Using Deep Q-Learning for Trading Cryptocurrencies

Experiments with Deep Q-learning to try to extract profits from an idealized market environment with real, historic prices.

Lars L. Ankile

NEURO 140 Biological and Artificial Intelligence Project Paper

Harvard College
Harvard University
Cambridge, MA

April 26, 2020

Using Deep Q-Learning for Trading Cryptocurrencies

Lars L. Ankile

April 26, 2020

Abstract

This paper applies Deep Q-Learning to the task of trading cryptocurrencies. Agents using Convolutional Neural Nets, of varying depths, as the Q-function approximator are trained on historic cryptocurrency data. The training produced high-frequency trading-agents that in some cases were able to beat the naïve buy-and-hold strategy. However, trading was conducted without commission and the results are not sufficiently consistent to conclude that the agents can beat the market consistently. More data, deeper networks, and more GPU-hours, might be required to more predictably exploit the markets.

1 Introduction

Trading is an activity that comprises buying and selling a security with the goal of extracting profits. Algorithmic trading is widespread, and can be based on a wide array of approaches. A major category is technical analysis, where patterns in historic price data are analyzed to try to predict future price movements. Deep Reinforcement Learning has proven effective at finding patterns and decision-making in many fields, and is therefore a natural candidate for an effective trading agent. The GitHub-project can be found here `https://github.com/ankile/reinforcement-trading`.

2 Theory

The classic case of machine learning, supervised learning, takes in a set of training examples with corresponding labels to train with. Training machine learning systems in this manner has proven very effective in many different domains. However, this kind of data isn't always available or practical to create. One solution is Reinforcement Learning since it is unsupervised and labelled training data is not needed. The idea is simple: let an agent interact with an environment and reward the agent when good outcomes occur and punish for actions that lead to bad outcomes. In this particular case, this means that we let the agent observe recent prices in the markets and let it take actions based on these observations. If these actions end up leading to the agent profiting, it gets rewarded, i.e. those specific actions are reinforced given the observations. If it did not profit, those actions are discouraged. The hope is that the agent will over time learn how to trade effectively in the markets by combining exploration of different actions in different states with the feedback on what is working and not.

Q-learning is a form of reinforcement learning that is based on a, so called, Q-function that maps each state and action into a real number representing the estimated *Quality* (a measure of expected reward) of the action given the state: $Q : S \times A \to \mathbb{R}$, where S and A is the sets of possible states and available actions, respectively.

In standard Q-learning, the Q-function is represented as a table indexed by s and a. Before learning starts, all entries are initialized to some value, often zeroes. Then, one will iteratively update the Q-values as the agent observes the environment and takes actions. At each time step, t, the agent will observe a state, s_t, take an action, a_t, receive a reward, r_t, before receiving a new state s_{t+1} in response to the action taken. Let α be the learning rate, i.e. how much weight the agent will place on new observations as opposed to the old Q-value. Let γ be the discount factor, i.e. how much less a future reward one time-step into the future is worth than reward right now. The update rule is given in equation (1)[1].

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_{a \in A} Q(s_{t+1}, a)) \tag{1}$$

The Q-function can be represented by a big table, indexed by s and a in small environments (e.g. games like pong or cart-pole), but in more complex environments (like the state-space of all possible price-movements) this becomes infeasible. This is where *Neural Nets* (NN) come in handy, as they can serve as function approximators for extremely complex functions.

When using NNs as a function approximator, the update rule becomes slightly different. In short, what you can do is to rearrange equation (1) slightly to produce a function that can be interpreted as a gradient update based on the difference between current Q-value and the new Q-value[1]. When doing this, one gets a function that can be differentiated, which allows for the use of *Gradient Descent* to update the network. This means that, given an expression for the difference between the current value and the desired value (as we have in the above described rearrangement), find the gradient of the expression by partial differentiation, and take a small step in the opposite direction of the gradient as to minimize the difference. The gradient will be a vector 'pointing' in the direction of steepest ascent, so the negation will 'point' in the direction that will decrease the difference between current and desired Q-value the fastest.

Given enough data, compute power, and time, the hope is that the agent, based on the above methods will be able to learn optimal strategies for extracting the maximum reward from the environment in which it finds itself in, without having any prior knowledge given to it.

This is obviously a very brief overview of the relevant theory, and barely scratches the surface of some of the complexities of the field. Still, I think it will suffice as context for the following experiments in trying to implement the above theory in code.

3 Method

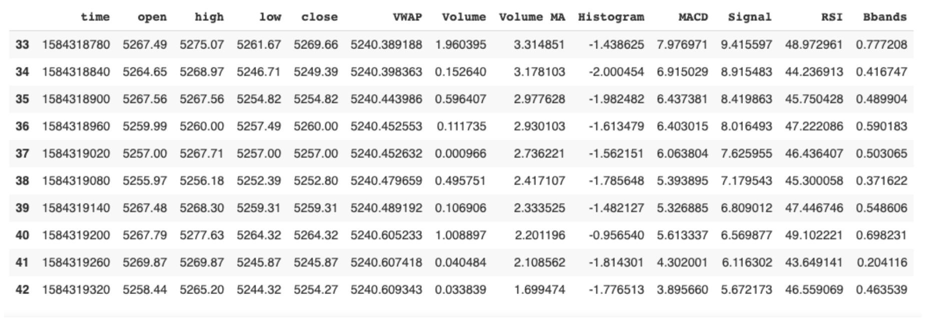
The following sections will briefly describe the major components to creating the results presented below.

3.1 Data Gathering and Preparation

For data, I collected as much as I could get of 1 minute data from `tradingview.com`. In total, I have 183 289 rows of crypto-data, spread out over 11 different currency pairs. 10 example rows can be seen in figure 1. The first column is the timestamp for the start time of the OHLC-bar (short for Open, High, Low, and Close, and is a way to represent the price movement during the one minute it is representing). In addition to the usual price data, there is also some columns with volume, volume-weighted average price (VWAP), and some common indicators, including MACD, RSI, Bollinger Bands. These are based on different ways of combining different lengths and types of moving averages, and will hopefully provide some more awareness of long-running trends for the agent.

I also have 262 108 rows of 1 minute OHLC-bars for the years 2015 and 2016 for Yandex stock (ticker YNDX). Here, I unfortunately do not have the indicators I have for the crypto-data, but can still be interesting to test with since it is a lot of data spanning a longer time frame than the other data I have.

To hopefully help the agent generalize better across time and currency pair, every feature that is provided to the agent is normalized in some way. For example, the columns for high, low, close, and VWAP are expressed as a percentage change from the open price, which itself is not included in the training data. Volume is expressed as a percentage change from the last bar. MACD is scaled down by the open price. RSI is divided by 100, because it lives in the interval $[0, 100]$. Bollinger bands are already expressed as a percentage. Now, all numbers are mostly contained within $[-1, 1]$, and does not depend on the price of the currency directly, but on its changes.



	time	open	high	low	close	VWAP	Volume	Volume MA	Histogram	MACD	Signal	RSI	Bbands
33	1584318780	5267.49	5275.07	5261.67	5269.66	5240.389188	1.960395	3.314851	-1.438625	7.976971	9.415597	48.972961	0.777208
34	1584318840	5264.65	5268.97	5246.71	5249.39	5240.398363	0.152640	3.178103	-2.000454	6.915029	8.915483	44.236913	0.416747
35	1584318900	5267.56	5267.56	5254.82	5254.82	5240.443986	0.596407	2.977628	-1.982482	6.437381	8.419863	45.750428	0.489904
36	1584318960	5259.99	5260.00	5257.49	5260.00	5240.452553	0.111735	2.930103	-1.613479	6.403015	8.016493	47.222086	0.590183
37	1584319020	5257.00	5267.71	5257.00	5257.00	5240.452632	0.000966	2.736221	-1.562151	6.063804	7.625955	46.436407	0.503065
38	1584319080	5255.97	5256.18	5252.39	5252.80	5240.479659	0.495751	2.417107	-1.785648	5.393895	7.179543	45.300058	0.371622
39	1584319140	5267.48	5268.30	5259.31	5259.31	5240.489192	0.106906	2.333525	-1.482127	5.326885	6.809012	47.446746	0.548606
40	1584319200	5267.79	5277.63	5264.32	5264.32	5240.605233	1.008897	2.201196	-0.956540	5.613337	6.569877	49.102221	0.698231
41	1584319260	5269.87	5269.87	5245.87	5245.87	5240.607418	0.040484	2.108562	-1.814301	4.302001	6.116302	43.649141	0.204116
42	1584319320	5258.44	5265.20	5244.32	5254.27	5240.609343	0.033839	1.699474	-1.776513	3.895660	5.672173	46.559069	0.463539

Figure 1: 10 example rows of raw cryptocurrency data.

3.2 Network Architectures

The main workhorse in the agent is a *Convolutional Neural Net* (CNN). I have tested three slightly different architectures. All uses some layers of 1-dimensional convolutions first, and then two or three layers of fully-connected layers to deliver the output, i.e. action to be taken.

The first network, referred to as small network 1, can be seen in listing 1. There, I am using two layers of convolutions before feeding the result to to the fully connected layers for mapping features to an action (that is why there are 3 out-features in the last layer, one for each of the possible actions, sell, buy, do-nothing). The other two networks, large network 1 and large network 2, are deeper, mainly in the number of convolutional layers. They also introduce some drop-out layers to try to mitigate over-fitting. The difference between large network 1 and 2 is that large network 1 has additional fully-connected layers, compared to large network 2, while large network 2 has additional convolutional layers compared to large network 1. These are not shown in the interest of saving space, but can be found in the code (`DQNConv1D`, `DQNConv1DLarge1`, `DQNConv1DLarge2` in `lib/models.py`).

Listing 1: Architecture of the Small Network

```
DQNConv1D(
  (conv): Sequential(
    (0): Conv1d(6, 128, kernel_size=(5,), stride=(1,))
    (1): ReLU()
    (2): Conv1d(128, 128, kernel_size=(5,), stride=(1,))
    (3): ReLU()
  )
  (fc_adv): Sequential(
    (0): Linear(in_features=5376, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=3, bias=True)
  )
)
```

3.3 Training Algorithm

Algorithm 1 Deep Q-Trading

```
Initialize neural network $\mathcal{N}$ and trading environment $\mathcal{E}$
Initialize $\epsilon_0 \leftarrow 1.0$, $t \leftarrow 0$
Initialize $s_t \leftarrow s_0$ get initial
while True do
  $t \leftarrow t + 1$
  $\epsilon \leftarrow \max(0.1, \epsilon_0 - t/1000000)$
  if $X \sim U(0,1) < \epsilon$ then
    action $\leftarrow$ choose action randomly
  else
    action $\leftarrow$ get action from network based on state $\mathcal{N}(s_t)$
  end if
  Take the chosen action and get reward and new state:
  $r_t$, $s_{t+1}$, Done $\leftarrow$ $\mathcal{E}$.step(action)
  if Done then
    break training loop
  end if
  Perform a step of gradient descent based on action taken, reward, and state
end while
```

In algorithm 1, the training algorithm used is outlined in a simplified version (full algorithm can be found in the code). In short, the code first initializes some variables used in training, e.g. the environment that the agent will interact with, the neural net that comprises the agent's brain, the epsilon start value (used in epsilon-greedy action selection), and the initial state. For every iteration of the loop, get current epsilon, and sample a random action to take if a random variable is smaller than epsilon (exploration), otherwise get the action from the network (exploitation). The action is taken in the environment, which gives back a new observation, a reward, and whether the episode is over. Based on the observation and reward, the network is updated by backpropagation. This cycle is repeated several million times.

3.4 Training and Different Experiments

There are in total 4 'experiments', or agents, I will test. None of the experiments will include commission on trades to keep it simpler. The first agent is the small network on the small feature set (only columns for high, low, close, volume), the second is large network 1 on the small feature set, the third is large network 1 on the large feature set, and the fourth agent is large network 1 on the large feature set. This is done so I can (more or less) directly compare the small and large network architectures (experiment 1 and 2) and also the small and large feature set (experiment two and three).

Since I have a year's worth of 1-minute bars for the YNDX stock for the small feature set, I initially train the first two agents on this data, and later the specific crypto-data. This will hopefully increase generalizability. The first two networks trained a little faster and was trained roughly 30 million iterations of the outer training loop, described in algorithm 1. Agent 3 and 4 was trained only on crypto-data, because the YNDX-data did not include the extra features the crypto data has. Those networks were trained for roughly 20 million iterations.

4 Results

In this section I will present results. There are 4 main experiments conducted. In figure 2 the results from running all networks on data never seen before, compared to the buy-and-hold (BNH) strategy at the bottom (which means to buy at the beginning of the period and sell everything at the end).

Each graph is the mean reward at each step over 10 runs of the network (there is some stochasticity in the agents). In the top left corner of each sub plot is info about the lowest drawdown, the highest reward, the standard deviation (a common measure of risk) and the total cumulative reward at the end of trading. The last plot shows the cumulative reward of the BNH strategy. The BNH strategy yielded a reward of 25% in this case (because ETHEUR happened to trend upwards in the data). The agents yielded 8.6%, 24%, 8.5%, and 1.8% respectively. The associated standard deviations were respectively 4.1, 6.3, 2.6, and 2.9 percentage points, compared to 8.4 percentage points for the BNH strategy.

In figure 3, the results from running the small network on a year of data for the YNDX 2015 data set is plotted. The top graph shows the cumulative reward, while the bottom graph shows the price of the stock, with green up-arrows at positions where the agent bought, and red down-arrows where the agent sold. The agent yielded roughly 25%, while the BNH strategy yielded roughly 7.8%.

5 Discussion

5.1 The General Findings

None of the agents were able to beat the BNH strategy in the experiments in figure 2, but all had significantly lower standard deviations, i.e. they varied less and could be considered less risky. All agents appear to be following the price fluctuations of the security pretty closely, too. The second agent (large network 1, small feature set) came pretty close to the yield

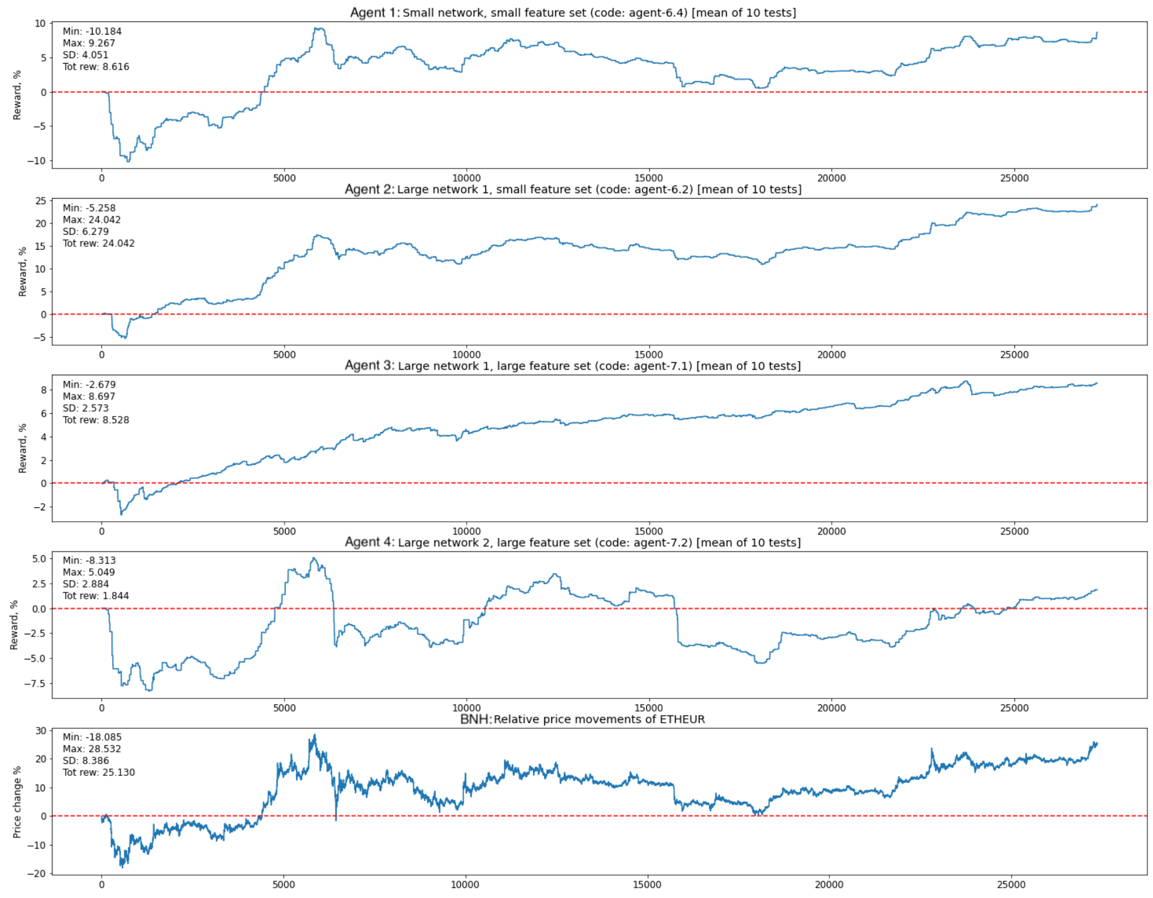


Figure 2: All four agents tested on held-out testing data pair ETHEUR, averaged over 10 tests.

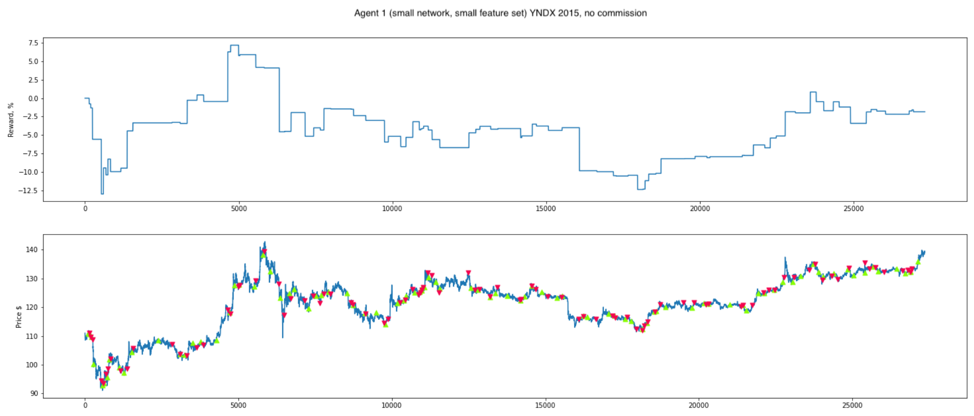


Figure 3: The small network, after 30 million training iterations, tested on YNDX data for 2015 with no commission.

of the BNH strategy, but at significantly lower SD, which, depending on a person's level of risk-aversion, could be desirable to the BNH strategy in this case. If this result would play out under other market conditions is unknown, however.

From figure 3, we can see that the agent makes a very high amount of trades. This will make a huge impact if we were to introduce commission and market slippage. Without commission, though, it beats the market in this case. Also here the reward follows closely to the price changes. However, we see that e.g. it is not falling between time step 50 000 and 80 000, as the price is, which is a good sign. It is also rising more sharply than the price from roughly 25 000 to 40 000.

5.2 Effect of Network Depth

The large network outperforms the small network in terms of biggest drawdown, highest high, and cumulative reward. This is at a higher level of variability, though. The network using the large feature set yielded much less than the same network using the small feature set. However, the yield had a much more non-decreasing monotone character to it (fewer periods losses) and much smaller SD, compared to the network using the small feature set. It is up to investors to decide what risk level seems reasonable in these cases, and one cannot conclude based on this what network is better, only that they display differing characteristics.

5.3 Effect of Number and Type of Features

From figure 2, we see that large network 1 outperforms large network 2 by a large margin. The difference is that large network 1 has an additional fully-connected (FC) layer before output than large network 2, while large network 2 has a couple of extra convolutional layers before the FC ones. The results seem to suggest that more convolutions does not improve performance, and that a deeper FC module at the output might be beneficial. One way to interpret this is that the features are extracted well enough by the convolutional layers to be made sense of by the deeper FC-module in large network 1, while the extra feature extraction large network 2 provides does not help because it is not able to make sense of it because of a more shallow FC module. This is just speculation, though.

6 Potential Issues and Next Steps

There are a couple of obvious issues with my attempts to beat the market. This area is extremely complex, both in terms of the purely financial and in terms of the purely computer science and machine learning related. Furthermore, it might be an impossible task if the efficient market hypothesis turns out to be correct (which is an open question). However, an honest attempt was made, progress was had, and a lot of new knowledge was acquired in the process. A couple of the most important things to do differently next time is discussed below.

6.1 Data Availability

The first, and probably most important one, is the amount of data I have available to train my models. I was able to gather 183 000 total rows of cryptocurrency data, which should be enough to do some early experimentation, but probably not enough to do any 'real' research on. These datapoints span around 4 weeks of time, i.e. one specific month of one specific year. To be able to (hopefully) generalize overarching patterns that could be traded on, that are valid more generally across time, I would need data that spans a lot larger time period, to capture dynamics present in the the different kinds of market conditions (high-/low uncertainty/volatility, bear/bull market, etc.).

To try to mitigate the problem of lack of data, I have started an effort to collect a lot more data than I currently have. This effort consists of a very simple server that constantly gets the latest data from an exchange, and saves it in files. This service will over time potentially be a very helpful resource for future experiments with cryptocurrency price data.

6.2 Data Correlation

Furthermore, my 183 000 data points are spread out across 11 different currency pairs, all for roughly the same time frame. Cryptocurrencies tend to be highly correlated, which results in a lot of the data being to some degree redundant or duplication. However, I still think the extra data has value because the dynamics are slightly different for each currency pair, which could help the network from overfitting to the specifics of one currency and rather pick up on more overarching patterns in the price data.

6.3 Lack of Experience and Expertise

In this project, there has been efforts to apply machine learning to the world of finance. I am very interested in both worlds, both personally and academically, and have been for a while. However, I am by no means an expert in either of the fields. I was confronted with an extremely steep learning curve in trying to tackle this project. I have conducted several experiments around tuning the model architecture, the feature space, and hyperparameters, but the search space is by no means exhausted. It is even uncertain whether DQN is the best suited tool for this task. Over the coming years, I will be studying these fields much more in-depth, and will certainly revisit this project at a later point in time with new knowledge, better ideas, and more data, to hopefully make some breakthroughs (in case breakthroughs are made, and the agent becomes wildly profitable, I have promised Mengmi to tell you guys and share my code – I'll be in touch).

7 Conclusion

Things tend to seem easier before one really starts digging into them. This time was no exception, and I must admit I probably suffered from the Dunning-Kruger effect to some degree before I started the project. Even though it was not as easy as hoped, the challenges the project posed has been an incredible learning experience, and I come out on the other side with a lot of valuable knowledge I would not have gotten if not for the hands-on experience from a project like this.

On the technical side, the network seemed to be performing reasonably well. It does not appear to be overfitted to the data. The larger network seems to outperform the smaller network. The larger feature set does not seem to outperform the smaller feature set, however.

In conclusion, this project will serve as a very rough initial test and learning platform for me to base future experiments into this realm upon in the future.

References

[1] The MIT6.036: *Introduction to Machine Learning staff. Lecture Notes: Chapter 11, Reinforcement Learning.* https://lms.mitx.mit.edu/assets/courseware/v1/ffb99647f23e8d02a087a224df2dfa6b/asset-v1:MITx+6.036+2020_Spring+type@asset+block/notes_chapter_Reinforcement_learning.pdf

[2] Wikipedia. *Q-learning.* https://en.wikipedia.org/wiki/Q-learning