Rendu de Reinforcement Learning

Eric PATARIN[*], Lucas SELINI[†]
M2 Data Science

March 5, 2024

Abstract

Dans cet article, nous proposons un cadre d'apprentissage par renforcement profond (DRL) multi-agents pour le trading algorithmique. Ce cadre est basé sur l'hypothèse du marché fractal (FMH) et utilise la coopération de plusieurs agents, chacun étant un expert du trading sur un intervalle de temps spécifique. L'objectif est d'apprendre les interactions entre les différents intervalles de temps et d'exploiter l'intelligence collective des agents. Le cadre fonctionne selon une structure hiérarchique dans laquelle le flux de connaissances va des agents qui tradent sur les intervalles de temps les plus élevés aux agents qui tradent sur les intervalles de temps les plus faibles, ce qui les rend plus robustes au bruit des séries temporelles financières. Nous utilisons un algorithme de Deep Q-learning pour entraîner les agents. La performance du cadre est évaluée par des expériences numériques menées sur un ensemble de données historiques de la paire de devises EUR/USD. Les résultats montrent que le cadre multi-agents, selon plusieurs mesures de performance basées sur le rendement et le risque, surpasse les agents indépendants et plusieurs stratégies de référence dans tous les intervalles de temps étudiés. La performance robuste du cadre multi-agents le rend adapté au trading algorithmique sur les marchés financiers.

[*]eric.patarin@ensta-paris.fr
[†]lucas.selini@ensta-paris.fr

Mots-Clés

Reinforcement Learning
Multi-agent
Algorithmic Trading
Multi-timeframe
Deep Q-learning

Contents

1 Introduction

Le présent rapport a pour objectif de faire comprendre point par point au lecteur les innovations apportées par Ali Shavandi et Majid Khedmati dans leur papier intitulé *A multi-agent deep reinforcement learning framework for algorithmic trading in financial markets.* [Ali22]

Historiquement la littérature s'attache majoritairement à étudier des techniques d'apprentissage supervisé aux séries temporelles que sont les prix des actifs sur les marchés financiers. D'une façon complémentaire, de nombreux acteurs financiers se sont mis à fréquemment employer des méthodes non supervisées comme l'analyse en composantes principales ou les K-means. Le Reinforcement Learning ne sauraient se classer parmi ces deux catégories : c'est dans ce contexte qu'il nous intéresse ici.
L'article que nous étudions se concentre sur le développement d'un algorithme de *Deep Q-Learning* Multi-Agent appliqué à des états représentant les prix possibles d'actifs financiers et à des actions représentant l'achat ou la vente des mêmes actifs.

2 Fondements Théoriques

2.1 L'apprentissage par renforcement Multi-Agent

Nous avons vu dans le cadre du cours de nombreux algorithmes concernant un unique agent. Le papier étudié ici considère un cadre différent : celui à plusieurs agents.

Reprenons tout d'abord l'illustration du Reinforcement Learning mono-agent issue du *Sutton & Barto* [Ric99] :

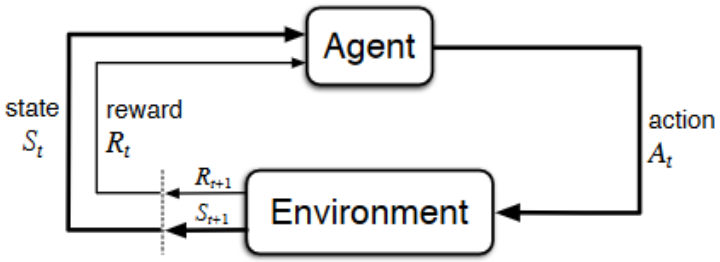

Figure 1: Le Reinforcement Learning Mono-Agent

Dans un état S_t donné, un agent effectue une action A_t et observe une récompense R_{t+1} ainsi qu'un nouvel état S_{t+1}. Nous considérons un cadre de travail un peu plus complexe dans notre étude :

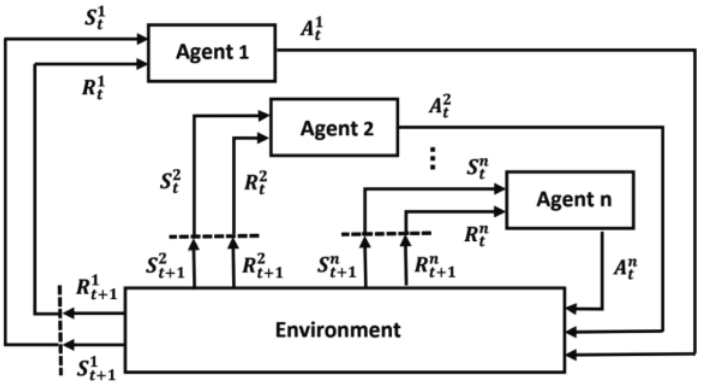

Figure 2: Le Reinforcement Learning Multi-Agent

Cette fois-ci, les états tout comme les récompenses peuvent varier selon les agents. Ce framework permet à l'étude de capturer plusieurs granularités temporelles et donc une meilleure compréhension du marché en agrégant les échelles. Intuitivement on comprend que les auteurs essayent d'approximer le fait que chaque acteur du marché raisonne à une échelle temporelle différente, et que les informations possédées par les agents afin de prendre une décision ne sont jamais véritablement identiques.

2.2 Formulation en MDP du Trading

Le Trading peut être formulé comme un problème de Reinforcement Learning puisque intuitivement le trader interagit avec le marché (**environnement**) en effectuant des transactions (**actions**) et observe un gain ou bien une perte financière (**reward**). Le trader est naturellement à la recherche d'une **politique** optimale afin de battre le marché.

Notons (S,A,P,R,γ) notre Processus de Décision Markovien (MDP).

2.2.1 Ensemble d'états

Les agents souhaitent savoir comment agir en fonction des prix observés. Une information communément analysée est la fameuse donnée "OHLCV" qui signifie Open High Low Close Volume *ie* Prix d'ouverture, Maximal, Minimal, Prix de fermeture et Volume d'une période donnée par exemple entre 16h00 et 16h01. C'est ce choix d'état qui est fait dans *A multi-agent deep reinforcement learning framework for algorithmic trading in financial markets* :

$$S_t = \big((\text{OHLCV})_t,(\text{OHLCV})_{t-1},\cdots,(\text{OHLCV})_{t-(n-1)}\big) \tag{1}$$

Naturellement, on peut considérer des périodes à de nombreuses granularités : 1min, 5min, 1h, 24h, 1 semaine. Une innovation par rapport au précédent état de l'art est la considération - grâce au framework Multi-Agent - de plusieurs granularités temporelles. Chaque agent va s'entraîner à une granularité différente. Ainsi les états considérés vont différer selon l'agent. Certains s'entraîneront sur des données OHLCV à l'échelle de la minute, d'autres sur une échelle de 24 heures.

2.2.2 Ensemble d'actions

On considère un marché où les agents peuvent trader un seul actif dans un marché à deux sens. L'espace d'actions le plus simple qui découle de cette hypothèse est le suivant : acheter, rester neutre, vendre. Ceci s'écrit :

$$a_t = \left\{ \begin{array}{l} 1\text{: Close the previous position (if any) and open a long position for tw,} \\ 0: \text{ Close the previous position (if any) and do nothing,} \\ -1: \text{ Close the previous position (if any) and open a short position for tw} \end{array} \right\} \tag{2}$$

où *tw* désigne une période de "freeze" minimale imposée à l'agent. Durant cette période, l'agent ne peut pas agir sur la position de trading qu'il vient d'ouvrir. On sait en effet qu'une fois une position ouverte sur le marché, il est rare qu'il soit optimal de la clôturer immédiatement, notamment à cause des frais de transactions qui ne sont pas pris en compte dans le modèle. Sans ce mécanisme, les auteurs constatent que les agents ont un volume d'actions bien trop élevé.

2.2.3 Fonction de récompense

On définit la reward function comme étant le return lié au trade pris. C'est probablement la quantité la plus étudiée en finance de marché. Il est parfaitement logique de l'employer car

c'est cette quantité que l'agent doit maximiser afin de maximiser son gain et la valeur de son portefeuille. Soit :

$$r_t = \frac{(p_{t+\text{tw}} - p_t)}{p_t} \times a_t \tag{3}$$

où p_t est le prix de fermeture (le C dans OHLCV) de l'actif traité au temps t, $a_t \in A$ le type de trade pris (où A désigne notre action set).

2.2.4 Probabilités de transition et Discount Factor

Le paramètre de Discount Factor $\gamma \in [0,1]$ sera optimisé au cours du processus d'apprentissage.

Les probabilités de transition représentent la dynamique du marché qui fait passer de l'état S_t à S_{t+1}. Le papier travaille sans modèle (Model-free RL) et fait l'hypothèse que ces probabilités seront calculées par les agents eux même.

Q : model free mais on pose une mdp et on va calculer les probas de transition ? normal ?

2.3 Deep Q-learning

2.3.1 Q-learning

Inventé en 1992 par Watkins et Dayan [Chr92], le *Q-Learning* est un algorithme off-policy principalement caractérisé par l'équation suivante :

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{4}$$

où Q désigne la table de valeur apprise, A_t l'action au temps t, R_t la récompense observée suite à l'action A_t, γ le facteur d'actualisation, et α celui d'apprentissage. Le *Q-Learning* fait partie des algorithmes dits TD *ie* utilisant la différence temporelle entre états pour la mise à jour.

Presque tous les algorithmes d'apprentissage par renforcement impliquent l'estimation de fonctions de valeur - fonctions d'états (ou de paires état-action) qui évaluent l'intérêt pour l'agent d'être dans un état donné (ou l'intérêt d'effectuer une action donnée dans un état donné). La notion de "qualité" est ici définie en termes de récompenses futures qui peuvent être attendues ou, pour être précis, en termes de rendement attendu. Bien entendu, les récompenses que l'agent peut s'attendre à recevoir à l'avenir dépendent des actions qu'il entreprendra. En conséquence, les fonctions de valeur sont définies par rapport à des façons particulières d'agir, appelées politiques.

L'idée du *Q-learning* est d'effectuer une moyenne pondérée entre la valeur précédente de la Q-value et le nouveau gain attendu, issu de l'équation de Bellman [Bel66].

Un intérêt de cet algorithme d'apprentissage est qu'il est indépendant de la policy choisie. Les garanties théoriques tiennent en conséquence indépendemment du choix de cette dernière. Ceci peut en un sens être vu comme une garantie de robustesse, qui nous permettra d'expérimenter différentes politiques lors nos expérimentations.

L'algorithme que nous utilisons est le suivant :

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q($ terminal, $\cdot) = 0$
Loop for each episode:
Initialize S
Loop for each step of episode:
Choose A from S using policy derived from Q (e.g., ε-greedy)
Take action A, observe R, S'
Do :

$$Q(S,A) \leftarrow Q(S,A) + \alpha \left[R + \gamma \max_a Q\left(S', a\right) - Q(S,A)\right]$$
$$S \leftarrow S'$$

until S is terminal

2.3.2 Deep Q-learning

Le problème principal auquel se heurte le *Q-learning* dans notre cadre d'applications est la taille de notre espace d'états. Nous rencontrons la fameuse "Curse of Dimensionality" (terme employé par Bellman en 1961) car l'espace des $(\text{OHLCV})_t$ est indénombrable. Nous souhaitons faire des calculs sur un espace d'état continu : nous allons estimer non pas un tableau en dimension finie mais une fonction Q ; la complexité serait beaucoup trop grande quand bien même le cardinal de notre ensemble d'actions n'est que de 3. En plus de ce problème de complexité, il est fort probable avec un espace d'états aussi vaste que l'algorithme ne converge jamais : il est impossible pour l'agent d'explorer tout l'environnement.

C'est ici que le *Deep Q-Learning* entre en jeu. Au lieu de $Q(s,a)$ on estime $Q(s,a,\theta)$ où θ est la matrice des paramètres d'un réseau de neurones. Le défi est de comprendre comment l'on peut se ramener à une tâche d'apprentissage supervisé : quels sont les labels pour ces estimations de $Q(s,a,\theta)$?

Il suffit de considérer les labels optimaux y issus de l'équation de Bellman :

$$y = r_t + \gamma \max_a Q\left(s_{t+1}, a; \theta_{i-1}\right) \mid s_t, a_t \tag{5}$$

Une fois ces éléments à notre disposition, nous sommes ramenés à un problème de Deep Learning classique. Nous cherchons les paramètres optimaux θ^* en minimisant la fonction de loss MSE suivante :

$$\mathrm{L}\left(\theta_i\right) = \mathrm{E}\left(y - Q\left(s_t, a_t; \theta_i\right)\right)^2 \tag{6}$$

A chaque itération i, $L(\theta_i)$ peut être minimisé en utilisant une descente de gradient stochastique (SGD) qui s'écrit :

$$\nabla_{\theta_i} \mathrm{L}\left(\theta_i\right) = \mathrm{E}\left(r_t + \gamma \max_a Q\left(s_{t+1}, a; \theta_{i-1}\right) - Q\left(s_t, a_t; \theta_i\right)\right) \nabla_{\theta_i} Q\left(s_t, a_t; \theta_i\right) \tag{7}$$

où $\nabla_{\theta_i} Q\left(s_t, a_t; \theta_i\right)$ est calculé avec l'algorithme de rétropropagation.

2.4 Hypothèses

Les marchés financiers sont intrinsèquement complexes et la création d'un système de trading complet nécessite de nombreuses considérations. Supposons que le trading est un processus ; plusieurs questions doivent être prises en compte depuis le début, lorsque les données sont reçues et prétraitées, jusqu'à la fin, lorsque les signaux de trading sont exécutés sur le marché.

L'article propose et évalue un cadre DRL multi-agents pour générer des signaux de trading. En conséquence, plusieurs hypothèses ont été prises en compte pour simplifier le processus :

1. Il n'y a pas d'écart ou de "slippage" du marché. Par conséquent, les signaux sont immédiatement aux prix exacts déterminés par les agents.

2. Les frais de commission sont nuls ; les coûts de transaction n'interviennent donc pas dans la structure de la fonction de récompense.

3. La taille de l'ordre est de 100% du solde disponible pour l'ouverture d'une position de négociation.

4. Les agents ne négocient qu'avec leur solde ; il n'y a donc pas d'effet de levier.

5. Les actions des agents ont un impact négligeable sur les transitions du marché ; Par conséquent, l'impact sur le marché est considéré comme nul.

Ces hypothèses sont très fortes et seront discutées en conclusion.

Il ne faut pas oublier de mentionner la principale motivation pour une modélisation Multi-Agents à plusieurs échelles temporelles qui est l'hypothèse fractale des marchés (FMH). Cette hypothèse traite le marché comme un système chaotique dans lequel les investisseurs échangent à des horizons de temps différents sous l'impulsion d'informations et d'interprétations pouvant être très diverses. Entraîner plusieurs agents sur plusieurs granularités est dans ce contexte d'une logique implacable.

3 Algorithme et méthode

Deux algorithmes sont proposés dans l'article de Shavandi et Khedmati. Le premier est un *Deep Q-Learning* pour agents indépendants. Le second concerne le framework Multi-Agent.

3.1 Défis pratiques liés au Deep Q-Learning

Malheureusement, l'apprentissage par renforcement est plus instable lorsque des réseaux neuronaux sont utilisés pour représenter les valeurs d'action. L'entraînement d'un tel réseau nécessite beaucoup de données, mais même si ce critère est rempli, il n'est pas garanti qu'il converge vers la fonction de valeur optimale. En fait, dans certaines situations, les poids du réseau de neurones peuvent osciller ou diverger, en raison de la forte corrélation entre les actions et les états.

Afin de résoudre ce problème, nous présentons dans cette section deux techniques utilisées par le Deep Q-Network présenté dans l'article :

1. Reprise d'expérience ou *Experience Replay*
2. Réseau cible ou *Target Network*

Dans cette construction, l'algorithme de *Deep Q-Learning* se décompose en deux phases :

1. **Sampling** : on réalise des actions et on stocke les tuples d'expériences dans la *Replay Memory*
2. **Training** : on tire aléatoirement un nombre fini d'expériences en mémoire (un batch). On apprend de ce batch en utilisant une descente de gradient à chaque étape.

Voici un pseudo code illustrant ce mécanisme :

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Sampling

Training

Figure 3: Les deux différentes phases mises en évidence.

3.1.1 Experience Replay Memory

L'experience replay memory permet de stocker des expériences réalisées dans l'environnement. L'idée de cette mémoire est de permettre d'utiliser nos expériences d'une façon plus efficiente. On se permet de réutiliser ce qui a été joué afin d'entraîner le réseau de neurones. C'est essentiellement un buffer permettant d'utiliser plusieurs fois une même expérience dans le processus d'apprentissage supervisé. Cela empêche également que le réseau n'apprenne que ce qu'il a fait immédiatement avant.

Le *rejeu* d'expérience présente également d'autres avantages. En échantillonnant aléatoirement les expériences, nous supprimons la corrélation dans les séquences d'observation et évitons que les valeurs d'action n'oscillent ou ne divergent de manière *catastrophique**. [Hug23] Dans le pseudocode de Deep Q-Learning, nous initialisons une mémoire tampon de relecture D avec une taille de N où N est considéré comme étant un hyperparamètre.

* *L'oubli catastrophique* : Le problème que nous rencontrons si nous donnons des échantillons séquentiels d'expériences à notre réseau de neurones est qu'il a tendance à oublier les expériences précédentes au fur et à mesure qu'il acquiert de nouvelles expériences. Par exemple, si l'agent se trouve dans le premier niveau, puis dans le second, qui est différent, il peut oublier comment se comporter dans le premier.

3.1.2 Target Network

Un problème rencontré fréquemment dans l'entraînement d'un Deep Q-Network est la non stationnarité entre la Q-valeur cible et la Q-valeur estimée. A chaque étape de l'entraînement, les deux varient ensemble dû à leur corrélation (ils sont calculés avec les même poids) autrement dit on se rapproche sans cesse de la cible ; mais la cible évolue elle aussi de son côté. On a en effet caché le caractère non trivial de l'utilisation de l'équation de Bellman pour obtenir des labels, mais il faut tout de même savoir estimer la Q-valeur cible dans l'équation de Bellman. Or si l'on utilisait un seul et même réseau de neurones pour estimer la cible et l'estimée on aurait à se battre sans relâche : en mettant les poids du réseau à jour ce sont les deux termes corrélés qui sont modifiés et la convergence qui stagne ! C'est pour ce faire que l'on à recours à un deuxième réseau de neurones : le *Target Network*

Le *Target Network* est en conséquence un réseau de neurones secondaire qui vise à stabiliser l'entraînement du modèle et à en améliorer la convergence. La technique associée est communément appelée *Fixed Q-Target*. Afin d'éviter tout syncrétisme les réseaux sont régulièrement mis en communs en égalisant Q (estimée vanille) et $\hat{Q}$ (estimée de la cible / Bellman) toutes les C étapes. Le *Target Network* est en général mis à jour moins régulièrement que le réseau principal afin de garantir des cibles plus stables.

3.2 Le mécanisme de stabilisation d'actions

Comme précédemment mentionné, les auteurs notent que leurs agents ont souvent tendance à abuser de leur liberté d'actions. Or une position de trading est bénéfique à priori seulement si elle est tenue pendant quelques temps (à une granularité donnée) et ce encore plus dans notre cadre qui fait l'hypothèse qu'il n'y a pas de frais de transactions.

Une innovation apportée par Shavandi et Khedmati se nomme "Action Stabilization Mechanism" : après avoir choisi une action, l'agent doit geler durant une durée tw toute nouvelle décision. Voici une illustration assez explicite de ce mécanisme :

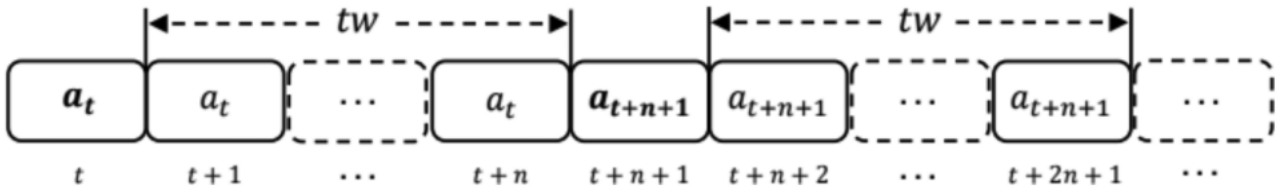

Figure 4: Mécanisme de stabilisation des actions

3.3 Obtention d'un signal de Trading par l'agrégation de décisions d'agents

Considérons n agents évoluant à des échelles de temps différents $(tf)^i$ tel que $(tf)^i < (tf)^{i+1} \forall i$
Notons l'ensemble d'agents :

$$\Lambda = \{agent_1, agent_2, \ldots, agent_n\}$$

Il est important de noter que plus la granularité temporelle diminue plus la qualité du signal (ici du prix) diminue : le bruit prédomine de plus en plus à mesure que nous diminuons tf. Pour autant, une plus grande échelle temporelle implique la possession de moins de détails par l'agent. Partant de ce constat, les auteurs décident de faire circuler l'information de trading de l'agent à la plus grande échelle temporelle vers celui à la plus faible. L'agent 1 reçoit en conséquence de précieuses informations robustes au bruit venant des autres agents, tout en ayant une finesse que n'ont pas les autres dans ses données.

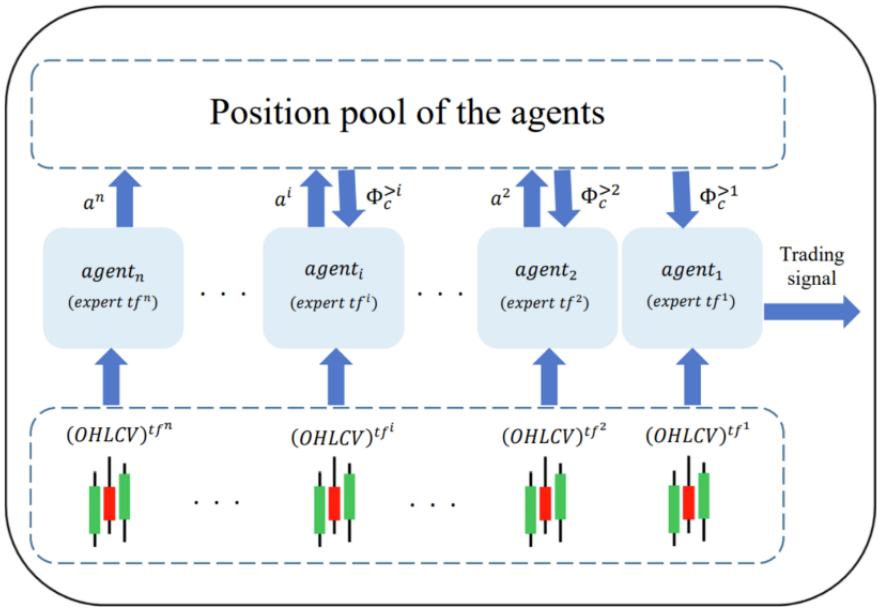

Figure 5: Illustration des interactions entre agents

Concrètement, l'$agent_i$ va recevoir dans ses réseaux de neurones $n-i+1$ nouveaux inputs que sont les positions prises par les agents qui le précèdent ; additionnellement à ses états $OHLCV$.

3.4 Algorithme 1 : Cadre Mono-Agent - Agents Indépendants

Le premier algorithme présenté est un Deep Q-Network classique, intégrant le Target Network et le Replay Memory mentionnés respectivement sous les notations $\hat{Q}$ et M.

Dans notre construction, chaque $Q_i \in R^{n \times |A|}$ où $|A| = 3$ est le cardinal de notre ensemble relativement simple d'actions et n la période de regard en arrière sur les prix. Notre batch est donc un tenseur de Q-valeurs de taille $Q \in R^{batchsize \times n \times |A|}$

On initialise aléatoirement les poids de nos deux réseaux de neurones ainsi qu'une mémoire de capacité N. A chaque étape c'est en considérant θ et non θ^- que l'on effectue la descente de gradient.

Algorithm 1: DQN Trading Algorithm for independent agents

Input: Experience replay memory and its size (N), Q network and its parameters, target network $\widehat{Q}$ and its parameters, market environment for the trading timeframe (S), the time window of action stabilization (tw), look back period (n), the initial value of exploration parameter (ε) and its decay rate, number of steps (C) to restart weights of the target network, discount factor (γ);

Output: Optimal parameters of Q network

Initialize the experience replay memory M to capacity N;
Initialize Q network with random weights θ;
Initialize target network $\widehat{Q}$ with weights $\theta^- = \theta$;
Set terminal = last time step in the OHLCV dataset (market environment);
Set tw = time window of action stabilization;
Set $w = 0$; Set n = look back period;
for $t = n$, *terminal* $-tw$ **do**
  if $w = 0$ **then**
    With probability ε, select a random action a_t from $\{1,0,-1\}$;
    Otherwise, select $a_t = \text{argmax}_{a \in A} Q(s_t, a; \theta)$;
    $a_c = a_t$;
    $w = tw$;
  else
    $a_t = a_c$
  Perform a_t and get a new observation $s_{t+1} =$ $((\text{OHLCV})_{t+1}, (\text{OHLCV})_t, \cdots, (\text{OHLCV})_{t-n+2})$;
  Get the reward $r_t = \frac{(p_{t+\text{tw}} - p_t)}{p_t} \times a_t$;
  Store experience $e_t = (s_t, a_t, r_t, s_{t+1})$ in M;
  Randomly sample a minibatch of experiences (s_i, a_i, r_i, s_{i+1}) from M;
  Set $y_i = r_i + \gamma \max_a \widehat{Q}(s_{i+1}, a; \theta^-)$;
  Perform a gradient descent step on $(y_i - Q(s_i, a_i; \theta))^2$ considering network parameters θ;
  Reset $\widehat{Q} = Q$ every C steps;
  Decay the ε-Greedy exploration parameter ε.;
  $s_t = s_{t+1}$;
  $w = w - 1$;

3.5 Algorithme 2 : Cadre Multi-Agent

Le deuxième algorithme est en réalité une combinaison sur tous les agents du précédent algorithme. Pour simplifier la lecture : voyez les indices exposants i qui apparaissent partout. On va cette fois-ci boucler sur tous les agents où l'$agent_i$ traite à l'échelle $(tf)^i$. La nouveauté est que l'$agent_i$ reçoit les actions des $n-i+1$ agents avant lui en entrée de son Deep Q Network comme mentionné plus haut. C'est cette idée qui justifie le choix d'un framework de RL Multi-Agent. Vous le trouverez décrit dans cette section un peu plus bas.

Algorithm 2: DQN trading algorithm for multi-agent framework

Input: Experience replay memory and its size (N^i), Q^i network and its parameters, target network $\widehat{Q}^i$ and its parameters, market environment for the trading timeframe (S^i), the time window of action stabilization (tw^i), look back period (n^i), the initial value of exploration parameter (ε^i) and its decay rate, number of steps (C^i) to restart weights of the target network, discount factor (γ^i), trading timeframe (tf^i);

Output: Optimal parameters of Q^i network

for *each agent* **do**

Set tf^i = trading timeframe for the agent i;

Initialize the experience replay memory M^i to capacity N^i;

Initialize Q^i network with random weights θ^i;

Initialize target network $\widehat{Q}^i$ with weights $\theta^{-i} = \theta^i$;

Set tw^i = time window of action stabilization for tf^i;

Set $w^i = 0$;

Set n^i = look back period;

Initialize current position of the agent, a_c^i;

Initialize s_t^i to a sequence of n previous $OHLCV^{tf^i}$ and $\Phi_c^{>i}$

repeat

for *each agent* **do**

if $OHLCV^{tf^i}$ *is completed at the current time (candle is closed)* **then**

if $w = 0$ **then**

With probability ε^i, select a random action a_t^i from {1,0, − 1};

Otherwise, select $a_t^i = \text{argmax}_{a^i}\, Q\left(s_t^i, a^i; \theta^i\right)$;

$a_c^i = a_t^i$;

$w^i = tw^i$;

else

$a_t^i = a_c^i$

$\left((\text{OHLCV})_{t+1}^{tf^i}, (\text{OHLCV})_t^{tf^i}, \cdots, (\text{OHLCV})_{t-n+2}^{tf^i}, \Phi_c^{>i}\right)$;

Get the reward $r_t^i = \frac{\left(p_{t+\text{tw}^i}^{tf^i} - p_t^{tf^i}\right)}{p_t^{tf^i}} \times a_t^i$;

Store experience $e_t^i = \left(s_t^i, a_t^i, r_t^i, s_{t+1}^i\right)$ in M^i;

Randomly sample a minibatch of experiences $\left(s_j^i, a_j^i, r_j^i, s_{j+1}^i\right)$ from M^i;

Set $y_j^i = r_j^i + \gamma^i \max_a^i \widehat{Q}^i\left(s_{j+1}^i, a^i; \theta^{-i}\right)$;

Perform a gradient descent step on $\left(y_j^i - Q\left(s_i^i, a_j^i; \theta^i\right)\right)^2$ considering network parameters θ^i;

Reset $\widehat{Q}^i = Q^i$ every C^i steps;

Decay the ε-Greedy exploration parameter ε^i.;

$s_t^i = s_{t+1}^i$;

$w^i = w^i - 1$;

until *termination time*;

4 Expérimentations et Résultats

Nous avons reproduit l'Algorithme 1 en langage Python.

Nous téléchargeons des données OHLCV via l'API de Yahoo Finance sur une certaine période. On ajoute en prime les prix "adjusted close" qui correspondent à des prix de fermetures dans lesquels les dividences ont été pris en compte. Ceci nous conduit à une dimension d'entrée de 6 pour nos réseaux. La dernière date contenue dans le jeu de données est considérée comme étant terminale *ie* T = *terminal* = taille des données.

4.1 Expériences sur l'Algorithme 1

4.1.1 Essai sur des états générés aléatoirement

Nous avons commencé par vérifier que notre algorithme renvoyait tout ce que nous souhaitions en écrivant moult *print* dans le code. Mais nous avons trouvé une autre idée : générer des états aléatoirement et observer les sorties. En voici un exemple :

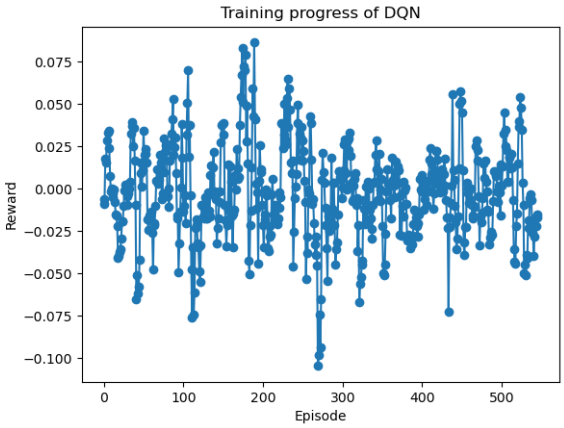

Figure 6: Récompenses obtenues au cours du temps

Les états ont été tirés selon une loi normale centrée réduite (à l'aide de torch.randn) et ont traversé nos réseaux profonds. Il en résulte une récompense centrée aux allures relativement gaussiennes. L'espérance est toujours très proche de 0 lors des différentes expériences aléatoires. Nous sommes satisfaits par ce premier essai réalisé avec un lookback $n = 1$.

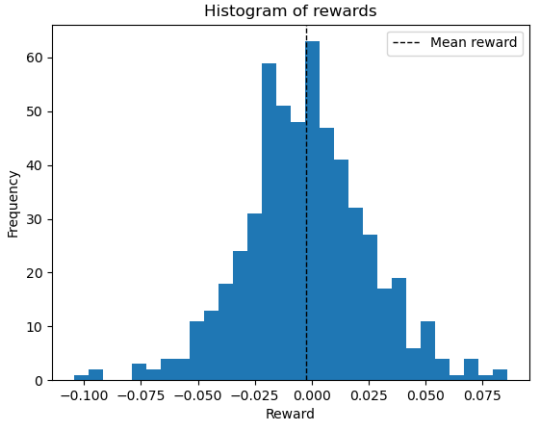

Figure 7: Histogramme des récompenses obtenues

4.1.2 Essais sur le Bitcoin avec un lookback n = 1

Nous choisissons dans cette section un actif très bruité et volatil : le Bitcoin (via la paire BTC-USD). Nous entraînons un agent seul avec un lookback de $n = 1$ autrement dit les états correspondent aux prix OHLCV au temps t et rien de plus dans le passé. La période considérée est du 2023-01-01 au 2023-11-22. On s'attend naturellement à une mauvaise performance dans cette version très simplifiée.

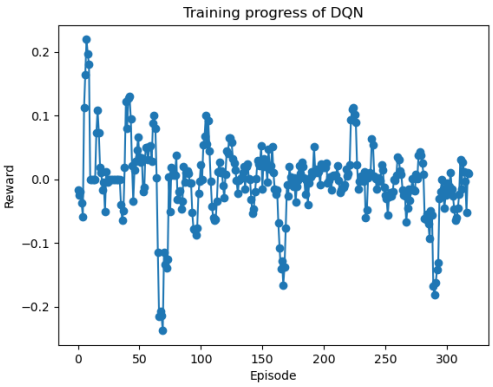

Figure 8: Première instance entraînée en 2023

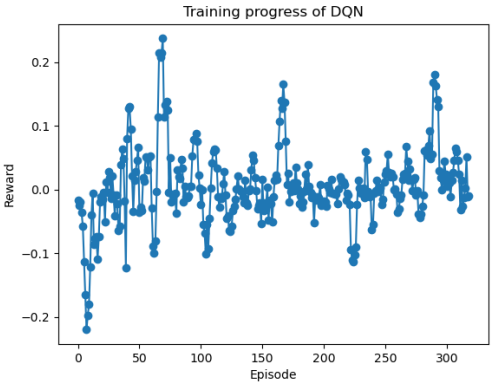

Figure 9: Deuxième instance diamétralement opposée

Comme attendu, deux instances différentes entrainées sur un seul épisode d'une courte période d'entraînement d'une année montrent des résultats fortement aléatoires. L'une présente une tendance positive l'autre plutôt négative. Le réseau ne parvient pas à saisir la logique inhérente aux mouvements du BTC-USD.

Nous poursuivons maintenant dans un cadre plus rigoureux. Nous considérons 5 époques d'entraînement. Nous utilisons presque 5 ans de données historiques pour entraîner le réseau. Nous proposons ensuite une visualisation de la stratégie proposée par l'agent sur l'année suivante (par rapport aux données d'entraînement) c'est à dire un essai *Out of Sample*.

Les paramètres considérés de façon purement arbitraire pour cet essai sont :

C	ϵ	γ	Batch size	tw	# Hidden layers	Activation functions
50	0.80	0.975	200	5	3	1: ReLu, 2: Sigmoid, 3: Tanh

Les dates étudiées sont START-DATE = '2018-06-01', END-DATE = '2023-03-22', OOS-START-DATE = '2023-03-28', OOS-END-DATE = '2024-03-01'.

Constatons tout d'abord que notre réseau semble bel et bien apprendre au cours des époques :

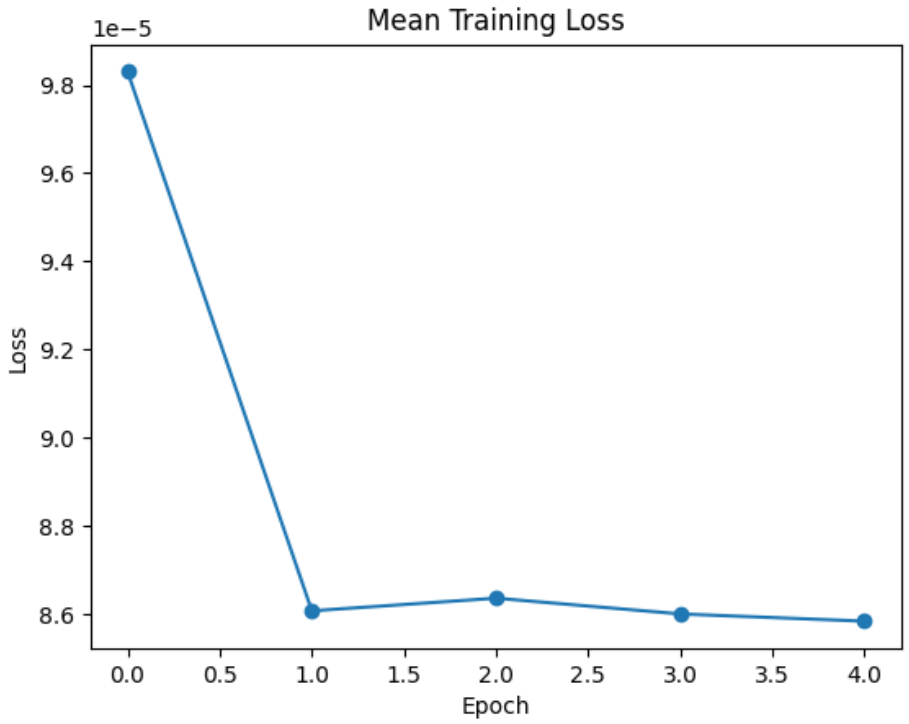

Figure 10: Loss très faible, mais décroissant

Nous notons toutefois qu'au cours de nos expérimentations nous obtenions parfois des instabilités au niveau du loss. L'ordre de grandeur du loss était entre 10 et 10^3. Nous considérons que ces anomalies étaient causées par un mauvais choix - arbitraire et non réfléchi pour l'exemple - des hyperparamètres. Ceci nous a conduit à considérer une approche de recherche dans une grille de paramètres plus vaste évoquée plus bas.

Voici un exemple de réalisation pour ce jeu de paramètres donnant un aperçu de ce que main.py peut donner en sortie pour un agent traitant à l'échelle journalière :

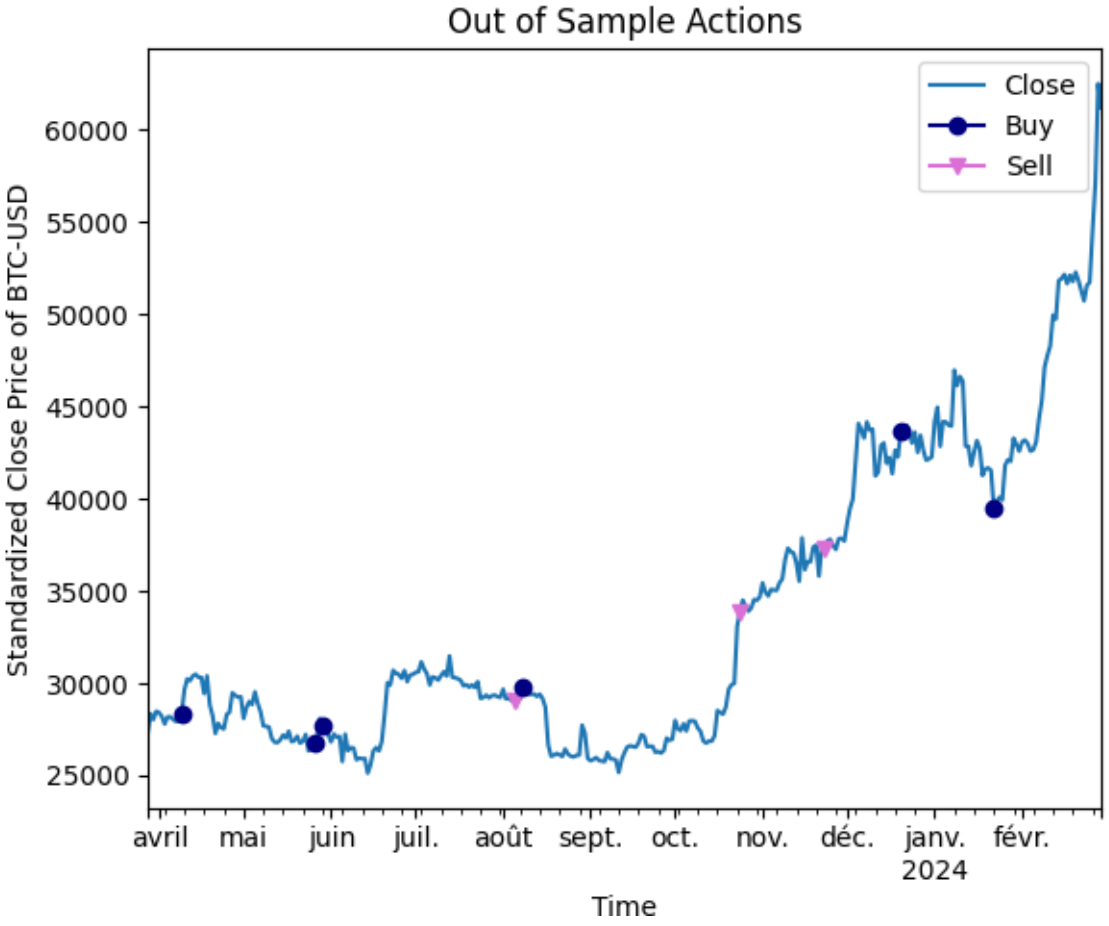

Figure 11: Exemple de stratégie proposée par l'agent

Evidemment, la stratégie n'est pas parfaite. Mais pour des paramètres complètement arbitraires et sur des données OOS ces humbles ordres d'achats et de vente nous sont tout à fait acceptables.

4.1.3 Essai après une Grid Search

Nous décidons d'affiner notre algorithme en intégrant une Grid Search. On ne considère plus un réel fixé par paramètre mais des listes discrètes de paramètres. Nous nous intéressons en particulier aux paramètres suivants :

1. ϵ, paramètre dictant l'exploration dans l'ϵ-greedy.
2. γ le facteur d'actualisation de Bellmab.
3. C, la constante qui dicte les mises à jour de poids du Target Network
4. La taille des batch.
5. La learning rate du réseau Q.
6. Le nombre de neurones par hidden layer.
7. Les fonctions d'activations de chaque hidden layer.

Nous entraînons l'algorithme sur un nombre donné d'époque puis nous effectuons un test sur des données out of sample après une grid search ; et ce sur le ticker Air Liquide (AI.PA) aux dates START-DATE = '2021-01-01' et END-DATE = '2023-03-22'. Le test OOS s'effectue ensuite jusq'à OOS-END-DATE = '2023-11-01'.

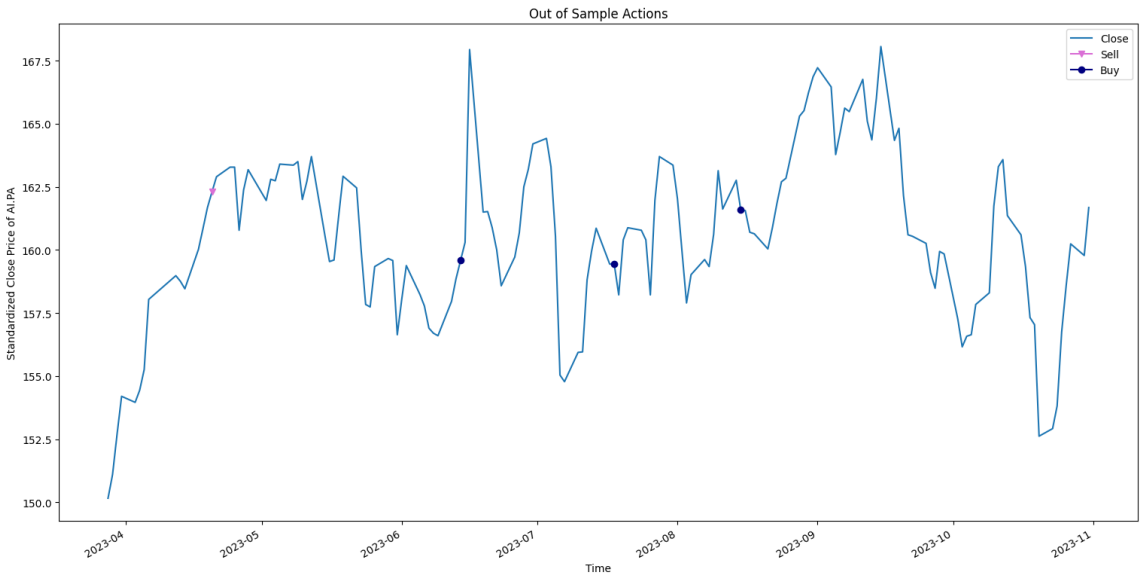

Figure 12: Les points d'achat et de vente sont judicieux.

On obtient ici un excellent résultat avec des points d'entrée sur l'action judicieusement choisis. En toute rigueur nous devrions conduire des tests statistiques et fourni des p-valeurs afin d'appuyer nos résultats pour chaque action. Ceci n'est pas l'objectif du présent rapport.

5 Conclusion

L'ensemble d'actions considéré est simpliste et devrait être amélioré. La plupart des gains sont à long terme fortement corrélés au volume des trades pris ainsi qu'au choix des actifs [Gar95] or notre algorithme ne le prend pas en considération. Nous souhaiterions intégrer un "bet-sizing" en complément afin d'affiner la taille de l'action set. Nous pourrions commencer par l'ajout d'actions telles que "acheter pour 20% de notre portefeuille *ie* ajouter 0.2 à $[-1{,}0{,}1]$ l'ensemble d'action précédent.

Une autre piste d'amélioration intéressante concerne le choix de la politique. Nous pourrions complexifier l'algorithme en passant d'un ϵ-greedy à une politique plus adaptée.

Les hypothèses concernant l'impact des agents sur le marché sont fausses dans la pratique. Il serait bon par exemple d'abaisser légèrement chaque reward d'une certaine constante (dans le cas de fees constantes sur un marché donné) pour prendre en compte les coûts de transaction.

Le code fourni dans main.py permet de modifier le nombre d'épisodes d'entraînement à sa guise ainsi que bon nombres d'autres paramètres. Cependant le lookback est fixé à une valeur de 1 afin de simplifier les calculs, au détriment de la précision. Nous observons bel et bien un loss décroissant dans la majorité des cas. La Grid Search contribue à une nette amélioration des performances comme attendu. L'algorithme développé est iso.

Merci pour votre lecture.

References

[Ali22] Ali Shavandi, Majid Khedmati. "A multi-agent deep reinforcement learning framework for algorithmic trading in financial markets". **in***Expert Systems With Applications*: (2022). URL: https://www.sciencedirect.com/science/article/pii/S0957417422013082?ref=pdf_download&fr=RR-2&rr=84d298ea7b550181.

[Ric99] Richard Sutton, Andrew Barto. *Reinforcement Learning: An Introduction*. Canada, 1999. URL: https://www.andrew.cmu.edu/course/10-703/textbook/BartoSutton.pdf.

[Chr92] Christopher J. C. H. Watkins, Peter Dayan. *Q-learning*. Semrock. 1992. URL: https://link.springer.com/article/10.1007/BF00992698.

[Bel66] Richard Bellman. *Dynamic Programming*. 1966. URL: https://www.science.org/doi/10.1126/science.153.3731.34.

[Hug23] Hugging Face Courses. *Deep RL Course*. Hugging Face. 2023. URL: https://huggingface.co/learn/deep-rl-course/unit3/deep-q-algorithm.

[Gar95] Gary P Brinson, L Randolph Hood, "Determinants of portfolio performance". **in***Financial Analysts Journal*: (1995).

Déclaration

Nous certifions par la présente que le présent rapport a été rédigé de manière autonome et que toutes les sources et références nécessaires sont mentionnées.

Eric PATARIN	March 5, 2024
Etudiant 1	Date

Lucas SELINI	March 5, 2024
Etudiant 2	Date