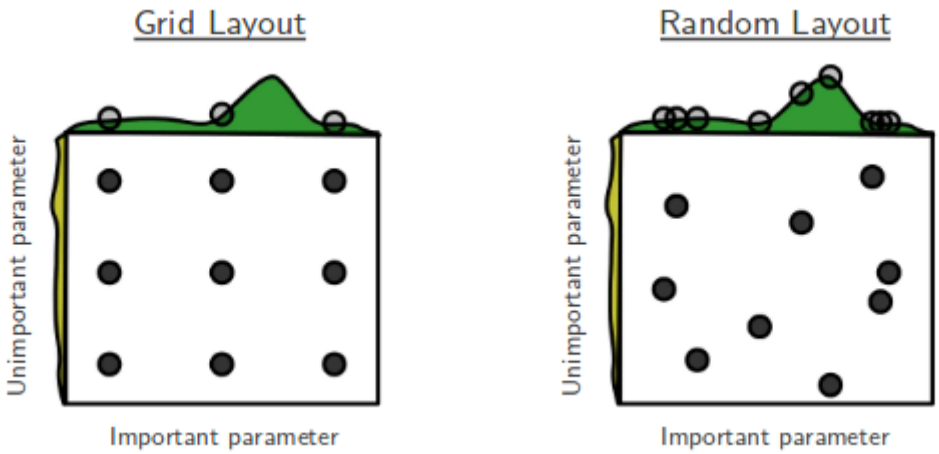

FIGURE 12 – Comparaison entre la méthode d'optimisation *grid search* et celle de *random search*

En moyenne, les maximaux locaux sont mieux explorés avec la méthode de recherche aléatoire. Pour mettre en place cet algorithme, il faut définir une fonction de coût, afin d'évaluer la qualité d'une solution. Les fonctions qui rentrent en compte sont les suivantes :

$$\Sigma := \frac{\sum_{i=0}^{N}(R_i)^2 I(R_i < 0)}{\sum_{i=0}^{N}(R_i)^2 I(R_i > 0)}$$

$$U(\bar{R}, \Sigma, \nu) := a(1-\nu)\bar{R} - \nu\Sigma$$

Où $\bar{R} := \frac{P_N}{N}$ est le profit moyen avec N le nombre d'éléments, ν l'aversion au risque et a est une constante. Dans la formule Σ, intuitivement on comprend que l'on veut un maximum de $R_i > 0$ et un minimum de $R_i < 0$, ce qui implique que la valeur optimale de cette formule vaut 0. Ceci est confirmé dans la deuxième formule U. En effet, on va soustraire $\nu\Sigma$, donc si Σ vaut 0, on maximise la fonction de coût. De plus $\bar{R}$ est défini comme le profit moyen, plus il sera grand, plus U sera grand.

La direction optimale à suivre est celle qui va maximiser les profits positifs tout en minimisant les profits négatifs, ce qui est intuitif. L'article mentionne d'autres raisons qui appuient ce choix [4] :

1. Les stratégies avec un grand risque et de nombreux *returns* négatifs sont pénalisés dans la définition de Σ.
2. Le fait de construire Σ comme un ratio permet d'obtenir des résultats similaires même si le volume augmente. Par exemple, pour une même stratégie qui serait stable, si l'on calcule avec 1'000 ou avec 1'000'000, le ratio serait similaire [1].

Le but étant de maximiser U, la fonction de coût devient :

$$\max_{\delta,\eta,\rho,x,y} U(\bar{R}; \Sigma : \delta, \eta, \rho, x, y; \nu)$$

Avec, pour rappel, δ le coût de transaction, η le paramètre d'adaptation, ρ le paramètre d'apprentissage, x le *stop trailling loss* et y le seuil. Malheureusement, cette fonction est trop complexe pour être optimisé de manière analytique. Il faut recourir à des heuristiques. L'article en propose une qui s'exprime sous cette forme :

$$\max_{\delta} \max_{\eta} \max_{\rho} \max_{x} \max_{y} U(\bar{R}; \Sigma : \delta, \eta, \rho, x, y; \nu)$$

Il s'agît d'une recherche aléatoire composante par composante. Ils fixent quatre paramètres puis ils tirent 15 valeurs aléatoires normalement distribuées autour de la valeur initiale, puis ils évaluent la fonction pour chacune des 15 valeurs et prennent la meilleure. Puis l'optimisation reprend avec le paramètre suivant. C'est une formule assez simple, qui a le mérite d'être peu coûteuse en ressources de calculs.

Comme pour le *Layer 2*, l'implémentation du *Layer 3* reprend fidèlement les équations proposées. Voici la manière dont fonctionne l'optimisation :

1. Nous sélectionnons un intervalle de *returns*.
2. Nous fixons 4 paramètres parmi les 5.
3. Nous tirons une valeur aléatoire normalement distribuée centrée sur la valeur initiale.
4. Pour tout l'intervalle, nous lançons le *Layer 1 & 2* avec les métas-paramètres.
5. Nous récupérons les valeurs importantes et nous calculons $U(\bar{R}; \Sigma : \delta, \eta, \rho, x, y; \nu)$.
6. Nous stockons le résultat.
7. Si nous avons terminé les 15 tirages, nous fixons la nouvelle valeur du paramètre avec le meilleur résultat [2] et retournons au point 2. Sinon nous reprenons au point 3.

1. De part notre hypothèse d'une stratégie stable
2. i.e : la meilleure valeur de $U(\bar{R}; \Sigma : \delta, \eta, \rho, x, y; \nu)$ parmi celles stockées du point 6.

Une fois que les cinq paramètres ont été optimisés, nous pouvons mettre à jour les nouveaux paramètres et reprendre l'entraînement du *Layer 1*. Les paramètres ont initialement une valeur aléatoire $\in [0, 1]$ car nous n'avons aucune indication sur leur valeur optimale. Nous avons donc choisi d'utiliser une distribution gaussienne :

$$\mathcal{N}(\mu, \sigma)$$

Où, μ est égale à la valeur initiale du paramètre et $\sigma = 1$. Ce sont des valeurs arbitraires. Nous estimons que chercher des valeurs avec une déviation standard de 1 permet d'explorer l'espace tout en évitant d'avoir des changements trop brutaux.

3.2.6 Implémentation *Layer 3*

Dans l'implémentation, les paramètres sont réunis dans une classe et c'est cette classe qui dispose des méthodes pour optimiser et mettre à jour les valeurs. De plus, par le caractère décorrélé des calculs pour les 15 tirages aléatoires, nous avons construit cela avec des *Thread*. Cela permet d'avoir de meilleur performance, car avec un *Pool* gérant les cœurs disponibles, on lance un *Thread* par valeur, pour un paramètre donné, et il suffit juste de synchroniser le moment où les *Thread* auront terminé et de prendre le meilleur résultat. De part le peu de synchronisation nécessaire, cela ne crée pas de perte de performance. Il est donc possible de gagner un facteur fac plus ou moins équivalent au nombre de cœurs engagés dans le calcul.

Afin de donner une idée du gain, utilisons des valeurs fictives. Soit le calcul pour un jeu de valeur t qui vaut 1, la comparaison afin de trouver la meilleur valeur *comp* vaut 0.5. Pour m paramètres et n tirages, cela donne dans notre cas $m = 5$ et $n = 15$:

$$mn(t + comp) = 5 * 15 * (1 + 0.5) = 112.5$$

Imaginons maintenant l'exemple avec $processus = 4$ en parallèles. Cela nous donne :

$$\frac{mn}{processus}(t + comp) + \epsilon = \frac{5 * 15}{4} * (1 + 0.5) = 28.125$$

Où ϵ est notre mécanisme de synchronisation, ce dernier reste négligeable de part la nature décorrélée des calculs. Le gain est très important tout en étant facilement *scalable*. En effet, l'ordonnancement des tâches est opéré par la machine virtuelle Java, ce sont donc des considérations hauts niveaux ne nécessitant pas de grande connaissances en parallélisme.

3.2.7 Points problématiques

Durant l'implémentation, nous nous sommes trouvés confrontés à plusieurs problèmes algorithmiques. Dans cette section, nous allons les détailler et présenter les solutions appliquées ou envisagées.

Le principal problème, qui nous empêche d'obtenir des résultats, est que les poids du réseau de neurones divergent durant la descente du gradient. Au bout de quelques centaines d'itérations, les poids croissent rapidement et deviennent tous `NaN`. Voici deux exemples de *runs* afin de montrer le comportement des poids :

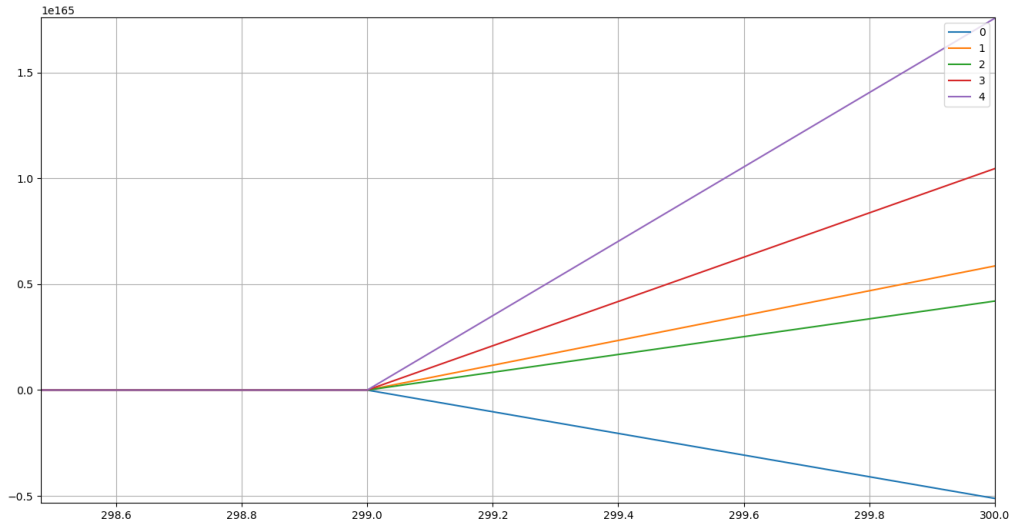

FIGURE 13 – Exemple de la divergence des valeurs des poids pour une *run* standard de l'algorithme, où l'axe des abscisses représente l'itération et l'axe des ordonnées la valeur.

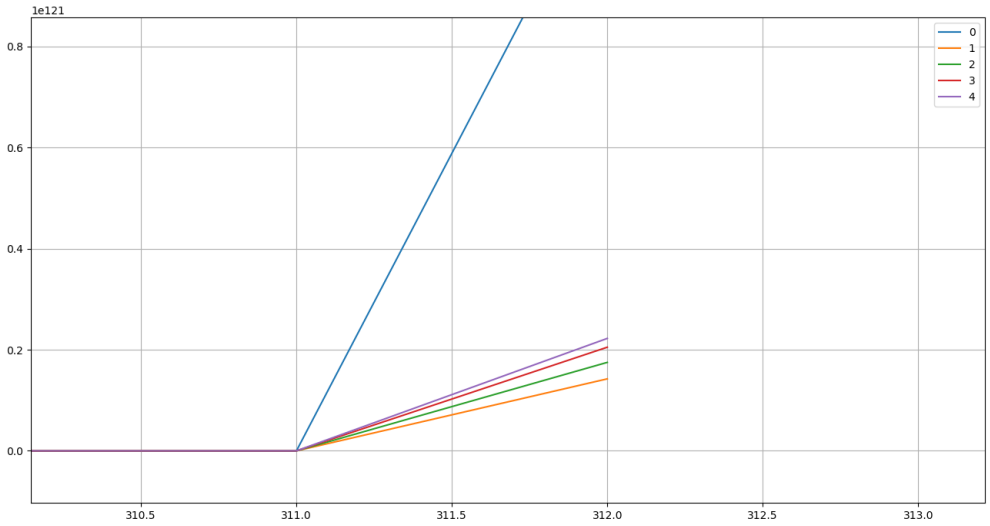

FIGURE 14 – Autre exemple de la divergence des valeurs des poids pour une *run* standard de l'algorithme.

Ce phénomène peut avoir pour cause un *learning rate* [1] trop grand. En effet, si ce dernier a une valeur trop importante, à chaque itération la valeur risque de croître et donc de diverger. Une solution intuitive consisterait à diminuer le taux afin d'éviter cela. Nous avons donc tenté cette méthode qui s'est révélée être infructueuse. Pour toutes les valeurs testées, pour la plupart comprises dans $[10^{-5}, 10^{-2}]$, l'algorithme avait deux comportements [2] :

— Pour les valeurs plus petites que 10^{-3}, le taux d'apprentissage était trop faible et les valeurs des poids ne changeaient pas.
— Pour les valeurs plus grandes que 10^{-3}, nous retrouvions un comportement divergeant.

L'article faisait mention d'un facteur $f < 1$ permettant d'éviter ces problèmes [4]. Il est possible de multiplier les poids, lorsqu'il dépasse une certaine valeur par le dît facteur, afin d'endiguer une croissance trop rapide. Le pseudo-code correspondant ressemble à cela :

```
for weight in weights
    if abs(weight) > some_threashold
        weight = weight * f
    endif
endfor
```

Ici encore, il y a un choix à faire sur la valeur de f, nous avons donc testé pour plusieurs cas. Il ressort deux cas de figures :

1. Dans notre cas, il s'agît de ρ.
2. Les valeurs sont indicatives mais cela peut varier.

— Si f a une valeur trop petite, alors les poids n'ont pas le temps de croître. Ils auront donc tendance à stagner ou à tendre vers 0.
— Dans le cas contraire, la multiplication ne fait pas assez diminuer la valeur des poids et ils tendent toujours vers l'infini.

Nous avons également tenté de combiner ces deux techniques en espérant que leur application conjointes permettent d'éviter ces problèmes, mais les résultats n'ont pas changé des situations mentionnées précédemment.

En essayant d'autres outils afin de voir si le problème persiste, nous nous sommes rendus compte qu'en utilisant une bibliothèque reconnue de réseaux de neurones [3], cette divergence existait également. Donc pour des données similaires, avec un algorithme d'optimisation similaire mais un réseau de neurones différent et une fonction de coût différente, nous arrivons au même problème. Nous sommes donc parti de l'hypothèse que le problème se trouvait dans la méthode d'optimisation, soit la descente du gradient. En changeant de méthode dans la bibliothèque, nous avons obtenu des résultats [1] plutôt bon.

Les algorithmes ayant données satisfactions sont [3] :
— Adagrad.
— Adadelta.
— RMSProp.

Nous avons donc lu la documentation et les publications afin de comprendre le fonctionnement de ces algorithmes. Et *in fine*, trouver les différences avec la descente du gradient qui pourraient expliquer les écarts de résultats.

En lisant les sources des algorithmes sus-mentionnés, nous nous sommes rendus compte de deux éléments très importants. Il s'agît de variantes de la descente du gradient et surtout, ils ont pour point commun d'avoir un taux d'apprentissage variable.

En effet, dans la descente du gradient, le taux d'apprentissage est fixé pour une *run* donnée, ce qui nous a posé les problèmes expliqués précédemment. Pour ces algorithmes, le taux d'apprentissage est modifié à chaque itération d'une *run* permettant plus de flexibilité, un apprentissage plus rapide et de diminuer le risque de divergence.

— Pour le RMSProp, il faut diviser le taux d'apprentissage par la moyenne des magnitudes des précédents gradients pour chaque poids [7].
— L'article sur l'Adadelta mentionne le fait que si l'on utilise [38] :

$$\Delta x_t = H_t^{-1} g_t$$

où H_t^{-1} est l'inverse de la matrice Hessienne et g_t le gradient, à l'itération t. Cela donne le poids optimal pour des problèmes quadratiques. Le coût de calcul étant trop important, l'article montre comment approximer cela en utilisant des heuristiques.
— Pour l'algorithme Adagrad, le taux d'apprentissage est calculé à partir des paramètres.

Il nous a semblé que l'algorithme RMSProp était le plus simple à mettre en place. Nous avons donc appliqué les formules suivantes [24] :

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$

1. i.e : Les valeurs des poids ne divergeaient plus.

où g est le gradient et $E[g^2]_t$ l'espérance du carré du gradient au temps t. À partir de cela :

$$\theta_{t+1} = \theta_t + \frac{\rho}{\sqrt{E[g^2]_t + \epsilon}} g_t$$

Nous avons donc un taux d'apprentissage qui varie en fonction des résultats précédemment obtenus, ce qui pourrait résoudre le problème rencontré avec la descente du gradient dans sa version simple.

Après avoir implémenté, cet algorithme, nous nous retrouvions également avec des poids qui divergeaient. En effet, de part les facteurs 0.9 et 0.1 le soucis persistait car l'espérance $E[g^2]_t$ n'était pas suffisamment grande pour compenser la croissance des poids. Nous avons donc changé les coefficients et testé plusieurs combinaisons. Nous avons obtenu de bons résultats avec 1 et 0.1 :

$$E[g^2]_t = E[g^2]_{t-1} + 0.1 g_t^2$$

La partie intéressante de cette méthode est la manière dont elle accélère l'apprentissage. En effet, si $E[g^2]_t < 1$, on a des changements minimes qui risquent de prendre du temps pour converger [1]. Cependant en divisant ρ par ce nombre, on va augmenter la "quantité de changement" rendant la convergence plus rapide. À contrario, si $E[g^2]_t > 1$, nous avons des changements conséquents et donc un risque de divergence [2]. Pour éviter cela, par la division, nous diminuons l'impact de ces modifications, diminuant *de facto* le risque de divergence.

Nous supposons, pour ce paragraphe, être dans un état où la descente du gradient et le *Layer 2* fonctionnent. Nous nous intéressons donc à la couche d'optimisation des métas-paramètres [3]. La liste de ces derniers est la suivante :

— ρ le taux d'apprentissage.
— η le paramètre d'adaptation.
— δ le coût de transaction.
— x le *trailing stop loss*.
— y le seuil.

Comme mentionné précédemment, l'optimisation de ρ perd de son intérêt de par l'utilisation de l'algorithme RMSProp conjointement à la descente du gradient. De plus, il paraît étonnant d'optimiser le coût de transaction δ. Cette valeur ne dépend pas d'un processus d'optimisation, il est étrange de vouloir la modifier.

À partir de là, nous allons travailler uniquement sur η, x, et y. En utilisant l'algorithme présenté plus haut, nous avons eu des problèmes de signal nul ou de `NaN`. Cela peut varier d'une *run* à l'autre. Ces phénomènes arrivent car dans le premier cas le seuil et/ou le *stop loss* sont trop élevés ne laissant aucun signal être accepté, et dans le second cas, les valeurs sont également trop grande, car qui provoque des erreurs de calculs en chaîne entraînant des poids divergeant malgré l'algorithme du RMSProp.

1. Avec la version classique de la descente du gradient.
2. Également avec la version classique de la descente du gradient.
3. Aussi noté hyperparamètres dans la littérature.

Pour palier ce problème pour $t = 0$, nous avons fixé les valeurs initiales plutôt que d'utiliser un tirage aléatoire. Pour $t > 0$ cependant, il faut modifier la déviation standard. Cette dernière, si la valeur est trop grande, risque de faire diverger les résultats de l'optimum. Les nouvelles valeurs des paramètres sont calculées de manière aléatoire suivant une distribution gaussienne :

$$x_{t+1} = \mathcal{N}(\mu, \sigma)$$

où $\mu = x_t$ et σ est initialisé à $\mu/10 + \epsilon$ afin d'éviter des sauts trop rapides. Cela permet d'explorer l'espace sans risquer de sortir de la zone d'attraction de l'optimum.

Nous soupçonnons le fait que les paramètres η et δ ont un impact trop grand dans le calcul de A_t et B_t ce qui pouvait entraîner les problèmes de poids. Pour éviter cela, nous avons fixé une valeur très faible, de l'ordre de 10^{-3}, ce qui fait disparaître le problème. De plus, malgré l'optimisation, les meilleurs résultats en moyenne sont obtenus avec les valeurs initiales pour x et y. Ces paramètres, lorsque x devient grand et y petit n'ont que peu d'influence sur la fonction de coût et donc il est compliqué de trouver de meilleures valeurs. De plus, nous ne connaissons pas le domaine dans lequel chercher les solutions ce qui handicape grandement la recherche. Si la déviation standard est trop importante cela fait diverger les valeurs et nous obtenons de très mauvais résultats [1] et si elle est trop petite, nous restons dans la zone d'attraction de la moyenne ce qui retire tout l'intérêt de l'optimisation. Nous n'avons pas de solution pour cette partie et il conviendrait de chercher une solution dans des recherches futures.

Afin de donner une vision d'ensemble, voici un pseudo-code résumant le schéma global du programme :

```
// initialization part
delta, rho, eta, x, y = Random()
weights = Random()
F[0] = 0                // the vector of signal
t = 1                   // the time
oldPrice = prices[0]    // the current prices
returns[0] = 0          // init the vector of return

// training phase with 2000 steps
for price in prices[1:2001]:
        // computing the return
        r = price - oldPrice
        oldPrice = price
        returns[t] = r
        // compute the signal
        F[t] = computedFt(t, w, F[t-1], returns, x, y)
        // update the weights
        w = updateWeights(returns[t-1], F[t-1], F[t], delta, rho, eta, x
            , y, returns)
        // recompute the F[t] with new best weights
        F[t] = computedFt(t, w, F[t-1], returns, x, y)
```

1. i.e : Un profit cumulé pouvant atteindre des valeurs aberrante comme des pertes de l'ordre de 100 unités.

```
        // if we reach the time to update parameters
        if (t % 500 == 0):
                delta, rho, eta, x, y = updateParameters(returns, w)
        t++

// for the testing phase, or the production phase
t = 1
pnl[0] = 0
// clean returns and F
F[0] = 0
returns[0] = 0
// the loop with index right after the training phase
for price in prices[2001:2501]:
        r = price - oldPrice
        oldPrice = price
        returns[t] = r
        F[t] = computedFt(t, w, F[t-1], returns, x, y)
        pnl[t] = pnl[t-1] + (F[t-1] * returns[t-1] - delta * |F[t-1] - F
            [t]|)

return pnl
```

3.2.8 Résultats

Refaire tous les graphiques.

Ajouter le grain des données.

Comparer les signaux entre eux sur deux runs.

Prendre les exemples déjà fait, mais décomposer en plusieurs graphiques (par ex diviser en 5, pour bien tout voir)

Afin de montrer l'aspect aléatoire des résultats, nous allons, pour chaque jeu de données, montrer deux exemples de *runs*. Nous avons choisi deux paires de monnaies différentes à des époques différentes afin de montrer sur des exemples très différents.

Les graphiques présentent trois éléments :

1. Le cours de change afin de visualiser la tendance.
2. Le signal. Cela permet de voir si le signal correspond à la tendance du cours ou de voir si le signal est nul.
3. Le profit cumulé.

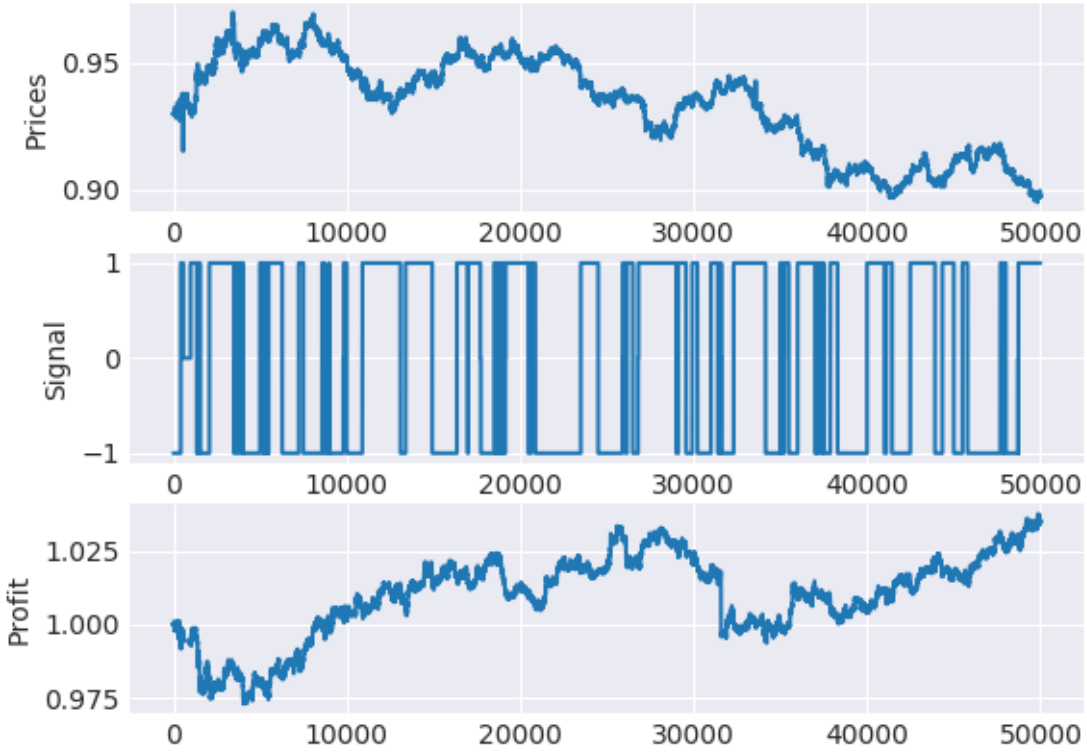

FIGURE 15 – Cours EUR/USD entre le 01.01.2000 et le 01.01.2001, avec $|w| = 20$, 2000 éléments d'entraînement, 500 de test, une optimisation des paramètres toutes les 200 itérations durant la phase d'apprentissage, $\delta = 0.001, \eta = 0.001, \rho = 0.01$ [1], $x = 0.005, y = 0.0001$.

1. Valeur de ρ avant l'optimisation par RMSProp

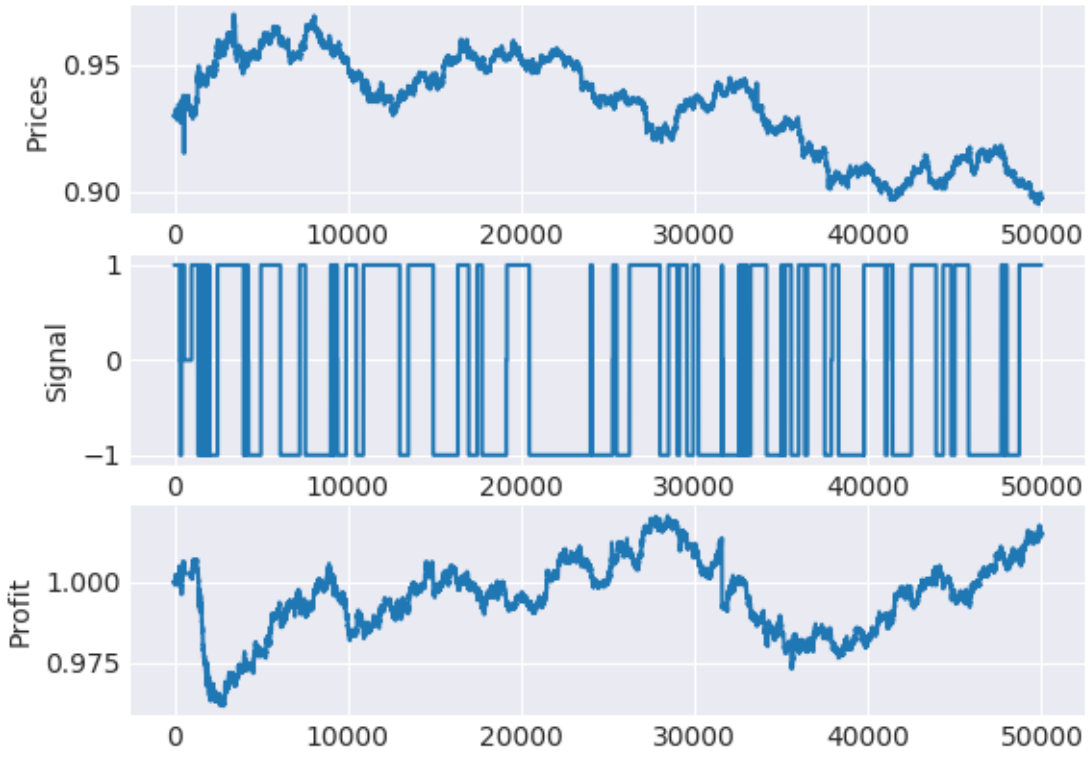

FIGURE 16 – Deuxième *run* du cours EUR/USD entre le 01.01.2000 et le 01.01.2001, avec $|w| = 20$, 2000 éléments d'entraînement, 500 de test, une optimisation des paramètres toutes les 200 itérations durant la phase d'apprentissage, $\delta = 0.001, \eta = 0.001, \rho = 0.01, x = 0.005, y = 0.0001$.

D'après ces deux exemples, les résultats peuvent varier d'une *run* à l'autre. Cela affecte le signal et par extension le profit cumulé. Il a peu de moment où le signal est nul à cause du *stop trailing loss*. Cela provient du fait que si la valeur de x est trop grande, nous ne dépassons jamais ce seuil et donc le signal n'est pas nul et si x est trop grand nous ne prenons presque jamais de position. Nous avons un problème récurrent qui concerne la valeur des métas-paramètres.

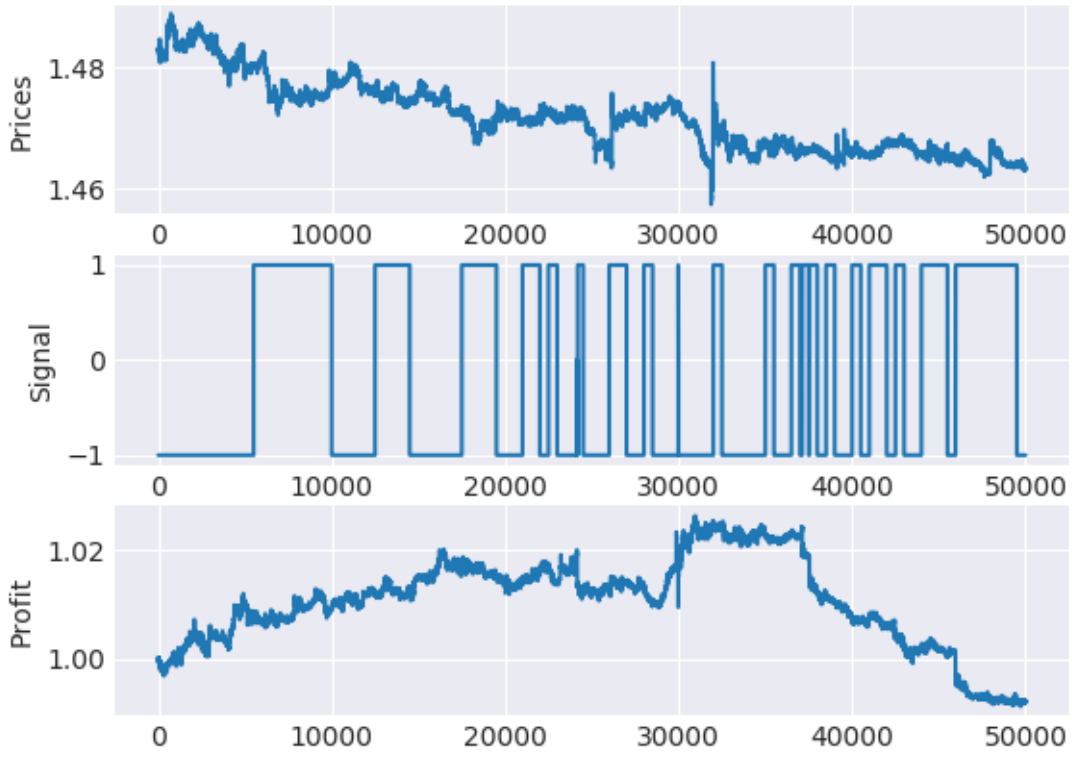

FIGURE 17 – Cours EUR/CHF entre le 03.01.2010 et le 30.12.2011, avec $|w| = 20$, 2000 éléments d'entraînement, 500 de test, une optimisation des paramètres toutes les 200 itérations durant la phase d'apprentissage, $\delta = 0.001, \eta = 0.001, \rho = 0.01, x = 0.005, y = 0.0001$.

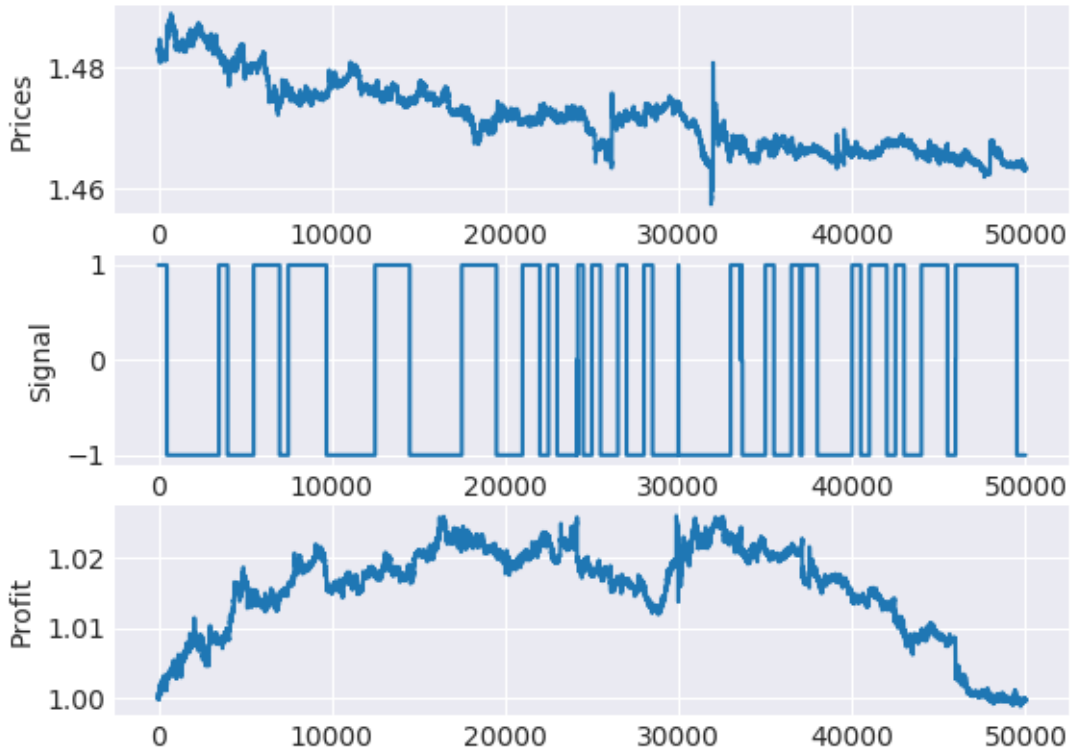

FIGURE 18 – Cours EUR/CHF entre le 03.01.2010 et le 30.12.2011, avec $|w| = 20$, 2000 éléments d'entraînement, 500 de test, une optimisation des paramètres toutes les 200 itérations durant la phase d'apprentissage, $\delta = 0.001, \eta = 0.001, \rho = 0.01, x = 0.005, y = 0.0001$.

Dans cet exemple, nous avons une tendance à la baisse avec un pic à un moment. Ce pic peut provenir d'une erreur dans les données. Nous pouvons confirmer les suppositions faîtes précédemment à savoir que :

— Il y a de l'aléatoire entre plusieurs *runs* pour de mêmes données.

— Le *stop trailing loss* a de la peine à se déclencher.

Concernant l'aléatoire, ce phénomène provient de l'initialisation des poids. Ces derniers étant générés via un `Random`, cela entraîne des différences entre deux générations.

Faire plusieurs boucles d'optimisation, genre 100, et ploter à chaque fin, le sharpe ratio.

Nous avons voulu observer l'évolution de *Sharpe ratio* pour diverses valeurs de η, afin de voir si le ratio est optimisé et l'impact du paramètre η dans ce ratio. Les graphiques suivants sont calculés à partir du cours EUR/CHF entre le 03.01.2010 et le 30.12.2011, avec $|w| = 20$,

2000 éléments d'entraînement, 500 de test, une optimisation des paramètres toute les 200 itérations durant la phase d'apprentissage, $\delta = 0.001, \rho = 0.01, x = 0.005, y = 0.0001$. Nous n'avons pris en compte les résultats de la première itération d'apprentissage. Soit $t = [0, 2000]$. Pour rappel le *Sharpe ratio* se calcule de la manière suivante :

$$\widehat{S}(t) := \frac{A_t}{B_t}$$

où $A_t := A_{t-1} + \eta(R_t - A_{t-1})$ et $B_t := B_{t-1} + \eta(R_t^2 - B_{t-1})$.

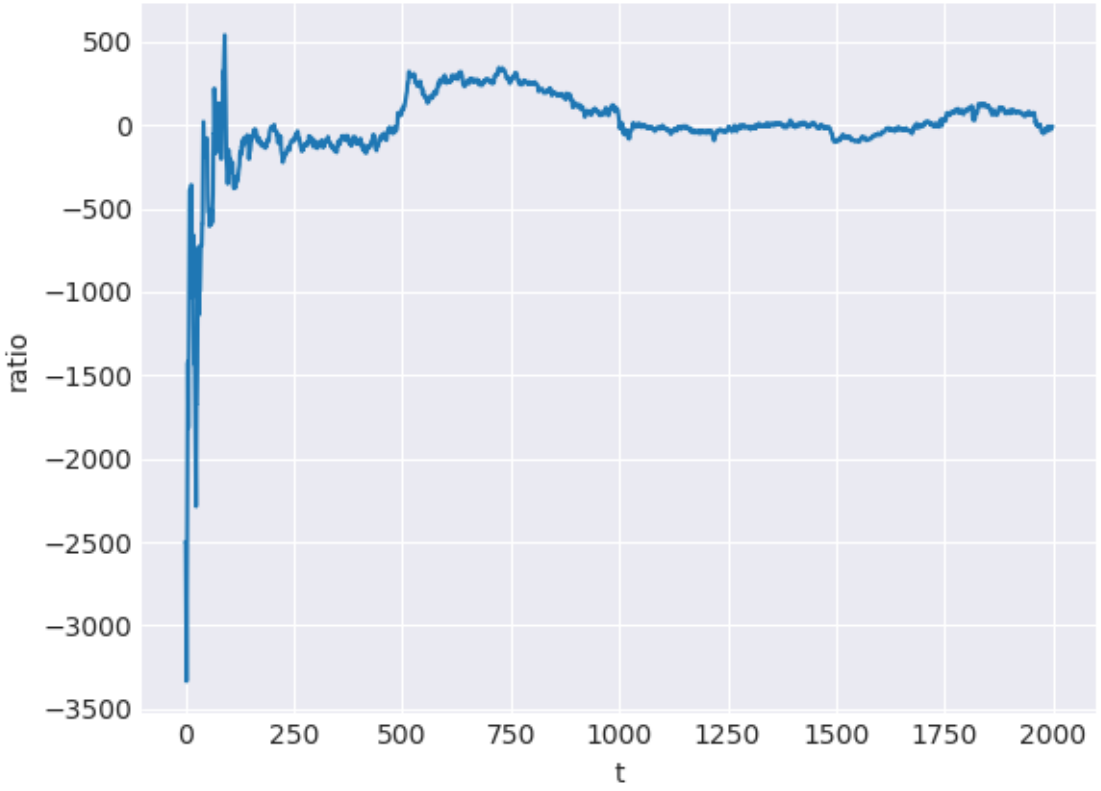

FIGURE 19 – Évolution du *Sharpe ratio* avec $\eta = 0.001$, $A_0 = 0, B_0 = 0$ de $t = 0$ à $t = 2000$

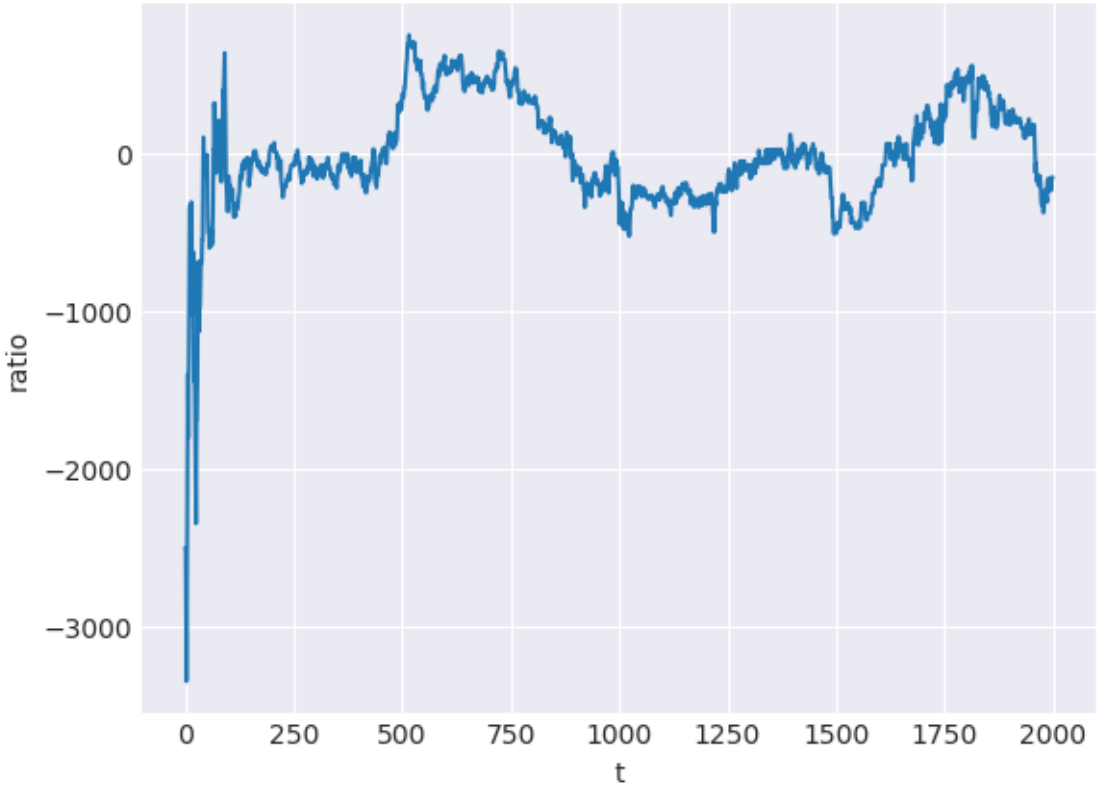

FIGURE 20 – Évolution du *Sharpe ratio* avec $\eta = 0.005$, $A_0 = 0, B_0 = 0$ de $t = 0$ à $t = 2000$

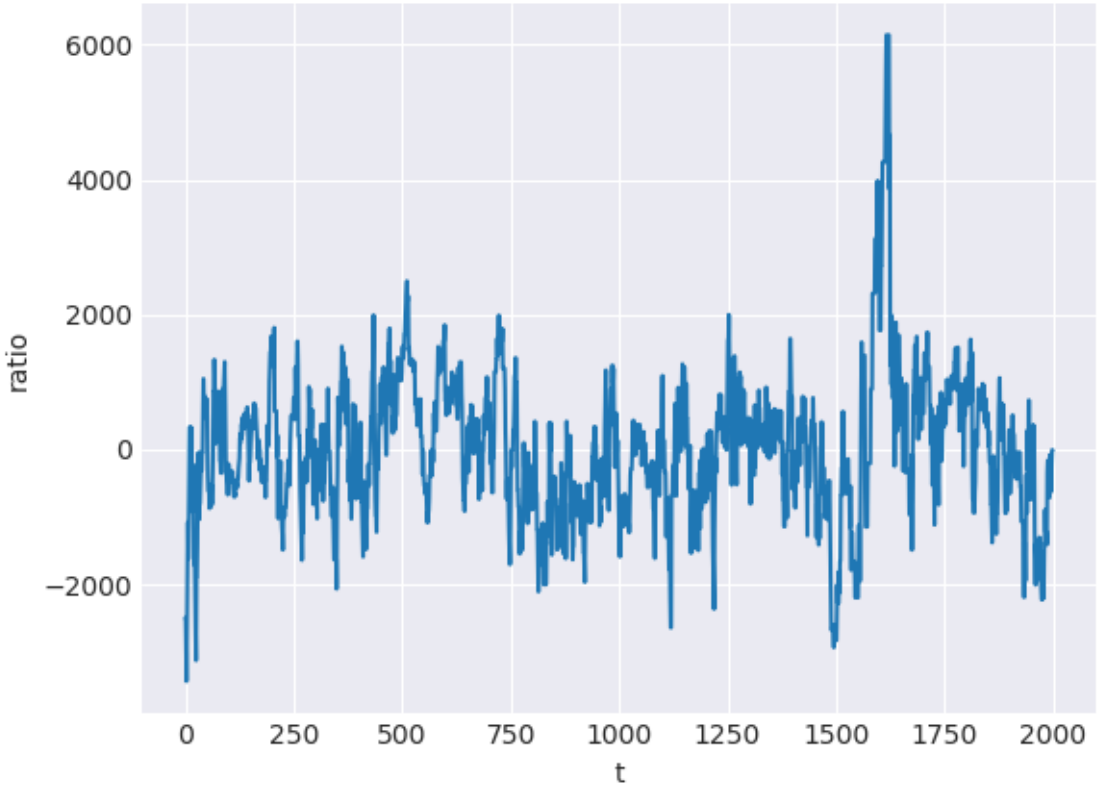

FIGURE 21 – Évolution du *Sharpe ratio* avec $\eta = 0.01$, $A_0 = 0, B_0 = 0$ de $t = 0$ à $t = 2000$

Ces graphiques nous montrent que si η est trop grand, il y a d'énormes variations dans le *Sharpe ratio* en plus de ne pas voir d'optimisation du dite ratio. Cette amplitude a un effet directe sur les poids, ce qui entraînera une divergence des poids. Si le *Sharpe ratio* est trop grand l'ajout du gradient sera trop important et le risque de `NaN` sera important.

On remarque également que si la valeur est faible,i.e : $< 10^{-3}$ alors l'algorithme tente d'optimiser correctement l'équation. Plus la valeur sera faible, plus la courbe sera lissée.

Avant la conclusion ajouter une discussion des performances et du temps : temps de calcul, phases d'optimisations, temps,...

3.3 Conclusion

Comme mentionné dans l'introduction, le but de ce projet était d'implémenter l'algorithme, de le faire fonctionner, de le tester et de trouver des pistes d'amélioration.

L'implémentation a été compliquée à cause du manque d'information dans l'article. L'algorithme n'y est que très peu décrit et, mis à part certaines équations, il n'y a aucun pseudo-code pour le comprendre. Il s'agît juste de descriptions vagues sur le fonctionnement

et l'idée du programme. À partir de cela, nous avons dû grandement interpréter et nous baser sur d'autres recherches théoriques afin de combler les manques. Il nous a fallu faire des compromis afin de faire fonctionner le programme, ce qui explique les divergences rencontrées avec l'idée initiale.

Ces divergences peuvent être prises comme des améliorations. En effet, l'implémentation initiale ne marchait pas et c'est grâce à ces idées que nous avons réussi à obtenir des résultats. Nous pensons principalement aux variantes de la descente du gradient qui ont permises de diminuer les risques de `NaN`.

En terme de résultats, contrairement à l'article, nous avons effectué nos mesures sur plusieurs cours à des périodes différentes. Un des gros problèmes provient des différences entre deux *runs*. Il est compliqué de dire si l'algorithme est performant ou non, car cela varie énormément. De plus, l'optimisation et la recherche de solutions sont ardues à cause de ce phénomène. Il nous est impossible de dire après un bon profit si cela est du à la chance ou non, et réciproquement avec les mauvais résultats.

On peut noter que malgré cela, l'algorithme trouve toujours le moyen de faire des profits indépendamment de la tendance ou des résultats précédant, ce qui est une satisfaction. En effet, ce phénomène implique qu'il y a effectivement apprentissage et que le réseau est capables sous certaines conditions de réussir à produire et interpréter un signal correctement. Afin d'améliorer le programme, il faudrait comprendre ce qui provoquer ces profits locaux et le généraliser afin d'obtenir un profit cumulé global croissant, ce qui n'est pas forcément le cas actuellement. En comprenant, le phénomène local de ce gain, on pourrait changer l'approche de l'algorithme et en accroître les performances.

En fixant les métas-paramètres [1], le *stop trailing loss* est moins visible que sur d'autres exemples. Ce qui nous permet de mentionner un problème récurrent dans ce projet : de part le manque d'information, les valeurs initiales de tout les paramètres sont extrêmement difficiles à trouver. La moindre petite variation peut aboutir à un crash ou à un profit `NaN`. Le caractère empirique du domaine de l'optimisation et du *machine learning*, ne nous permet pas d'avoir des valeurs standards fonctionnant avec n'importe quel algorithme. Cela revient donc à tester plusieurs valeurs en espérant qu'une fonctionnera.

Concernant la question de recherche, nous n'avons pas été capable de faire fonctionner l'idée originale. Cependant nous avons pu saisir les raisons ou éléments qui empêchaient son fonctionnement. Nous avons ensuite proposer des améliorations reposant sur ces constations. Ces dernières ont permis d'améliorer les performances de l'algorithme tout en reposant sur des bases mathématiques solides.

Concernant les pistes pour des recherches futures, nous en voyons plusieurs :
- Améliorer les métas-paramètres.
- Intégrer une version alternative de la descente du gradient.
- Proposer un réseau multi-couche.

1. À cause du problème soulevé dans la section *Layer 3*

Pour les métas-paramètres, il a plusieurs points à soulever. Il faudrait dans un premier temps trouver les domaines des valeurs afin de permettre une recherche de l'optimum plus aisée. De plus, si nous pouvions réduire le nombre de métas-paramètres, comme nous l'avons fait pour ρ avec l'algorithme RMSProp, cela permettrait d'améliorer la vitesse tout en proposant des méthodes d'optimisation plus précises et donc d'augmenter la performance global.

En utilisant, une version alternative de la descente du gradient, comme le RMSProp, le Adagrad ou le Adadelta, nous pourrions éviter certain soucis inhérent à la descente du gradient. Cette dernière est une méthode ancienne et toutes celles proposées sont des améliorations qui tendent à pallier les problèmes de la descente du gradient. De plus, cela pourrait avoir des effets bénéfiques sur l'ensemble du programme en diminuant le nombre de paramètres et donc la complexité interne de l'algorithme, sans faire de concession en terme de performances.

La construction d'un réseau de neurones n'est pas une science exacte. Cependant, le fait d'ajouter une ou plusieurs couches permettrait d'augmenter la vitesse d'apprentissage de ce dernier et de diminuer les erreurs. En sacrifiant du temps de calcul, car un réseau de neurones à plusieurs couches est plus lent qu'un autre à une seule couche, on pourrait deviner la direction plus aisément. De plus avec les gains de vitesse des propositions précédentes, les performances resteront bonnes.

Références

[1] Contributions to the problem of approximation of equidistant data by analytic functions.

[2] Vincent Cheung and Kevin Cannons. An introduction to neural networks. http://www2.econ.iastate.edu/tesfatsi/NeuralNetworks.CheungCannonNotes.pdf.

[3] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[4] M.A.H Dempster and V. Leemans. An automated fx trading system using adaptive reinforcement learning, 2004.

[5] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves.

[6] George Forman and Ira Cohen. Learning from little : Comparison of classifiers given little training. www.ifp.illinois.edu/~iracohen/publications/precision-ecml04-ColorTR-final.pdf.

[7] Nitish Srivastava Geoffrey Hinton and Kevin Swersky. Neural networks for machine learning.

[8] David Guerreiro. Chapitre 1 : Le marche des changes monnaie et finance internationales. https://economix.fr/docs/1045/chap_1_2015-16.pdf.

[9] Investopedia. American option. http://www.investopedia.com/terms/a/americanoption.asp.

[10] Investopedia. Backtesting. http://www.investopedia.com/terms/b/backtesting.asp.

[11] Investopedia. Bid and asked. http://www.investopedia.com/terms/b/bid-and-asked.asp.

[12] Investopedia. Currency carry trade. https://www.investopedia.com/terms/c/currencycarrytrade.asp.

[13] Investopedia. European option. http://www.investopedia.com/terms/e/europeanoption.asp.

[14] Investopedia. Forex. https://www.investopedia.com/terms/f/forex.asp.

[15] Investopedia. Option. http://www.investopedia.com/terms/o/option.asp.

[16] Anil K. Jain, Jianchang Mao, and K.M. Mohiuddin. Artificial neural networks : A tutorial. http://csc.lsu.edu/~jianhua/nn.pdf.

[17] Elad Hazan John Duchi and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization, 2011.

[18] Ning Lu. A machine learning approach to automated trading, 05 2016.

[19] Estelle Mermet. La règle des trois unités du marché des changes. http://www.forex.fr/newslist/8696-la-regle-des-trois-unites-du-marche-des-changes.

[20] Tom Mitchell. Descision tree learning.

[21] Tom Mitchell. Machine learning, 1997.

[22] John Moody and Matthew Saffell. Learning to trade via direct reinforcement, 2001.

[23] Ni, Jiarui, and Chegqi Zhang. An efficient implementation of the backtesting of trading strategies, 2005.

[24] Sebastian Ruder. An overview of gradient descent optimization algorithms. http://ruder.io/optimizing-gradient-descent/index.html#rmsprop, 2017.

[25] Tom Schaul, Sixin Zhang, and Yann LeCun. No more pesky learning rate.

[26] Eleftherios Soulas and Dennis Shasha. Online machine learning algorithms for currency exchange prediction.

[27] Financial Times. Real investors eclipsed by fast trading. https://www.ft.com/content/da5d033c-8e1c-11e1-bf8f-00144feab49a?mhq5j=e1, 2012.

[28] Jason Weston. Support vector machine (and statistical learning theory) tutorial. http://www.cs.columbia.edu/~kathy/cs4701/documents/jason_svm_tutorial.pdf.

[29] Addison Wiggin. Comment on est passé de l'étalon-or à l'étalon-dollar. http://la-chronique-agora.com/etalon-or-etalon-dollar/.

[30] Wikipedia. Line search. https://en.wikipedia.org/wiki/Line_search.

[31] Wikipédia. Algorithme du gradient. https://fr.wikipedia.org/wiki/Algorithme_du_gradient.

[32] Wikipédia. Algorithme génétique. https://fr.wikipedia.org/wiki/Algorithme_g%C3%A9n%C3%A9tique.

[33] Wikipédia. Broker. https://en.wikipedia.org/wiki/Broker.

[34] Wikipédia. Les accords de bretten woods. https://fr.wikipedia.org/wiki/Accords_de_Bretton_Woods.

[35] Wikipédia. Régression statistique. https://en.wikipedia.org/wiki/Regression_analysis.

[36] Wikipédia. Support vector machine. https://fr.wikipedia.org/wiki/Machine_%C3%A0_vecteurs_de_support.

[37] John Wiley and Sons. Algorithmic trading : Winning strategies and their rationale (wiley trading series), 2013.

[38] Matthew D. Zeiler. Adadelta : An adaptive learning rate method, Decembre 2012.

[39] Alice Zheng. Practical hyperparameter optimization : Random vs. grid search. https://stats.stackexchange.com/questions/160479/practical-hyperparameter-optimization-random-vs-grid-search, 2017.