RL Trading Agent with Gym-AnyTrading

Jérémie Dentan, Xavier Durand, Meryem Jaaidan, Louis Proffit, Paul Théron

***Abstract*—The advent of crypto-currencies since 2017 is encouraging more and more people to invest in them. Notably, the price of Bitcoin peaked at $69,000 on October 11, 2021, it was multiplied by 10 in 18 months.**
As a result of this craze, crypto-trading algorithms have emerged. We will explore one category of these algorithms based on reinforcement learning. We have implemented an environment based on the bitcoin price in dollars to develop several reinforcement learning algorithms : deep Q-learning with a Deepsense network, and Policy Gradient (with REINFORCE and A2C algorithms). We also implemented a more classical approach such as AdaBoost classifier to have comparison results to show the interest of the reinforcement learning approach.
Finally, we will discuss the performance of our networks, and their possible use in the real world.

This paper is provided with a git repository accessible **here**.

I. Introduction

Forex trading (or currency trading) is often considered one of the most attractive areas of finance, combining quantitative skills, finance knowledge and programming skills. As a result, the increasing complexity of models and computing power has led to the development of even more powerful trading agents capable of tackling extremely complex problems. In recent years, RL models have been more and more applied to forex trading (for example [3] [4] [5]). Indeed, models involving RL have several advantages over conventional tools:

- They can be trained directly on financial data, without the need to add other indicators.
- Traditional methods, especially supervised learning, can perform very well on past data, but are less efficient on future data (e.g. in the case of a crisis or macroeconomic changes). Thus, RL models, which are self-adaptive by construction, adapt faster to such changes and are more resilient.
- It is very easy to modify the agents' reward to add other metrics to the system as needed.

For this project, we applied state-of-the-art RL models for forex trading, and in particular **for BTC/USD trading** (bitcoin/US dollar trading). The key features of our approach are the following :

- Our model is based on policy-predictions, and not on price-predictions. In fact, the RL agent only trains to be able to take efficient decisions based on the financial data.
- Our dataset is composed of 7 features of the BTC/USD market for every minutes since 2017, which are : open, high, low, close, Volume BTC, Volume USD and evolution of open price. At each step, the agent sees the 7 features over the past 180 minutes.
- Our action space contains 3 acts : SELL, STAY and BUY. Thus, our agent only trades a fixed amout of currency (1 BTC) at each step (1 minute). Moreover, it has an infinite budget, and it can short BTC to own negative quantities without constraints.
- At the end of the game, the agent exits the market, which means selling its BTC or buying some if it had shorted BTC (and thus owns negative quantities). Our goal is to maximize the amount of USD earned during the whole game.
- Since the feature space is continuous, the state space of our RL models is infinite. So we only used deep-learning network to predict the policy based on the previous features : **we implemented both Q-learning approaches** (double deep Q-Learning with the DeepSense network) **and Policy Gradient methods** (REINFORCE and Advantage Actor Critic).

For our implementation, we used the Gym-AnyTrading environment. This environment is designed for forex trading and implements many functionalities we needed. However, we adapted it to our needs, and in particular to our datasets and features, by creating a new environment inspired by Qym-AnyTrading : `CryptoEnv`, in the file bitcoin_env.py.

Moreover, we based our model on several research papers and a significant part of our work was to adapt the architectures to our concrete needs :

- We implemented the deep Q-Learning framework by our own. For the DeepSense architecture, we used the article of *Deep Reinforcement Learning for Trading* [1][1], which provided an implementation in tensorflow v1. However, we simplified the network and translated it in Pytorch since tensorflow v1 is depreciated.
- For Policy Gradient approaches, we used the code of *Morvan Zhou* [10], which provides an implementation in tensoflow. We also translated it in Pytorch and adapted the network to our needs and our datasets.
- We use sklearn library for all classic classifier algorithm (LogisticRegressor, AdaBoostClassifier, etc...)

II. Background and Related Work

a) Financial background: Forex (Foreign Exchange) is the market on which currencies are traded. This can state currencies like USD or EUR, or cryptocurrencies like BTC or ETH. Regarding the volume of trading, forex is by far the most important market in the world (mean daily volume of 6.6 Trillion in 2019). That's why forex trading is a very important field of research in quantitative finance.

The idea of using computers to help traders over perform the market has emerged really early in the history of AI. For example, Chan [3] in 1995 used a neural network to generate trading signal before regular technical indicators do.

This gives one the opportunity to enter, and exit trades before the crowd. Those algorithms have quickly been applied to the forex market. For example, in 1997, Neely [4] used a genetic approach to compute technical trading rules, with a significant improvement in performance for many currencies. Today, countless researchers have applied RL techniques, for example Deepsense [2] and Sutta [5].

b) Quick reminder on RL, autonomous agent and Q-Learning: In this paper we will use the standard notions and notations in Reinforcement learning, which are for example reminded in the course *INF581* [7]. An agent evolves in an environment represented by a state space $\mathcal{S}$, which in our case is infinite. At each (discrete) step, the agent can take an action within the action space $\mathcal{A}$. Those actions have an impact on the state of the environment, and provides the agent a reward r. The function mapping a state to an action is called a policy π (or π_θ when it depends on a parameter). At time t, the agent estimates a value V of each step : $V(S_t)$. This value corresponds to the expected value of the reward in the future, with a depreciation term γ which represents the preference of the agent for the present.

$$G_t := \sum_{t'=t+1}^{+\infty} \gamma^{t'-t-1} r_{t'}$$
$$V(S_t) = \mathbb{E}\left[G_t|\pi\right] := \mathbb{E}_\pi\left[G_t\right]$$

From this value function is derived the q function, which is computed with a parameter $w \in \mathbb{R}^p$. It takes as argument a state S_t and an action A_t, and returns the estimated value of the next state if the agent takes action A_t :

$$[q_w(S_t)]_{A_t} = \mathbb{E}_\pi\left[V(S_{t+1})|A_t, S_t\right]$$

For a policy with a parameter θ, we introduce an objective function for the agent, which is the cumulative gain of the game :

$$J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T} r_t\right] := \mathbb{E}_\theta\left[\sum_{t=0}^{T} r_t\right]$$

Given those definitions, we implemented two different approaches for our agent : deep Q learning and policy gradient. For Q-Learning, the idea is to learn the function q by improving the parameter w and to derive a policy from this learned q function. On the contrary, the policy gradient approach directly optimizes θ to improve the objective function $J(\theta)$.

III. The Environment and Agent

a) Our environment: As described in Section I, we used an environment inpired by Gym-AnyTrading : bitcoin_env.py. We have implemented several adaptations to this environment that doesn't inherit from gym-anytrading. Having a well-working environment was an important part of our project. Its main features are the following :

- **State space :** a continuous state matrix $\mathcal{S} = \mathcal{M}_{180,7}(\mathbb{R})$: seven features over the past 180 periods. The periods are intervals of 1 minute and the features are *open* (open price), *high* (highest price in the period), *low* (lowest price in the period), *close* (close price), *Volume BTC* (volume of BTC sold during the period), *Volume USD* (volume of USD sold during the period), *diff* (the difference between the open price and the previous one, with a padding of 0). For the moment, the quantity of BTC owned by the agent is not part of the feature because we considered that it is not really important due to the infinite budget. However, in the next version, we will integrate this as a feature, since the current quantity of BTC owned is important to forecast the future financial profit of the agent.
- **Action space :** We wanted to have a fixed action space for our agent, because most RL algorithms are designed for such action spaces. That is why we chose an agent with an infinite USD budget, and who can short BTC to own negatives quantities of BTC. For more simplicity, the amount traded at each step t is always the same, $h_t = 1$ BTC. This explains why our action space is size 3 : $\mathcal{A} = \{\text{BUY}, \text{SELL}, \text{STAY}\}$.
- **Final states :** Our agents evolve during $T = 10$ steps, and the 10th is always a final step. At this moment, the agent sells all the BTC (or buy them if they shorted BTC). We decided that T will be equal to 10 steps because it is sufficient at the beginning of the training phase (to learn how to behave for short-term profit). However, in the next version, we will increase gradually this number to obtain an agent capable of improving long-term profit.
- **Reward :** At step $t < T$, the reward is given by the variation of the valuation of the portfolio plus the fees :

  $$r_t = c_t \times (p_t - p_{t-1}) + \epsilon_t^{fees} \times f \times h_t$$

  where : c_t is the total quantity of BTC owned, p_t is the price of BTC at time t, ϵ_t^{fees} is 0 if the agent STAY, else 1, and f is the fee rate.
  At step T, the reward is :

  $$r_T = USD_T - USD_0$$

  where USD_t is the amount of USD earned / due at time t. Thus, r_T represents the total gain of the agent during the game. It has been more efficient for the training to implement this instead of computing r_T similarly to r_t for $t < T$.

IV. First approach : deep Q-Learning

a) (Double) deep Q-Learning: With Q-Learning in general (whether it is deep or not), the principle is to learn the q function, which means, in our deep case, to learn the parameter w. Then, when the function q is known or at least correctly approximated, the agent's policy can for example be given by the ε-greedy algorithm : with probability ε the agent takes a random action, and otherwise takes the best action according to q.

Thus, we need to learn w to determine the agent's policy. As explained in the introduction, since the state space is continuous, we only implemented models with a neural network to take actions given the input features. For our implementation of the deep Q-Learning, the weights w of the network are

updated with a learning rate α with the deep Q-Learning TD formula:

$$w_{t+1} = w_t + ...$$
$$\alpha \left[\underbrace{(r_t + \gamma \max_{A \in \mathcal{A}} [q_{w_t}(S_{t+1})]_A)}_{\text{target}} - \underbrace{[q_{w_t}(S_t)]_{A_t}}_{\text{prediction}} \right] \nabla_w [q_{w_t}(S_t)]_{A_t}$$

Thus, the network is used twice : first to predict the best action to take, and then to evaluate this prediction. This might sound strange, but this bootstrap principle has achieved a good performance. Moreover, as described by the seventh lecture of the course [6] and the article *Deep Reinforcement Learning with Double Q-learning* [8], we used a double DQN, which means that the two calls of our network are in fact two different networks : a slow learning target network, and a fast learning prediction network of the same structure. This enables significantly better learning performances.

b) Our network architecture: (Double) deep Q-Learning is a general principle that applies to many RL situations. However, the choice of the network used for the evaluation / prediction has a significant impact on the agent's performances. The main characteristic of our environment is that our features are **time series of correlated variables**. Thus, simple MLP or CNN on the feature matrix are not really efficient because they lack the temporal information of the input. For example with a regular MLP fed by the concatenated input $x \in \mathbb{R}^{1260}$, it is really hard to learn that there is in fact 7 channels of size 180.

Numerous neural networks tackle time series of correlated variables. LTSM and GRU are probably the most well-known classes of such networks, and Rafal Jozefowicz *et al.* even declared in their review of existing RNN architectures in 2015 [9] : *"We have evaluated a variety of recurrent neural network architectures in order to find an architecture that reliably outperforms the LSTM. Though there were architectures that outperformed the LSTM on some problems, we were unable to find an architecture that consistently beat the LSTM and the GRU in all experimental conditions."*

This is why, for our main neural network architecture, we used an architecture based on GRU. This architecture is inspired by the Deepsense one [2], which is a deep neural network originally designed for mobile sensing data processing. Indeed, mobile needs the analysis of many correlated time series, such as accelerometers, gyroscopes, and magnetometers, so we supposed it could work well on financial data with the same characteristics. The original architecture presented by the article *Deepsense* [2] is presented by Fig. 1. Moreover, this network has already been used for deep Q-learning in finance by *Kartikay Garg* [12].

Apart from translating the original code into Pytorch and adapting it to our environment and agent, we have simplified the architecture. Our final architecture can be found in the `DQNSolver` class in the DQN_deepsense.py file :

- Two convolutional layers with ReLu activation and dropout. Those convolutional layers are operated separately on each channel (the 7 features). However, the convolutional layers are operated on a matrix : the time series of length 180 is divided by strides of length 9, forming a matrix of size 9×20.
- A unique GRU unit fed by the (linear) result of the convolutions.
- A 3-layer MLP fed by the output of the GRU.

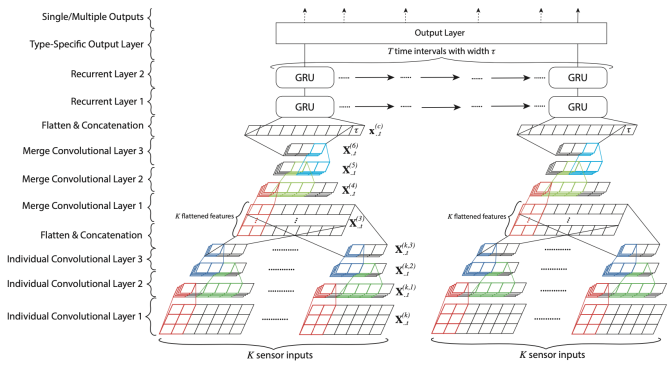

Fig. 1. The original deepsense architecture presented by [2]

V. Second approach : policy gradient

a) Policy gradient principle: As explained before, policy gradient directly optimizes a policy's parameter θ to maximize the objective function $J(\theta)$. Thus, the principle of the policy gradient is to update θ with the following gradient (see [1] or [6] for more details) :

$$\nabla_\theta(J(\theta)) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t|S_t) G_t$$

b) First implementation : the REINFORCE algorithm: We firstly implemented the REINFORCE. We based our implementation on the code of *Morvan Zhou* [10], however we translated the code into Pytorch, and we adapted it to our environment.

The most important part here is the architecture of the network for π_θ. In our case, we used a 4-layer MLP. Thus, the fact that we are dealing with correlated time series is not fully exploited, and this could probably be improved for future versions of this code.

c) Second implementation : Advantage Actor Critic algorithm: One of the main problem of REINFORCE is that is suffers from high variance and noisy gradient. A first approach to reduce the high variance is to introduce a baseline. The baseline $b(S_t)$ is subtracted from the cumulative gain G_t. Indeed, as explained by *Chris Yoon* [11], making cumulative gain smaller enables to really concentrate on the difference between different gains and thus to reduce the variance. Thus, we first consider that :

$$\nabla_\theta(J(\theta)) = \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t|S_t)[q_w(S_t)]_{At}$$

Then we subtract the baseline to $[q_w(S_t)]_{At}$. To do this, we replace the q function by the *advantage* of a given action from a given state :

$$A(S_t, A_t) = [q_w(S_t)]_{At} - V(S_t) = r_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

Finally, with the Advantage Actor Critic, we used this approximation of the gradient of the objective function :

$$\nabla_\theta(J(\theta)) \simeq \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(A_t|S_t)\,(r_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

Then, we trained our agent with the same policy network than for REINFORCE.

VI. Learning strategies

a) Our pipeline: It quickly appeared that teaching an agent to trade BTC/USD with our algorithms is not simple at all. Indeed, probably due to the high volatility of the BTC/USD exchange rate, our agent performed very poor results on our simulations. That is why we decided to first train it on much simpler datasets, where the exchange rate is approximately a sinus curve. The idea of this approach is that the agent should learn how to behave when the rate increases or decreases. The idea is to vary the parameters of the sinus curve (in particular period, offset and noise) so that the agent learns how to make profit with various situations of increasing or decreasing rates. We also trained our agent on more complicated curves, for example on a sum of two sinuses (the second one with no offset and a period and an amplitude divided by two).

In addition to this learning set on sinus curves, we adapted our learning rate during our pipeline. We wanted our agent to first learn on sinus curves with a great learning rate and then to adapt to real curves with much more epochs but a smaller learning rate. Our pipeline parameters are as follows:

#	curve	period	offset	ampl.	noise	lr	epochs
1	sin	0.2	900	50	0	1	10.000
2	sin	0.4	1000	150	0	1	10.000
3	sin	0.2	900	50	5%	1	10.000
4	sin	0.2	900	50	0	0.1	10.000
5	sum sin	0.2	900	50	8%	0.1	10.000
6	real	-	-	-	-	0.1	50.000
7	sin	0.2	900	50	5%	0.3	10.000
8	real	1	900	-	-	0.01	150.000

TABLE I
Our learning pipeline, for an agent first trained on sinus curves and then on real data.

In addition to this pipeline, we tried an other trick to improve the performance of our agent : training it on the same episode (which corresponds to the same input of size 7×180) 100 times to gradually teach our agent to make profit on specific inputs.

b) Performance comparison: To evaluate the performance of our agent, we run it on a test set which corresponds to the last months of our dataset, and on which the agent has never trained. Then, we compare our objective function with our trained policy $J(\theta)$ with two deterministic policies :

- STAY-ONLY : a policy whose agent always choose STAY.
- OPTIMAL : a deterministic algorithm which computes an approximation of the optimal policy (computing the optimal one in our case has exponential complexity). This algorithm can be found on the git repository with the `env_scorer.test_optimal` function.

VII. Results and Discussion

a) Our results: Unfortunately, the performance of our algorithms was not good, despite our efforts to adapt our learning strategies and pipeline. One major difficulty for the learning phase is the time it takes : about 10 batches of size 256 per second on a Nvidia Quadro RTX 4000. Nevertheless, despite its poor performances on real data, our agent was able to make significant profit (close to the one of OPTIMAL strategy) on sinus curves only. For example, figure 3 shows that the performance of our agent is increasing and that it finally makes a profit.

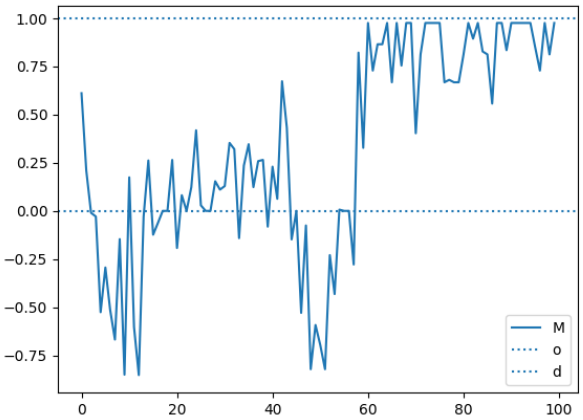

Fig. 2. Relative performance of the deep Q-Learning algorithm with the Deepsense network, compared to OPTIMAL (1.00) and STAY-ONLY (0.00), on the first sinus curve of the pipeline presented in table I

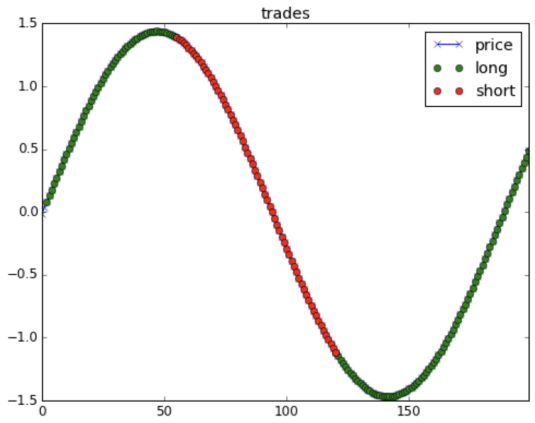

Fig. 3. Plot of the decisions made by the deep Q learning agent after training, on the first sinus curve of the pipeline presented in table I

Moreover, we trained our agent with our other approaches (Policy gradient with REINFORCE and A2C). Unfortunately, our performance was even worse than with deep Q-Learning.

b) Other approaches we tried: We implemented a more classical, the AdaBoost classifier from the Sklearn package. The algorithm has three objective functions that are not optimal but are a good approach. They take all the training data as an input, so they just teach the classifier the way of thinking. The first function sorts prices, sells the top half and buys the other. The second function is based on the fact that everything is sold at the previous step, so it sells when the price is above the one before, and buys otherwise. The third one is more local, it buys if the following price is above the current one and sells otherwise.
The classifier got positive results on the two last objective functions, staying at 0.89 of relative performance. The first strategy dropped to -0.89, so it was not the best choice. This result is quite better at the beginning than the Q-learning algorithm, but tends to be less good at the end of the training.

c) What could we try to get better results ?: As described above, we tried many techniques to obtain acceptable results with our agent : using sinus curves, increasing training time, changing learning rate, changing batch size, trying to overfit a given input by training our agent many times on it, fine-tuning our networks... in vain. However, here are few ideas of improvement that we did not have the time to implement but that could improve our results :

- Deep Q-Learning networks are known to be hard to train, they need a really long learning phase. A significant improvement our our epoch number might be helpful (as in figure 4, where the agent only starts to get better results at the end our our 10.000 epochs). However, due to the learning pace (about 10 batches of size 256 per second), this would require a federated-learning approach to keep a reasonable computation time.
- Even though *Kartikay Garg* [12] obtained correct results with the Deepsense network, we probably should have changed our network to have a simpler one, probably easier to fine tune and to train.
- Some other RL techniques are known to perform great results, and are probably easier to implement and train, such as genetic algorithms.

VIII. Conclusion

Unfortunately, our work was not successful because our agent did not have the expected results in terms of financial performance. The ideas presented in section VII-0c might improve this, even though we did not have time to try them.

Nevertheless, this project enabled us to get more familiar with major RL techniques such as double deep Q-Learning or policy gradient approaches (e.g. REINFORCE, A2C), whether it is for the academic side or for the implementation, which represented an important part of our work. Moreover, it confronted us to the difficulties of the training phase such as designing a correct pipeline, choosing an appropriate batch size and learning rate, correctly evaluate our performance... Thus, we learned a lot from this project.

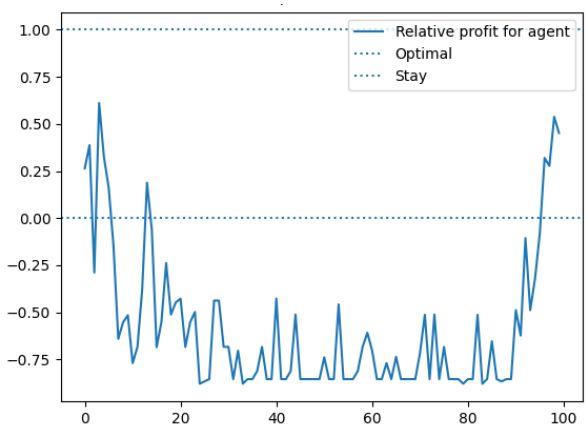

Fig. 4. Relative performance of our agent. This is an agent on step 4 of our pipeline, after having been trained on steps 1, 2 and 3. We see that a longer learing phase would probably have been needed to adapt to a sum of sinus curves.

References

[1] Zihao Zhang, Stefan Zohren, Stephen Roberts. *Deep Reinforcement Learning for Trading*. 2019.

[2] Shuochao Yao, Shaohan Hu, Yiran Zhao, Aston Zhang, Tarek Abdelzaher. *DeepSense: A Unified Deep Learning Framework for Time-Series Mobile Sensing Data Processing.* Published in WWW2017.

[3] Chan, K. C., Teong, F. K.. *Enhancing technical analysis in the Forex market using neural networks.* Paper presented at the Neural Networks, 1995. Proceedings., IEEE International Conference on.

[4] Chris Neely, Paul Weller, Rob Dittmar. *Is Technical Analysis in the Foreign Exchange Market Profitable? A Genetic Programming Approach.* Journal of Financial and Quantitative Analysis, October 1996

[5] Sutta Sornmayura. *Robust Financial Trading System With Deep Q Network (DQN).* International College, National Institute of Development Administration 2017

[6] In Lecture VII - Reinforcement Learning III. *INF581 Advanced Machine Learning and Autonomous Agents*, Jesse Read, 2022.

[7] *INF581 Advanced Machine Learning and Autonomous Agents*, Jesse Read, 2022.

[8] Hado van Hasselt, Arthur Guez, David Silver. *Deep Reinforcement Learning with Double Q-learning*, 2015.

[9] Rafal Jozefowicz, Wojciech Zaremba, Ilya Sutskever. *An Empirical Exploration of Recurrent Network Architectures*. Journal of Machine Learning Research, 2015.

[10] MorvanZhou git repository : Reinforcement Learning With Tensorflow : https://github.com/MorvanZhou/Reinforcement-learning-with-tensorflow/tree/master/contents.

[11] Chris Yoon. Understanding Actor Critic Methods and A2C : https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f.

[12] Kartikay Garg (samre12) git repository : deep-trading-agent : https://github.com/samre12/deep-trading-agent.