Reinforcement Learning Project

Achraf Maalej Skander Kazdaghli Jed Houas

April 21, 2020

1 The Environment : the financial Trading System

The FTS that we chose is inspired by F. Bertoluzzo al [3].

We start by defining the state as the last five percentage returns of the traded asset (corresponding to the last five trading days):

$$s_t = (r_{t-4}, r_{t-3}, r_{t-2}, r_{t-1}, r_t)$$

Where r_t is the log return of the asset $r_t = log(P_t) - log(P_{t-1})$

Concerning the actions, we use the following :

$$a_t = \begin{cases} -1 & \text{sell signal} \\ 0 & \text{stay out of the market signal} \\ 1 & \text{buy signal} \end{cases}$$

As for the immediate reward: $r_t = a_{t-1} * (log(P_t) - log(P_{t-1})))$

We define the total gains of the trading strategy as follows :

$G = \sum_t r_t$

2 The Data

We used the Kaggle "Huge Stock Market Dataset" [1]. It contains the full historical daily price and volume data for all US-based stocks and ETFs trading on the NYSE, NASDAQ, and NYSE MKT. For each RL method, we choose a stock, we subdivide it into two parts : train and test sets. We then train our trading agent on the first part and then test its performance on the second part.

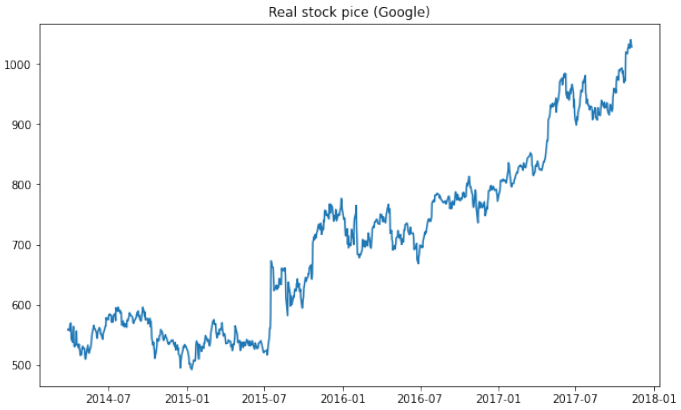

Figure 1: Example of a US-based stock

3 Trading using Kernel-Based Reinforcement Learning

3.1 The KbRL algorithm

The KbRL protocol that we used in this project is taken from D. Ormoneit al[2]

Let $S_a = \{(x_s, y_s^a)|s = 1..m_a\}$ be a collection of historical transitions where action a is taken. The initial state is x_s and the next step where action a is taken is y_s^a.

Let ϕ be a "mother kernel" function and $k_{S_a,b}$ be the weighting function.

$$k_{S_a,b}(x_s, x) = \phi(\frac{\|x_s - x\|}{b}) / \sum_{(x_u, y_u^a) \in S^a} \phi(\frac{\|x_u - x\|}{b})$$

Where b denotes the bandwidth.

Given the optimal state-value function $Q^*(s_t, a)$ and the optimal value function $V^*(s_t)$, we define the operator Γ_a :

$$Q^*(s_t, a) = \Gamma_a V^*(s_t) = E[r(x, X_{t+1}, a) + \alpha J_{t+1}^* | X_t = x, a_t = a]$$

We approximate the unknown operator Γ_a using a random operator $\hat{\Gamma}_a$ based on the history of transitions and outcomes :

$$\hat{Q}(x, a) = \hat{\Gamma}_a J(x) = \sum_{(x_s, y_s^a)} k_{S_a,b}(x_s, x)[r(x_s, y_s^a, a) + \alpha J(y_s^a)]$$

This weighting method give equal values to equidistant points. The above equations measures the value of applying action a in state x by looking at historical cases where the action is taken at a similar state and by averaging over the immediate rewards and value estimates. We can control the smoothing with the "bandwidth" parameter which needs to be chosen carefully. As for the mother kernel function, we chose a Gaussian kernel for our application.

Next, we define the operator $\mathcal{T}$ such that $J(x_t) = max_a Q(x_t, a) = \mathcal{T}Q(x_t, a)$

We thus have : $\hat{J} = \mathcal{T}\hat{\Gamma}_a \hat{J}$

Given the above equation, we arrive at the compact form for the value iteration update rule :

$$\tilde{J}' := \mathcal{T}(\mathcal{O}[\tilde{R} + \alpha \tilde{J}])$$

Where $\tilde{R}$ is an mxM matrix (M is the number of actions and m is the number of transitions) with entry $r(x_s, y_s^a, a)$ at (s, a)

$\mathcal{O}$ is an mxMxm tensor with entries $k(x_s, y_{s'}^a, a)$ at (s', a, s)

$\mathcal{T}$ is an operator that takes an mxMxM tensor and maximizes over its second dimension.

The value functions $\tilde{J}'$ and $\tilde{J}$ are matrices of dimensionality mxM.

After approximating the value function, we use $\hat{Q}(x', a) = \hat{\Gamma}_a \hat{J}$ to derive the action-state value function for new states x' and we pick the optimal actions to take.

3.2 Experiments and results

We the google stock price time series in our experiment. We divide it into a training set and a testing set. We estimate the value function J by with the above algorithm using the first training part of our time series data. We then estimate an optimal trading strategy for the test part of our time series.

Figure 2: Results of the KbRL based on a real stock (test set)

In the first panel is the price of the asset. The second panel shows the actions taken by the FTS. In the third panel are the rewards r_t and the fourth panel shows the cumulative gains by our FTS.

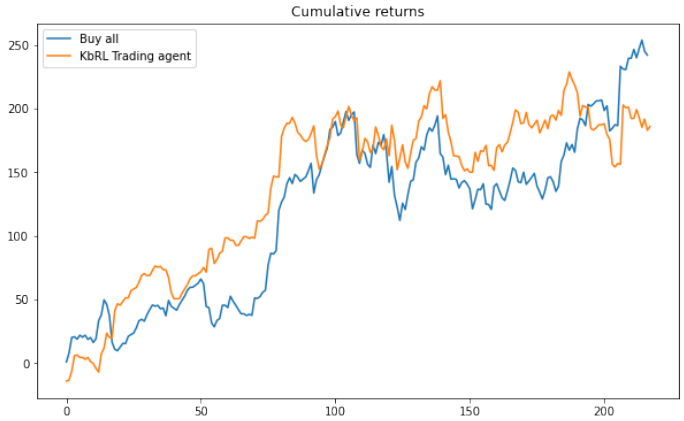

Figure 3: The cumulative gain with our RL trading agent compared to a buy all strategy for the google stock(we obtain a final return = 185.8)

Finally, we try the same experiment for the amazon stock price and we obtain the following cumulative returns plot:

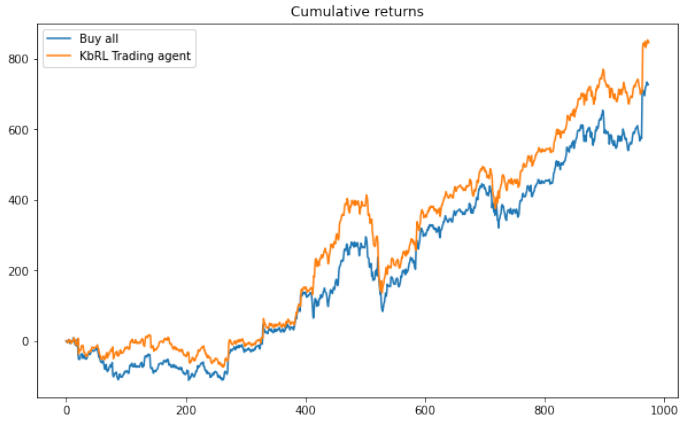

Figure 4: The cumulative gain with our RL trading agent compared to a buy all strategy for the amazon stock (we obtain a final return = 846)

4 Trading using double Q-learning

4.1 The double deep Q-learning algorithm

4.1.1 Remind on the Q-learning algorithm

The development of **Q-learning** (Watkins Dayan, 1992) is a big breakout in the early days of Reinforcement Learning.
The Q learning model-free algorithm using TD control is the following :
S is the space of states, A is the space of actions

1. At time step t, we start from state S_t and pick action according to Q values, $A_t = argmax_{a \in A} Q(S_t, a)$; ϵ-greedy is commonly applied.
2. With action A_t, we observe reward $R_t + 1$ and get into the next state $S_t + 1$. Update the action-value function: $Q(S_t, A_t) \leftarrow (S_t, At) + \alpha(R_t + 1 + \gamma\ max_a Q(S_t + 1, a) Q(S_t, A_t))$
3. t = t+1 and repeat from step 1.

4.1.2 Deep Q learning

Theoretically, we can memorize $Q*(.)$ for all state-action pairs in Q-learning, like in a gigantic table. However, it quickly becomes computationally infeasible when the state and action space are large. Thus we use functions like **neural networks** to approximate Q values and this is called function approximation. For example, if we use a function with parameter θ to calculate Q values, we can label Q value function as Q(s,a;θ).

A mechanism introduced to improve the training of the DQN is called **experience replay** : All the episode steps $e_t = (S_t, A_t, R_t, S_t + 1)$ are stored in one replay memory $D_t = e_1, \ldots, e_t$. D_t has experience tuples over many episodes. During Q-learning updates, samples are drawn at random from the replay memory and thus one sample could be used multiple times. Experience replay improves data efficiency, removes correlations in the observation sequences, and smooths over changes in the data distribution.

The function loss to minimize during the Deep Q-learning algorithm is the TD target :

$$L(\theta) = \mathbb{E}_{(s,a,r,s\prime) \sim U(D)}(r + \gamma\ max_{a\prime} Q(s\prime, a\prime, \theta) - Q(s, a, \theta))$$

where U(D) is a uniform distribution over the replay memory D.

4.1.3 Double deep Q learning

Double DQNs, or double Learning, was introduced by Hado van Hasselt in his paper [4]. This method handles the problem of the overestimation of Q-values.

At the beginning of the training we don't have enough information about the best action to take. Therefore, taking the maximum q value (which is noisy) as the best action to take can lead to false positives. If non-optimal actions are regularly given a higher Q value than the optimal best action, the learning will be complicated.

The solution is: when we compute the Q target, we use two networks to decouple the action selection from the target Q value generation. We:

- Use the DQN network to select what is the best action to take for the next state (the action with the highest Q value).
- Use our target network to calculate the target Q value of taking that action at the next state.

we note $Q^A(s, a; \theta)$ the DQN network and $Q^B(s, a; \theta^-)$ the frozen target DQN.
The loss to minimize under the DDQN algorithm is :

$$L(\theta) = \mathbb{E}_{(s,a,r,s\prime)\sim U(D)}(r + \gamma Q^B(s\prime, argmax_{a\prime} Q^A(s\prime, a\prime; \theta); \theta^-) - Q^A(s, a; \theta))$$

Double DQN help for reducing the overestimation of q values and, as a consequence, helps for faster training and have more stable learning.

We did a small research on the dueling double DQN, in which the change is the decomposition of the Q value into two parts : a state value V(s) and advantage value of taking an action A(s,a). We did not find this method useful for our case because the dueling method is particularly useful for states where their actions do not affect the environment in a relevant way. However, in our case each action has a its own effect no matter is the state.

4.2 Experiments and results

Regarding the state size (the length of the window over the past returns) we tested a range of values and we found that we get good results from taking a value over 10-15. So we set this hyperparameter to 20 for the training.

Were will train and test the algorithm on the Google and amazon stock and the period (between 2013-01-01 and 2017-12-31). We split the data into train and test. the separator date between the two sets is 2016-01-01.

Regarding the update frequency of the target DQN, we fixed it to 10.

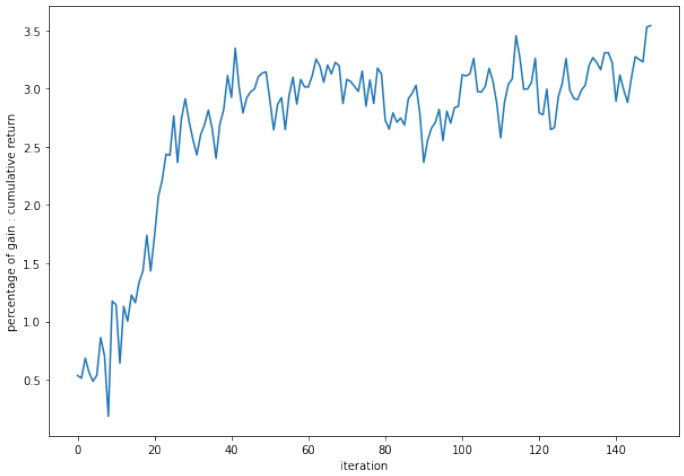

Figure 5: The cumulative return (gain) in percentage while training

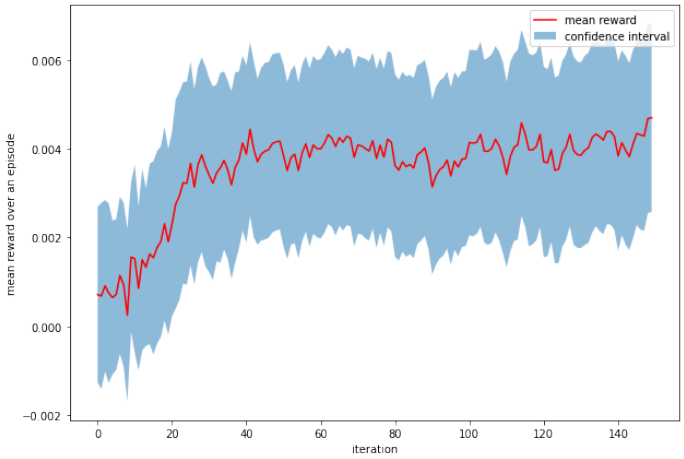

Figure 6: The mean daily reward while training with confidence interval

The cumulative return of the strategy is increasing with the training epochs. We converge to a good result very fast. On the other hand, we observe some instability in the training process.

In the next figure, we show the result of the agent trading on the train data (the data of the environment on which he is trained).
Actions are : 0=no action, 1=buy(long), 2=sell(short)

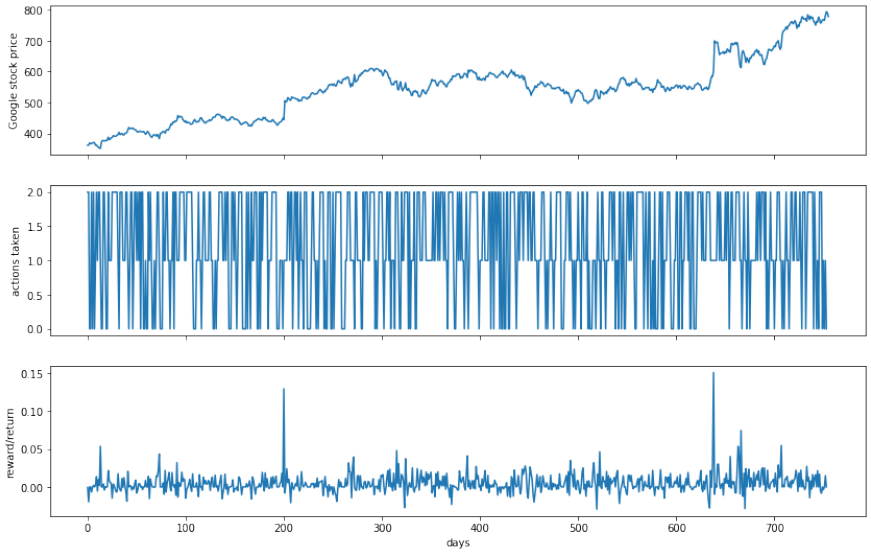

Figure 7: Result of the ddqn agent strategy on the google stock (train data)

The cumulative return (gain) of the agent on the training data :

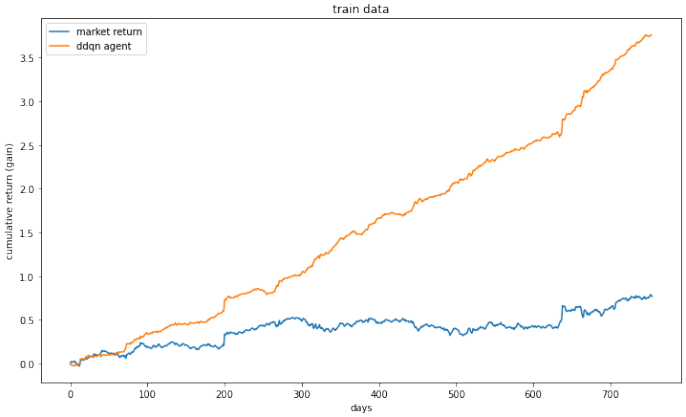

Figure 8: cumulative return on train data

we achieve a good gain using the ddqn strategy which is 3.5% of the initial price of the stock. A hold-and-buy strategy will outperform our daily trading strategy because obviously the google stock has a strong trend upwards. But if we take a more stationary stock our daily trading agent will be more efficient than a hold-and-buy strategy.

When we test on new stock data, we have a less high gain. But it is still good and the agent is overall outperforming the cumulative return of the market. A longer period of training data would be useful for the agent to capture more of the stock dynamic and learn a bigger space of state values .

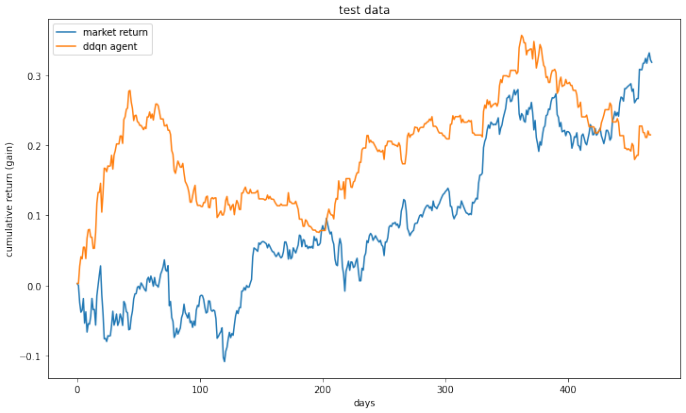

Figure 9: cumulative return on test data

5 Trading using Deep RL with Evolution Strategies

5.1 Evolution Strategies

Evolution strategies (ES) belong to the big family of evolutionary algorithms. The optimization targets of ES are vectors of real numbers, $x \in \mathbf{R}$.

This method used in optimization algorithms is inspired by natural selection, ie the belief that individuals with traits beneficial to their survival can live through generations and pass down the good characteristics to the next generation.

Technically, this ES algorithm tackles the following problem:
Let's consider a function f(x) that we want to optimize, for which we cannot compute the gradient directly

(but we still can evaluate f(x) given any x and the result is deterministic). Our belief in the probability distribution over x as a good solution for the optimization problem of f(x) is noted $p_\theta(x)$, parameterized by θ. The goal is to find an optimal configuration of θ.
Note that Here given a fixed format of distribution (i.e. Gaussian), the parameter θ carries the knowledge about the best solutions and is being iteratively updated across generations.

The algorithm starts with an initial value θ_0 and continuously updates by looping three steps:
1. Generate a population of samples $D = \{(x_i, f(x_i))\}$ where $x_i \sim p_\theta(x)$.
2. Evaluate the "fitness" of samples in D.
3. Select the best subset of individuals and use them to update θ, generally based on fitness or rank.

We present here the example of the **Simple Gaussian Evolution Algorithm** which is the most basic one and which we will use in our Reinforcement Learning agent:
The Simple Gaussian Evolution algorithm models $p_\theta(x)$ as an n-dimensional isotropic Gaussian distribution, in which only tracks the mean μ and standard deviation σ.
1. First generate the offspring population of size Ξ by sampling from the Gaussian distribution:
$D^{t+1} = \{x_i^{t+1} \mid x_i^{t+1} = \mu^t + \sigma^t y_i^{t+1}$ where $y_i^{t+1} \sim \mathbf{N}(x \mid 0, I), i = 1, \ldots, \Xi\}$
2. Select a top subset of λ samples with optimal $f(x_i)$ that we'll call elite set labeled as D_{elite}^{t+1} **3.** Estimate the new mean and std for the next generation using the elite set:

$$\mu^{t+1} = avg(D_{elite}^{t+1})$$

$$\sigma^{(t+1)^2} = var(D_{elite}^{t+1})$$

4. Repeat steps 2 and 3 until convergence.

5.2 Application in Deep Reinforcement Learning

The concept of using evolutionary algorithms in reinforcement learning can be traced back long ago, but only constrained to tabular RL due to computational limitations.

We propose to use the Natural Evolution Strategy as a gradient-free black-box optimizer to find optimal policy parameters θ that maximizes the return function $F(\theta)$. The key is to add Gaussian noise ϵ on the model parameter θ and then use the log-likelihood trick to write it as the gradient of the Gaussian pdf. Eventually only the noise term is left as a weighting scalar for measured performance.[5]

Let's say the current parameter value is $\hat{\theta}$ (the added hat is to distinguish the value from the random variable θ). The search distribution of θ is designed to be an isotropic multivariate Gaussian with a mean $\hat{\theta}$ and a fixed covariance matrix $\sigma\mathbf{I}$

$$\theta \sim \mathbf{N}(\hat{\theta}, \sigma\mathbf{I})$$

Equivalent to:

$$\theta = \hat{\theta} + \sigma\epsilon, \epsilon \sim \mathbf{N}(0, \mathbf{I})$$

The gradient for θ update is:

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{\theta \sim \mathbf{N}(\hat{\theta}, \sigma\mathbf{I})} F(\theta) &= \nabla_\theta \mathbb{E}_{\epsilon \sim \mathbf{N}(0, \mathbf{I})} F(\hat{\theta} + \epsilon\sigma) \\
&= \int_\epsilon p(\epsilon) \nabla_\epsilon log(p(\epsilon)) \nabla_\theta \epsilon F(\hat{\theta} + \epsilon\sigma) d\epsilon \\
&= \frac{1}{\sigma} \mathbb{E}_{\epsilon \sim \mathbf{N}(0, \mathbf{I})} [\epsilon F(\hat{\theta} + \epsilon\sigma)]
\end{aligned}$$

In one generation, we can sample many ϵ_i, i=1,...,n and evaluate the fitness in parallel. One important feature about this design is that no large model parameter needs to be shared. By only communicating the

random seeds between workers, it is enough for the master node to do parameter update. This approach is later extended to adaptively learn a loss function.
Here is a pseudo-code of the ES-agent:

Algorithm 2 Parallelized Evolution Strategies

Input: Learning rate α, noise standard deviation σ, initial policy parameters θ_0
Initialize: n workers with known random seeds, and initial parameters θ_0
for $t = 0, 1, 2, \ldots$ **do**
  for each worker $i = 1, \ldots, n$ **do**
    Sample $\epsilon_i \sim \mathcal{N}(0, I)$
    Compute returns $F_i = F(\theta_t + \sigma\epsilon_i)$
  end for
  Send all scalar returns F_i from each worker to every other worker
  for each worker $i = 1, \ldots, n$ **do**
    Reconstruct all perturbations ϵ_j for $j = 1, \ldots, n$ using known random seeds
    Set $\theta_{t+1} \leftarrow \theta_t + \alpha \frac{1}{n\sigma} \sum_{j=1}^{n} F_j \epsilon_j$
  end for
end for

Figure 10: The algorithm for training a RL policy using evolution strategies

To make the performance more robust, we can adop virtual batch normalization (BN with mini-batch used for calculating statistics fixed) and mirror sampling (sampling a pair of (ϵ, ϵ) for evaluation).

5.3 Experiments and results

We tested the ES agent in the same trading environment as the two other agents. Were also tested on the same stock (Google) and the same period (between 2013-01-01 and 2017-12-31).
While training and when testing on the training set (in sample), we get a very high cumulative return as the agent learn very fast when to long and when to short the stock.

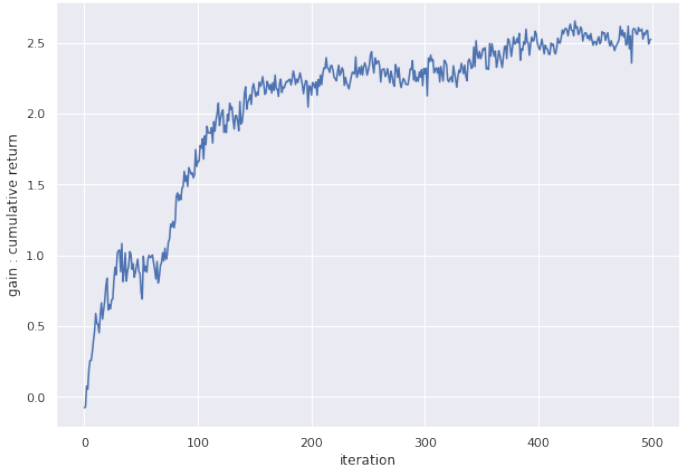

Figure 11: The cumulative return while training

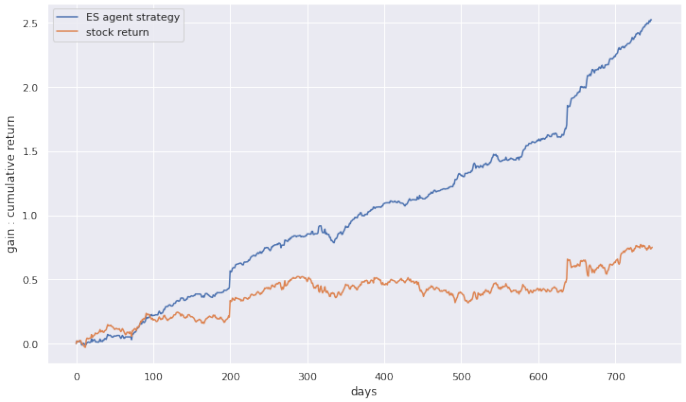

Figure 12: The cumulative return of agent on the training set

The results on the test set (out of sample) is obviously less high, but we still get a lucrative trading strategy where the agent seems to avoid periods of fall in the underlying stock by shorting it and still capture periods of growth.

Figure 13: The cumulative return of agent on the test set

We can improve this performance by training on a larger period of time. Two years of training is very low in the case of trading strategies as the market would probably follow only one scheme and the agent wouldn't then learn different states of the market (Bullish/Bearish, Volatile/Stable, Growth/Recession, ...).

References

[1] Huge stock market dataset. https://www.kaggle.com/borismarjanovic/price-volume-data-for-all-us-stocks-etfs.

[2] D. Ormoneit & al. Kernel-based reinforcement learning. 2002.

[3] F. Bertoluzzo & al. Testing different reinforcement learning configurations for financial trading: Introduction and applications. 2012.

[4] David Silver Hado van Hasselt, Arthur Guez. Deep reinforcement learning with double q-learning. 2015.

[5] Xi Chen Szymon Sidor Ilya Sutskever Tim Salimans, Jonathan Ho. Evolution strategies as a scalable alternative to reinforcement learning. 2017.