Neural Networks Project

2019-2020 Class

Nicola Di Santo

March 25, 2020

Abstract

The main focus of the project was to reimplement the paper by Lucarelli and Borrotti [1]. The paper simply tries to adapt some Deep Reinforcement Learning techniques, namely DQN [2], Double DQN [3] and Dueling Double DQN [4], to allow an agent to trade cryptocurrency stocks. The trading environment gives the possibility to either buy, hold or sell a position, the profit made during trading is measured with Sharpe ratio index or a simple profit difference function. For all technical detail we refer to [1].

Quick Review on Lucarelli and Borrotti Job

We would like to quickly recap the difficulties we encountered trying to replicate this job from the lack of information in the Lucarelli and Borrotti paper. Indeed the first and most important criticism we can make is that they miss a feature engineering description, so we do not know how they manipulate dataset except for the hourly aggregation step. Second but not less important, we find some misleading information about the Deep Q Network architecture description without any theoretical explanation. Indeed, citing textually their architecture description we can read: *"The Q-learning trading system based on the D-DQN is composed by 2 CNN layers with 120 neurons each. In the case of DD-DQN, 2 CNN layers with 120 neurons each are followed by two streams of FC layers: the first with 60 neurons dedicated to estimate the value function and the second with 60 neurons to estimate the advantage function"*. Our objections to this statement are the following:

- Since we do not have any feature description we can not imagine which kind of convolution (one, two or three dimentional) is used.

- Does it really make sense to speak about convolutional layer neurons instead of number of filter, number of channels, padding, stride, dilatation?

- Which are the considerations that makes them diverge from the classical Dueling architecture and make them use only *"the first [...] 60 neurons dedicated to estimate the value function and the second [...] 60 neurons to estimate the advantage function"* instead of the entire flattened layer?

- Furthermore we can read *"In both cases, the number of epochs is set to 40 as well as the batch size."* and we think it is generally a low number of episodes to obtain a well trained agent.

- There is not any reference to the Replay Memory Buffer technique and we can not know if it has been used or not.

- The simple DQN architecture used to compare result in not specified.

We have tried to face this work doing things consciously and then we slightly diverged from their implementation as explained in the next section.

Our Implementation

The delivered .zip file contains: an ipython notebook from Google Colab, the final dataset we should use and six pretrained models. The code implemented in Colab has been implemented and delivered also as a python script. The Colab notebook contains the results that are reported below so if needed they can be checked out and it is also delivered just in case there is the needing to re-train the models exploiting Google's GPU. The python scripts instead are an organized version of the code in Colab with two executables:

- *main.py* allows to test the pretrained models (no training required) and it will produce both a plot and a table with the performance report (in terms of cumulative percentage return). Its execution flow is simple: instantiate the agents (dqn, double dqn, dueling double dqn), instantiate the environment (training and test) with sharpe and profit reward, train the models, test the models, print result. Note: to execute the script correctly all the files (scripts, dataset and models) must be in the same directory.

- *train_test.py* will produce the same output of *main.py* with the only difference that it is going to train a new model. Note: a runtime warning will trigger if the user execute the *train_test.py* script using cpu since it will require lots of time.

The non executable scripts instead contains:

1. *Environment.py* the definition of the stock market environment. In particular our environment simulates the stock market by storing the dataset of bitcoin prices. Its implementation recall the Open Ai Gym Environment implementation [5] with the *Environment.reset()* and *Environment.step()* methods. Its behavior is quite simple: the reset method just initialize the env. to be properly used while the step method, given an action from the agent, returns the reward for the given action and a flag telling us if we are in a final state. The reward function is specified when the env is initialized and can either be a profit function or a Sharpe ratio index as defined in [1] .

2. *Agent.py* contains the definition of the agent. This class simply instantiate the network as specified with the appropriate parameter, define the training and testing loops and implements the ϵ-greedy action selection. Test and Training methods are basically the same loop: get the state, choose and execute an action, collect reward. The only difference is that the training loop will call an optimization methods that will optimize the Q network parameters. There are two optimization functions, one for training using the double Q network formula, the other using the simple deep Q network formula as described in the section Teoretichal Background.

3. *models.py* contains only the PyTorch implementation of the models used in [1]

4. *utils.py* contains the replay memory buffer and Transition definition as well as some useful functions to report test results and load data.

It is important to highlight that, according to what is stated in [1], even if the dataset is of 34.000 samples, each time we instantiate the environment

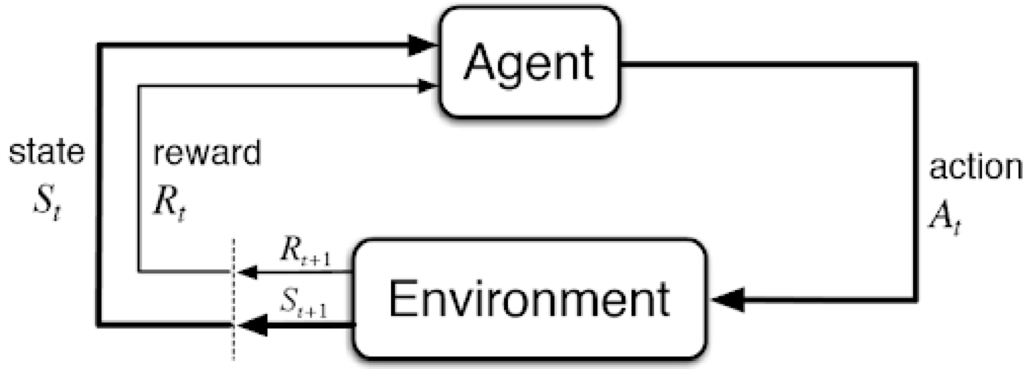


Figure 1: Simple but eloquent picture about reinforcement learning interaction

it is made only of 4000 samples (trading period). The first of those sample is selected randomly and all the others are the subsequent instant of time. Due to this randomized setting it is very probable that testing a pretrained model will not produce the same performances as reported below because the agent is trained on a different sub-sequence.

Theoretical Background

We briefly describe the main concept behind our agent model. We have used experience replay memory for training our DQN. It stores the transitions that the agent observes, allowing us to reuse this data during training. By sampling from it randomly, the transitions that build up a batch are decorrelated. It has been shown that this greatly stabilizes and improves the DQN training procedure.[1]

Our aim will be to train a policy that tries to maximize the discounted, cumulative reward $R_{t_0} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} * r_t$, where R_{t_0} is also known as the return. The discount, γ, should be a constant between 0 and 1 that ensures the sum converges. It makes rewards from the uncertain far future less important for our agent than the ones in the near future that it can be fairly confident about.

The idea behind Q-learning is that if we had a function $Q^* : StatexAction \rightarrow R$, that could tell us what our return would be, if we were to take an action in a given state, then we could easily construct a policy that maximizes our rewards: $\pi^*(s) = argmax_a Q^*(s,a)$. However, we don't know everything about the world, so we don't have access to Q^*. But, since neural networks are universal function approximators, we can simply create one and train it to resemble Q^*.

For our training update rule, we'll use a fact that every Q function for some policy obeys the Bellman equation: $Q^\pi(s,a) = r + \gamma Q^\pi(s', \pi(s'))$. The difference between the two sides of the equality is known as the temporal difference error, δ: $\delta = Q(s,a) - (r + \gamma max_a Q(s',a))$.

To minimise this error, we will use the MSE loss over δ as specified in [1]. We calculate this over a batch of transitions B of size 40, sampled from the replay memory: $Loss = \sum_{(s,a,s',r) \in B} L(\delta)$, with $L(\delta)$ the MSE loss over δ.

We can note that maximizing over Q overestimated values as such is implicitly taking the estimate of the maximum value. This systematic overestimation introduces a maximization bias in learning. The solution involves using two separate Q-value estimators, each of which is used to update the other. Using these independent estimators, we can unbiased Q-value estimates of the actions selected using the opposite estimator. We can thus avoid maximization bias by disentangling our updates from biased estimates [3]. The Q function became then $Q^\pi(s,a) = r + \gamma Q^\pi(s', argmax_a Q(s',a;\theta_t); \theta'_t)$, that

[1] https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

is the definition of the Double Deep Q Learning.

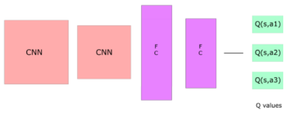
For the dueling architecture, instead, given the action value $Q^{\pi}(s,a)$ and state value $V^{\pi}(s,a)$ we can define the advantage as $A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s,a)$. When forwarding feature trough the network we split the flow into two different linear estimations (right after the convolutions), one for $V(s,a)$ and one for $A(s,a)$. To merge the branches and obtain the Q values then we can just compute $Q(s,a) = V(s,a) + (A(s,a) - \frac{1}{|A|}\sum_{a'} A(s,a'))$

Feature Extraction

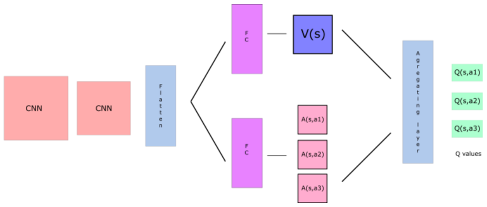
To correctly extract feature from a time series and design a Q function approximator we are inspired from Nathan Janos and Jeff Roach work [6].
As a feature extraction step we have decided to consider only the *"Close"* price of the crypthocurrency stock to maintain training time low and since we obtained quite interesting results we decided to do not include other features. In detail, if we denote the stock price at time t as p_t, then a single feature is $x = [p_t, p_{t-1}, p_{t-2}, ..., p_{t-23}]$ representing a short history of stock prices of length 24 hours.

Network Architectures

Given our features we have decided to implement the convolutional layers composed by a one dimentioanl convolution, a max pooling filter and a leaky relu (the only parameter specified in [1]).



(a) DQN Architecture



(b) Dueling DQN Architecture

In particular the first 1-D convolution will use data with a single channel in input and 24 features as described above, a kernel of size 8, no padding and both stride and dilatation are set to 1, and will produce as output 64 filters. Then we apply the activation funciton and then a Max Pooling with kernel size set to 2. The second convolution layer is similar to the first one with the only difference that it uses a kernel size of 4 and it has 64 channels in input. The Dueling Q network architecture is exactly the same as DQN for the convolutional layers but the stream then is splitted according to what is described in [4] and recalled in the Teoretichal Background section of this report.

Result Comparison

Follows a comparison of our result with the Lucarelli and Borrotti [1] results in terms of the average percentage returns (econometric measure) over ten trading periods and some plots with 95% confidence intervals of our results.

		Lucarelli and Borrotti — Ours		
Trading System	Avg. Return (%)	Max. Return (%)	Min. Return (%)	Std. Dev.
ProfitD-DQN	**3.74** — 1.52	21.31 — **83.98**	-10.74 — **-3.02**	**4.87** — 6.43
ProfitDD-DQN	**4.85** — 2.27	17.34 — **85.79**	-8.49 — **-5.04**	**5.10** — 7.68
ProfitDQN	**2.32** — 2.22	22.59 — **68.56**	-17.97 — **-7.10**	7.93 — **7.04**
SharpeD-DQN	**5.81** — 0.04	**26.14** — 12.36	**-5.64** — -8.68	5.26 — **1.03**
SharpeDD-DQN	**3.04** — 0.00	**13.03** — 1.33	-8.49 — **-0.91**	3.81 — **0.04**
SharpeDQN	**1.83** — 0.72	15.08 — **21.30**	**-9.29** — -10.19	5.46 — **2.85**

Table 1: Results Comparison (to check those result just open the attached colab notebook)

As we can immediately see the average on 10 test of the cumulative return (computed as $\frac{(currentstockprice)-(initialstockprice)}{(initialstockprice)}$) of [1] is always superior to ours. Nonetheless our models reaches a lot higher maximum return and loose less in terms of cumulative reward with all the models using a profit reward. From this point of view we might say that out model trade-off average return with the ability to minimize the negative peaks of cumulative return. Under those conditions also our standard deviation is a bit higher than the one of Lucarelli an Borrotti so the models show less stability. On the other side, considering a Sharpe ratio based reward function we can see how our model has huge difficulties into increasing the average return but we are paid in stability of the model: the std deviation is very low wrt values reported in [1] .

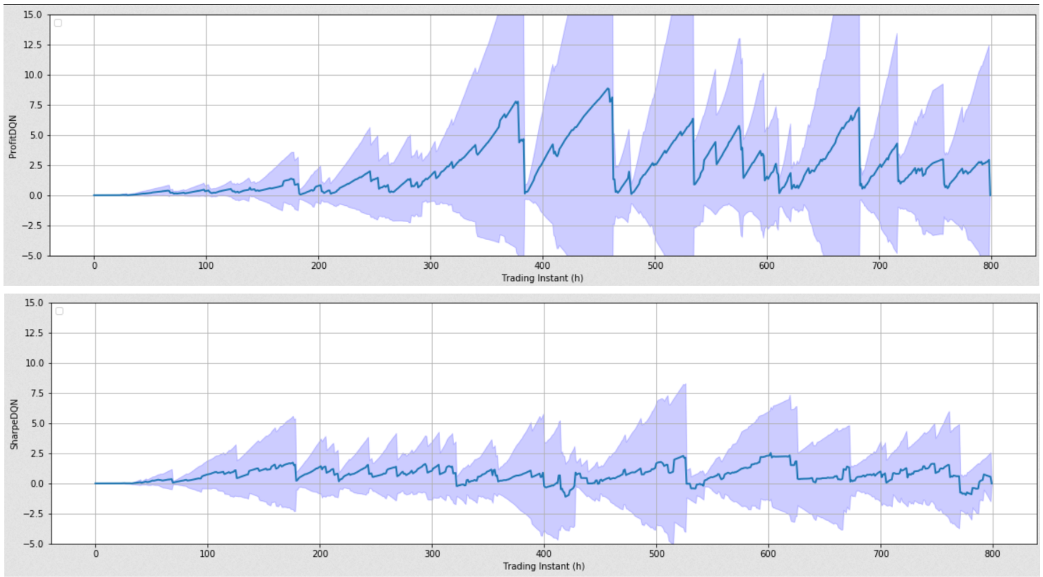


Figure 3: Our DQN Avg. Return

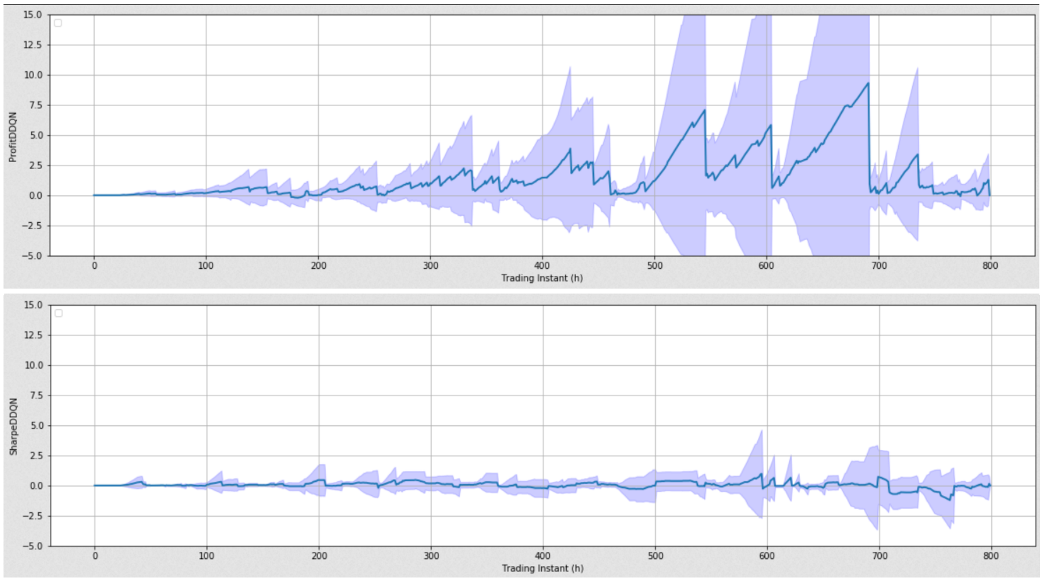


Figure 4: Our Double DQN Avg. Return

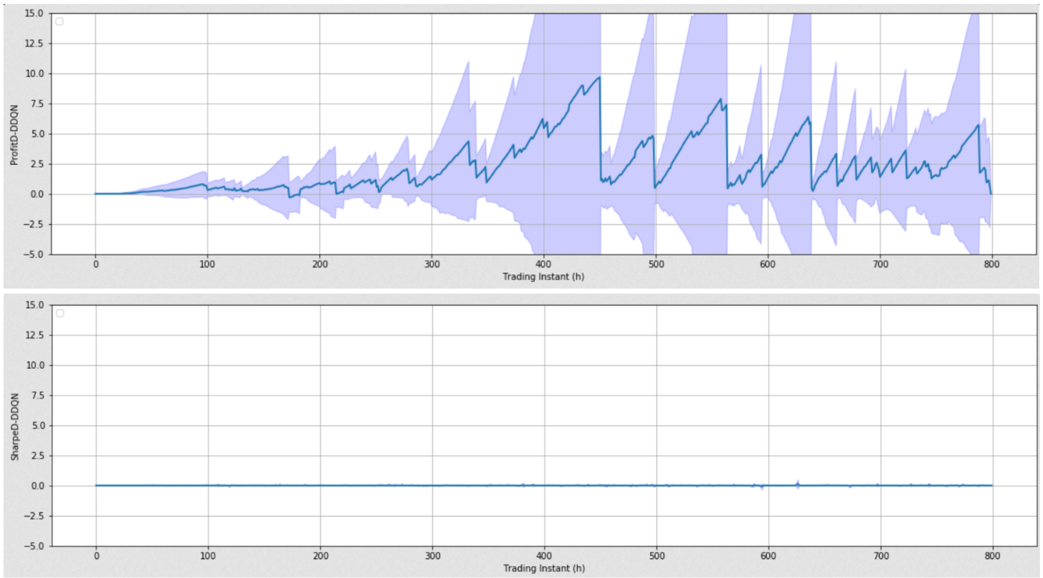


Figure 5: Our Dueling Double DQN Avg. Return

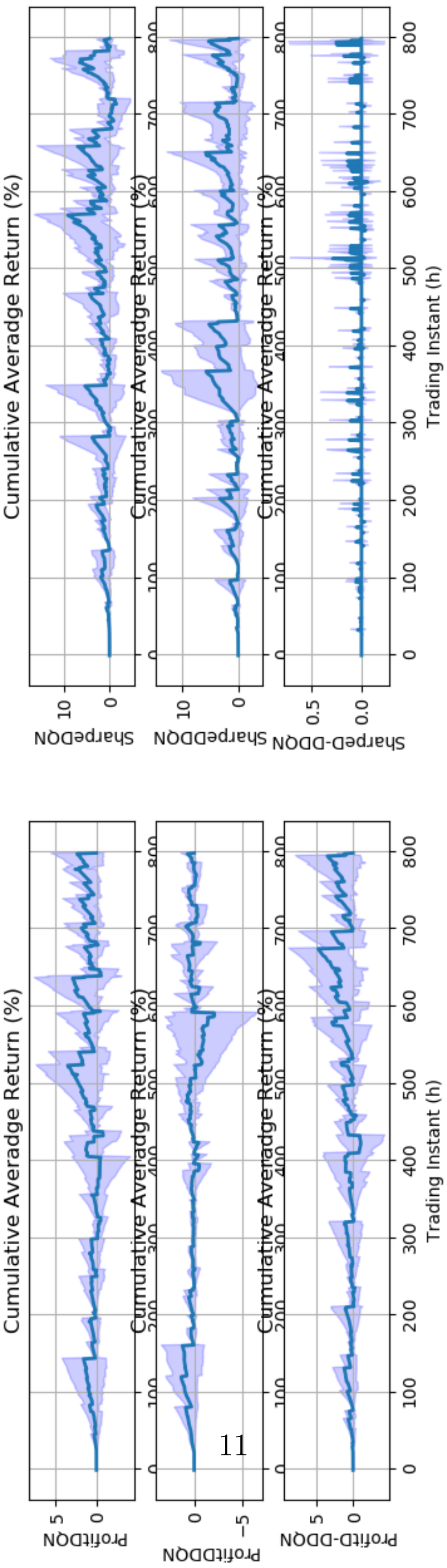


Figure 6: Other testing results

References

[1] Giorgio Lucarelli and Matteo Borrotti. A deep reinforcement learning approach for automated cryptocurrency trading. pages 247–258, 2019.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[3] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[4] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning. 2015. cite arxiv:1511.06581.

[5] Getting started with gym. http://gym.openai.com/docs/.

[6] Jeff Roach Nathan Janos. 1d convolutional neural networks for time series modeling. https://www.youtube.com/watch?v=nmkqwxmjwzg.