Trading using Deep-Q Learning

Javier Montero

October 2018

Abstract

We present here the capstone project report for the Machine Learning Nanodegree from Udacity. We have borrowed the Deep Q Learning algorithm originally developed by Google Deepmind and adapted to play the financial trading game. The agent developed was trained to learn trading Bitcoin in our simulated BitMEX exchange.

1 Definition

1.1 Overview

In this paper, we study the possibility to train an agent to make long term profitable trading decisions operating in a bitcoin exchange. We use the historical prices since January 2017 of one bitcoin derivative product from BitMEX[1] one of the biggest digital trading exchange.

The agent is trained using the concepts of Deep Reinforced Learning algorithm [Mnih et al. 2015] first used by Deepmind Technologies to play Atari games. The techniques fall in the area of Reinforcement Learning within the realm of machine learning science.

1.2 Problem Statement

With the success of RL techniques beating humans playing games, we were curious to test whether if those achievements can be seen in the field of financial trading.

[1] http://bitmex.com

Our test hypothesis is:

"Deep Reinforcement Learning agent can make profit trading Bitcoin"

Our purpose was to create a framework that allows us to test the performance of Reinforcement learning agents. For that, we developed a solid trading simulator and benchmark models that can be used to test and compare different state of the art reinforcement learning architectures like Deep Deterministic Policy Gradients (DDPG) or Asynchronous Actor-Critic Agents (A3C).

1.3 Metrics

But we have not used any of the mentioned metrics. Making a model in RL that learns is a hard task and it is very sample inefficient, taking lots of time to run the experiments. For those reasons we used a very simple and straightforward metric that show us clear and intuitively any performance improvement made by the model: the **profit and loss (PnL)**. The *PnL*, as the accumulated profit or loss of the individual trades, is the simplest metric and although it does not take risk into account, is the most appropriate metric for our task.

As a future improvements, and once the agent begin to generate a positive PnL, we could use **alpha**[2] and **beta**[3] metrics. Alpha gauges the performance of an investment against a market index or benchmark which is considered to represent the market's movement as a whole. Beta, on the other hand, measures volatility or risk.

Sharpe ratio[4] is also commonly used to measure to trading decisions and it characterizes how well the return of an asset compensates the investor for the risk taken

[2] https://en.wikipedia.org/wiki/Alpha_(finance)
[3] https://en.wikipedia.org/wiki/Beta_(finance)
[4] https://en.wikipedia.org/wiki/Sharpe_ratio

2 Analysis

2.1 Data Exploration

We designed our study to be *asset agnostic* as it can be used with any market or *commodity* like FX, stocks, indexed futures, ... We have used, just for convenience, the **Bitcoin/US dollar Perpetual Inverse Swap Contract** (XBTUSD) traded in the BitMEX market. This offers two important features useful for the algorithm:

- Bitcoin markets operates 24/7, giving us a continuous dataset and eliminating the complexities of market opening and closing behaviour.
- XBTUSD in BITMEX allows *short selling* which means that one can make profit selling high and buying low, doubling the trading opportunities.

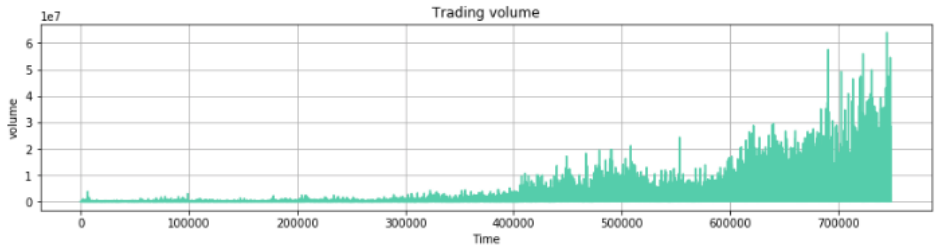
We downloaded one minute period price data from BitMEX exchange using the provided API[5] from January 2017 to June 2018 consisting fo 748800 data samples. This price dataset consist of the period timestamp, open, high, low and close price values as well as the volume (OHLCV) traded in each minute:

Table 1: BitMEX XBTUSD dataset

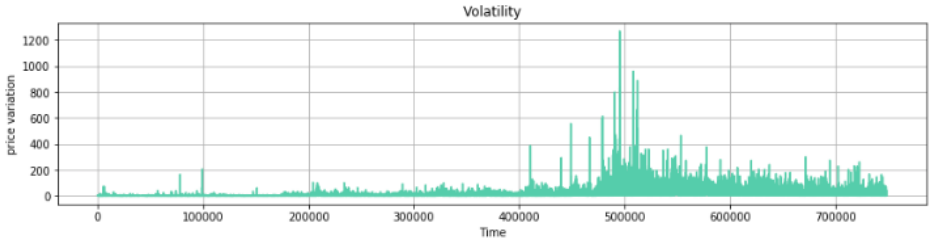
timestamp	open	high	low	close	volume
2017-01-01T00:01:00.000Z	968.29	968.76	968.49	968.7	12993
2017-01-01T00:02:00.000Z	968.7	968.7	967.2	968.43	73800
2017-01-01T00:03:00.000Z	968.43	968.0	967.21	967.21	3500

It is worth noting that the interest in Bitcoin has increased exponentially over the last year, where the year trading volume in the BitMEX exchange has increased a 161% since 2017 taking into account just the first 6 months of 2018.

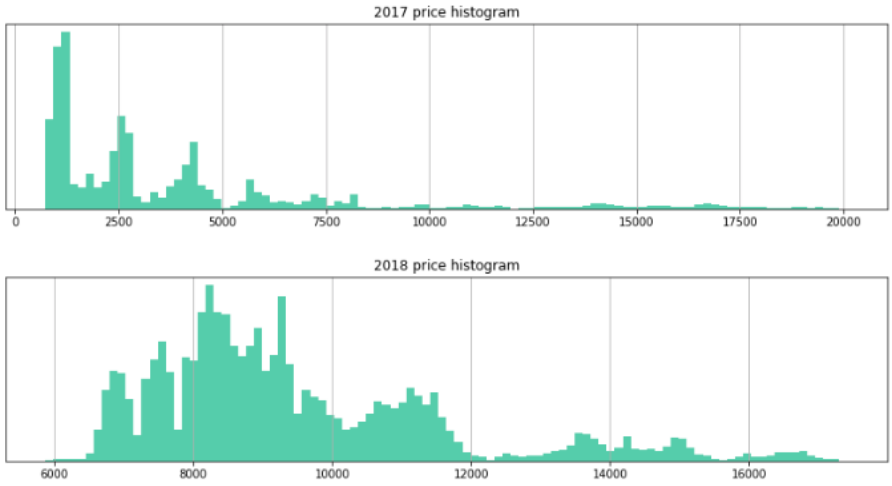
[5] https://www.bitmex.com/api/explorer/



Bitcoin is an very volatile asset, fact increased in the case of the XBTUSD as BitMEX exchange allows up to 100x leverage. During the testing period, the maximum price fluctuation within a minute was 1.271 USD.



Over the sampling period the closing price has moved from a minimum of 728 USD to a maximum of 20,090 USD at the beginning of 2018. To further note how much variance in price this asset shows, the standard deviation σ was 4,354. Below we show in what ranges the prices is moving in 2017 and 2018.



2.2 Algorithms and Techniques

The main motivation to use the deep reinforcement learning is to see if the recently success for this techniques *migrates* to the financial trading task.

Reinforcement learning is an important type of Machine Learning technique where an agent learn how to behave in a environment by performing actions and seeing the results. It will learn from the environment interacting with it and receiving rewards for performing actions.

This can formally be formulated as a Markov Decision Process (MDP). We have an agent acting in an **environment**. Each **time step** t the agent receives as the input the current **state** S_t, takes an **action** A_t, and receives a **reward** R_{t+1} and the **next state** S_{t+1}. The goal of the agent is to maximize the expected future returns:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

The **agent** is the algorithm we want to learn to trade a in simulated market.

The **environment** is the simulation of the exchange where the agent operates (acts) making trades in the hope that they will be profitable.

The environment also shows the agent the **state**, a representation of the environment itself that the agent uses as the features for its prediction model. The **state** at each time step t is the historical trading prices and volume plus the *position* taken by itself (buy, sell or hold).

As we used minute data of XBTUSD from the market, each **time step** is a minute. Although the exchange simulator (environment) we have developed allows the use different periodicity in a transparent way.

The **action** space modeled is a discrete set the actions sell, hold or buy coded as 0, 1 and 2 respectively. The *hold* action means do nothing.

The **reward** function is one of the most important factors to determine the learning or not of our agent [Sutton et al. 1998], and designing a reward function is difficult [Irpan 2018]. We have defined our reward function as the log returns of the portfolio value.

$$r_t = log \frac{p_t}{p_{t-1}}$$

Where the portfolio value p_t is the Realized PnL + Unrealized PnL (open positions)

The original Deep-Q Learning algorithm learns receiving only the pixels and the game score as inputs, using convolutional neural networks to process as part of the deep neural network architecture. We have replaced the convolution layers by fully connected ones.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    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 $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

One of the important idea behind the algorithm is the **Experience replay** that has proven to be an effective mechanism for improving both the data efficiency and stability of deep reinforcement learning algorithms [Mnih et al. 2015]. The main idea is to store transitions in a replay buffer, and then apply learning updates to sampled transitions from this buffer.

Deep Q Learning uses a Neural Network to estimate the Q-value function. The output is the corresponding Q-value for each of the action. The neural network outputs an assigned probability for each action.

The network has one input layer, which takes the state size space. Then there are three hidden layers layers with 128, 256 and 96 hidden units respectively. Activation functions are rectified linear units (ReLU). The loss function is chosen to be mean squared error, and the optimizer is the Adam optimizer.

2.3 Benchmark

We used a **random trading agent** to compare our agent with. The purpose of this agent is to buy or sell randomly, following the same constraints as

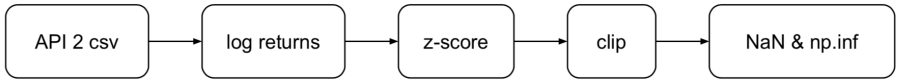
the real agent.

As the number of operations have a huge impact in the performance due to the fees, we limit the number of the trades the random trader can generate to the same amount as the real agent for the same period. We have limited this number to around 450 trades for a period of 3 month of data.

3 Methodology

3.1 Data preprocessing

The data is collected from the **BitMEX API**. This is a tricky process as BITMEX has denial-of-service and server overload protection mechanisms. And as we need to make a lot of server calls to retrieve the entire historical data, we have to include pauses in the data gathering mechanisms. This was a very time consuming task as the delays were learn by trial and error.



The dataset is then saved as **CSV file**.

To feed the data into the algorithm we extract the OHLCV (open, high, low, close, volume) data and first calculate the **log returns** on every column except the timestamp. After that, we perform a **running z-score**[6] normalization of period 60 (1 hour of data). Finally we **clip** the dimensions to 3 to eliminate the outliers.

The return is defined as the actual price minus previous period price.

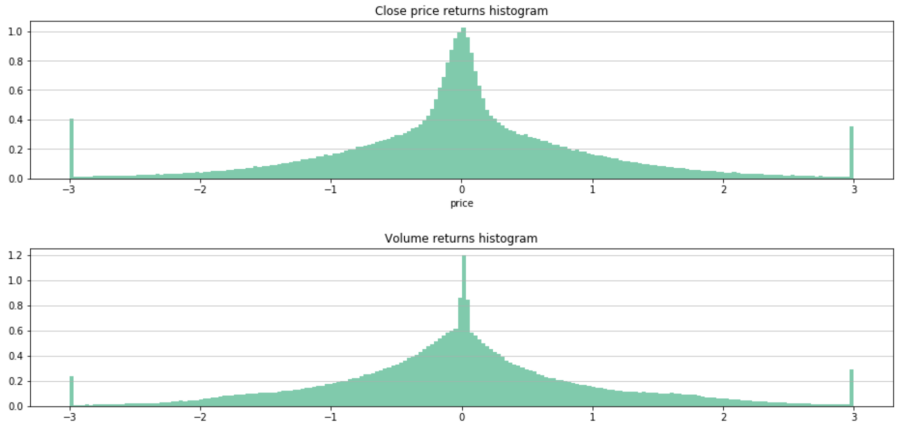
A z-score or standard score indicates how many standard deviations an element is from the mean.

We tackle the non numeric values (NaN, np.inf and -np.inf) searching for them and zeroing. This is especially important because although the BitMEX data is perfectly clean, the z-score algorithm can introduce this values if there is no change in the data during periods longer than the rolling window (60 in our case).

[6] https://en.wikipedia.org/wiki/Standard_score

Below is the close price evolution of the complete dataset:

The histogram representation of the dataset after preprocessed shows a nice normal distribution with peaks in the extremes. The peaks shows that the dataset has a large number of outliers, all clipped to 3.



3.2 Implementation

The implementation took place around 3 classes and 1 task.

Data.py CSV reading and data preprocessing tasks described above.

Market.py One of the principal component responsible of the trading simulation, position management and bookkeeping of the trading actions to measure its performance with the metrics defined. This class, the environment, also logs the actions taken for analysis and bug tracking. The **reward function** is also implemented in this class.

DQNAgent.py The Deep Q Network agent itself.

trader.py This is the task itself that implement all the classes described and orchestrates a simulated *trading session* in which the agent interacts with the environment.

3.2.1 Environment details

We applied the 0.075% *taker fee* that BitMEX exchange charges to each trade (market orders). BITMEX, and many other exchange, offers a much lower fee to liquidity makers, that is, traders placing *limit orders*.

The minimum trade size is 1 XBTUSD, market as always enough liquidity and fillings are complete. In conclusion our simulated exchange has **no slippage**.

There is **no spread**. We simply the market assuming there is no bid and ask prices, so no spread exist.

The **orders are always executed and at the desired price**. XBTUSD is a highly volatile asset and during periods of extreme volatility the price seen and the execution price can be very different. Also, in periods of high volume there are chances that orders are rejected by the market.

The agent start with an initial balance of 10.000.

3.2.2 Description of the trading Markov Decision Process

The position contains the action taken in each step, corresponding to one of the actions in the action space described below.

The *state space* is a 21 dimension vector corresponding to the last 4 most recent OHLCV data as well as the position.

The *action space* is coded as 0, 1 or 2 corresponding to the actions Sell, Hold and Buy. Being Hold the action of doing nothing.

The agent can only have an open position of 1 unit short or long at every moment. That means that, at every step, there are can be *forbidden actions*. In case that the current position is long (buy +1) and the agent again outputs a buy action in the next time step, there will be no trade executed.

3.2.3 Testing strategies

We created two strategies to test the reinforcement learning trading algorithm developed, the *look-forward* one, that slices the data in consecutive train and test sets and the *online* one, resembling real-time trading, were the training takes place at the same time as the acting.

Table 2: *look forward strategy with 5:1 ratio*

day1	day2	day3	day4	day5	day6	day7	day8	day9	day10
train	train	train	train	train	**test**				
	train	train	train	train	train	**test**			
		train	train	train	train	train	**test**		
			train	train	train	train	train	**test**	
				train	train	train	train	train	**test**

Due to the extremely long execution times we decided to use mainly the *online strategy* but we were able to complete one execution of the *look-forward strategy* so we could compare both.

3.2.4 Main Challenge

Maybe the biggest problem to even determine the performance of our agent is that, due to the nature of the Deep Reinforcement Learning algorithm, it is very **sample inefficient** and is extremely slow to run.

The need to iteratively sample each of the data samples makes parallelization unfeasible.

The size of the neural network is also very small to make us of the GPU. In fact, we compiled tensorflow[7] for CPU and GPU in our OSX environment and found the GPU version takes approximately **10 times longer** than the CPU version per episode. So although we have a high-end *NVIDIA GTX 1080ti* GPU, we were unable to take advantage of it.

Time taken by the agent to run with the same 3 month of data:

[7]The compilation of tensorflow from sources in OSX were a headache by itself. As OSX is not supported the the tensorflow team, there are no instructions about the combination of OSX, bazel, Xcode and tensorflow version to successfully build the image.

- 16 hour in a *online strategy*.
- 120 hour (5 days) in a *look-forward strategy* with a ratio of 5:1 of train:test days and training for 5 episodes.

Citing [Irpan 2018], "deep reinforcement learning can be horribly sample inefficient".

3.2.5 Tuning the model

Tuning the model parameters were very difficult given the time taken by the agent to perform a single trading session. Even though we could perform tests with different values of:

- Neural Network architecture (number of hidden layers and number of neurons in each layer).
- Batch and memory size
- The discount factor γ
- The learning rate

We started testing our agent with a neural network with 2 hidden layers of 12 and 24 hidden units each, a batch size of 24 and a replay memory of 10,000. Our final hyper-parameters used are listed below.

Table 3: Hyper-parameter values after tuning

Hyperparameter	Value
Learning rate	0.0005
Replay memory size	7200
Batch size	96
Discount factor	0.98
NN hiden layers	3
NN neurons	128/256/96

4 Results

4.1 Generalities

We restricted the *trading sessions* to 3 months of data (132.480 samples) using the *online testing strategy*. This allow us to have feedback about the performance at tune the parameters accordingly. Each trading session take approximately 14 hour to complete.

We run around 61 trading sessions that roughly takes more than 1500 hours of CPU time.

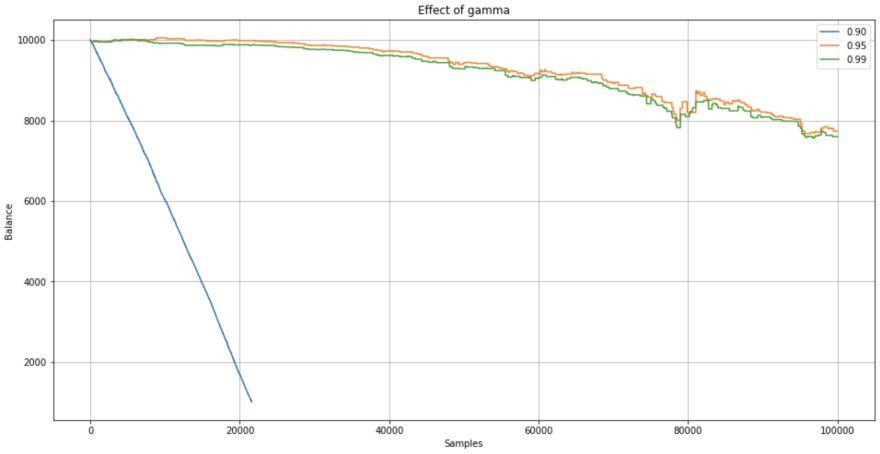
4.2 Effects of discount factor γ

When γ approaches one, future rewards are given greater emphasis about the immediate reward. When it is zero, only immediate rewards is considered.

We found that γ in the learning process has a huge impact in the performance of the agent. With gamma around 0.90 the agent becomes *trade hungry* making a lot of trades, most of them non profitable. Our intuition is that, given the huge impact of fees, the profitable strategy is to an open position (either short or long) and hold for a period of time. A high frequency trading (HFT)[8] strategy is only profitable for corporations not individuals given the fee levels.

This graph shows the behaviour described above. We can see that with a gamma of 0.90 the agent lost the whole initial balance in just 20.000 steps ($\approx$ 13 hour of data)

[8] https://en.wikipedia.org/wiki/High-frequency_trading



The sweet spot was at $\gamma \approx 0.98$.

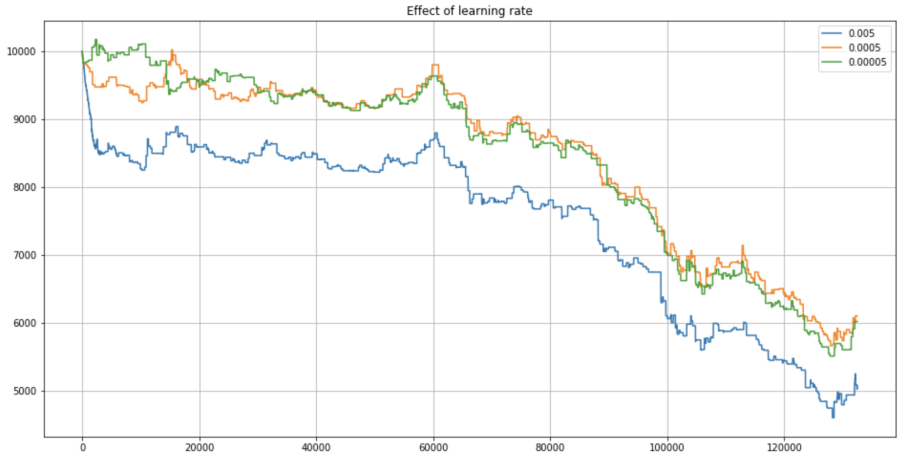
4.3 Effects of Adam optimizer learning rate

Adam optimizer for the Neural Network was used because it is fast and it is suggested as the default optimization method for deep learning applications.[9] The learning rate tells the optimizer how far to move the weights in the direction of the gradient for a mini-batch.

If the learning rate is too high, then, training may not converge or even diverge. Weight changes can be so big that the optimizer overshoots the minimum and makes the loss worse.

Empirically we discovered that a learning rate of 0.005 is too high making the model to perform worst. We see no difference in performance decreasing the learning rate from 0.0005 to 0.00005, so we fixed it at 0.0005.

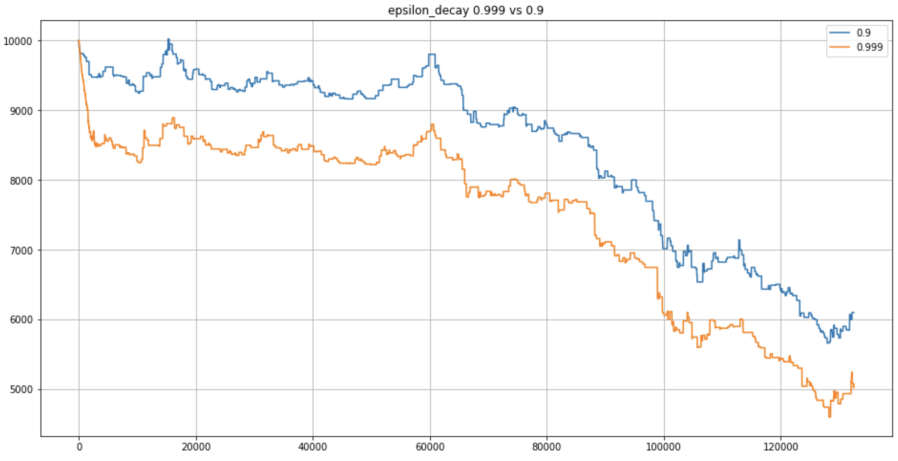
[9] https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/



4.4 Tuning ϵ-greedy

ϵ-greedy is an algorithm to solve the exploration-exploitation dilemma, in which the agent first start exploring the environment taking actions randomly and slowly entering in a exploitation phase where it start taking the most rewarded actions known so far. The ϵ decay rate determines how fast the agent pass from exploration to exploitation phase.

We found that delaying exploitation leads to a decline in performance. Again, the effect of the fees have a huge impact in any agent taking random decision.

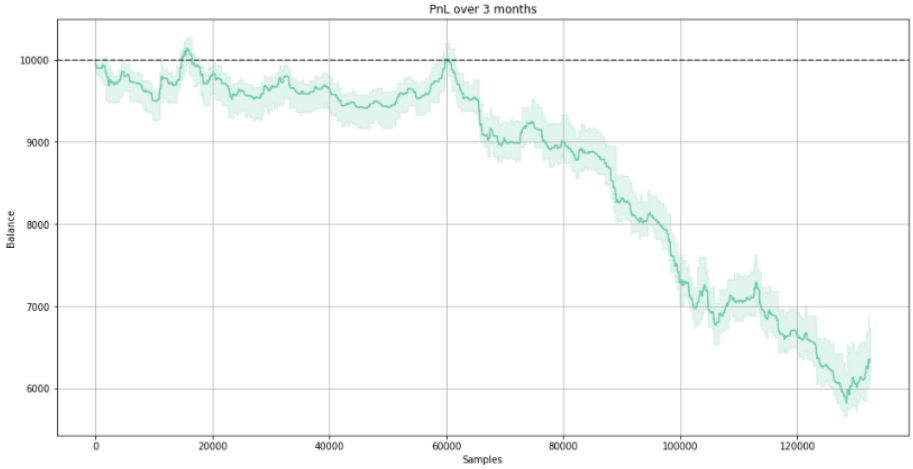


4.5 Results

The agent did not learn to find enough profitable actions to make a positive PnL.

The agent's trading sessions show a low performance, where in just 1 of the session the agent finalized with a positive PnL balance.

Below is the cumulative PnL or balance in a random period of three months showing the performance of the agent. The trading session was executed 4 times to appreciate the mean and variance of the results.

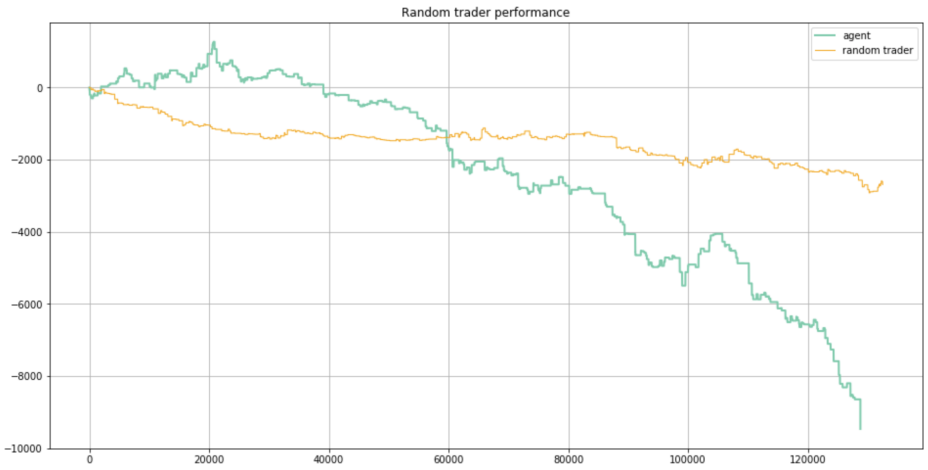


The **random agent benchmark** was executed limiting the number of allowed trades. Number of trades has a high impact on performance due to the fees. We measured the mean number of trades executed by our agent during our standard period of 3 months of data to be around 450.

We run the random trader 20 times to get some statistics about its behaviour. We found that it outperformed our agent by a factor of 4. The mean final balance was -1,100 in the case of the random agent and -4,600 in our agent, showing also less variance than ours.

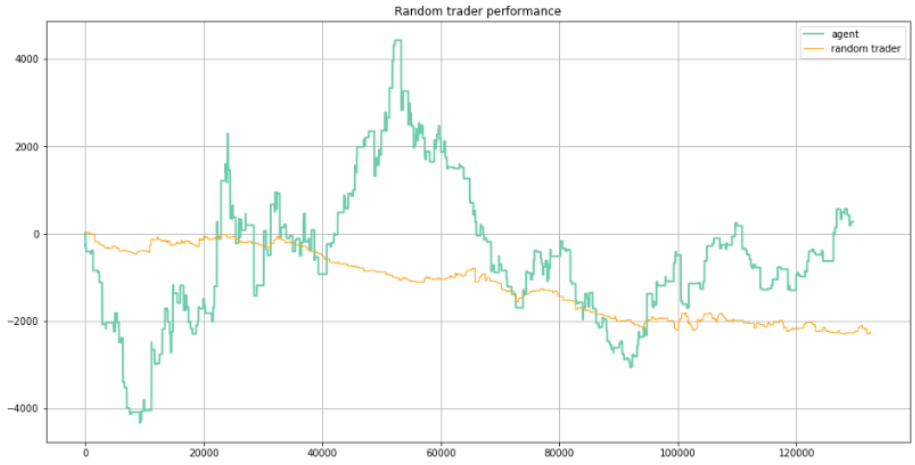
PnL	DQN Agent	Random trader
mean	-4,627.32	-1,100.47
standard deviation	2,647.94	985.34

Below we can find a sample random session comparing the performance of the benchmark and our agent. We can see that our agent is making profits for some time but the random agent smoothly looses money since the beginning.



The graph represent the numbers shows for the timeframes where the most of the trading sessions take place (2017-06-05 - 2017-09-05). A small number to trading sessions were run outside those time frames that suggest that those number are not that bad.

Here we show another time frame where the agent outperformed the random trader:



5 Conclusions

We have used the reinforcement learning framework to build and agent that learn how to trade Bitcoins. Also we have coded our own exchange simulated and run a series of *trading sessions* using historical prices since January 2017 to June 2018.

We tested the agent using 2 different strategies, the *look-forward* and *online*.

The agent, modeled with the Deep Q Learning algorithm, after tuning several hyper-parameters were not able to find a policy that consolidates a positive PnL.

We compared the results of our agent against a random trader to realize that ours perform worst in most of the cases, but were some sessions where the agent performed better than the random trader (see section above).

Our tests were severely limited by the amount of time it took to run (16 hours). Although we could only run the experiment once, we found that the *look-forward strategy* performed better that the *online strategy* one. The *look-forward strategy* sees more than 3 million samples and the *online strategy* only 130.000 for the same 3 month of testing period, so it seems like a confirmation about the *sample hungriness* of the reinforcement learning models [Irpan 2018].

6 Improvements

We added the 4 most recent OHLCV prices to the features to include time patterns in the algorithm, but it is quite naive to assume that just past price and volume have some sort of information about the future evolution of them. An approach worth exploring can be to use insights taken from the market *order book* as some authors have found evidences that characteristics from the order book are systematically related to price volatility and trading activity [Naes and Skjeltorp 2004].

The environment can be adapted to be *openai gym* compatible. This will allow to test advanced reinforcement models from third party frameworks like Google Dopamine[10].

[10] https://github.com/google/dopamine

References

Irpan, A. 2018. *Deep Reinforcement Learning Doesn't Work Yet.* https://www.alexirpan.com/2018/02/14/rl-hard.html.

Mnih, V., Kavukcuoglu, K., Silver, D., et al. 2015. Human-level control through deep reinforcement learning. *Nature 518*, 7540, 529–533.

Naes, R. and Skjeltorp, J.A. 2004. Order Book Characteristics and the Volume-Volatility Relation: Empirical Evidence from a Limit Order Market. *SSRN Electronic Journal*, 1–33.

Sutton, R.S., Barto, A.G., and others. 1998. *Reinforcement learning: An introduction.* MIT press.