TARAS SHEVCHENKO NATIONAL UNIVERSITY OF KYIV
Faculty of Computer Science and Cybernetics
Department of Theory and Technology of Programming

Qualification work

to obtain a Bachelor's degree

in the educational and professional program "Artificial Intelligence"
specialty 122 "Computer Science" on the topic:

**INTELLECTUAL TRADING STRATEGY
USING REINFORCEMENT LEARNING**

Prepared by a 4th year student
Vynnyk Dmytro Oleksandrovych ___________
(signature)

Supervisor:
Associate professor, PhD
Panchenko Taras Volodymyrovych ___________
(signature)

I certify that in this work there are no borrowings from the works of other authors without appropriate references.

Student ___________
(signature)

The work was reviewed and accepted for the defense at the meeting of the Department of Theory and Technology of Programming
"_____"_______________ 2021,
protocol No. _____
Head of Department
Nikitchenko M.S. ___________
(signature)

Kyiv – 2021

CONTENTS

ABSTRACT

The bachelor's thesis consists of an introduction, four chapters, conclusions, a list of used sources (28 items), and 1 appendix. The work contains 15 figures and 5 tables. The total volume of the work is 32 pages, the main text of the work is laid out on 20 pages.

REINFORCEMENT LEARNING, ENVIRONMENT, FUTURES, STOCK INDICATORS, FRACTIONAL DIFFERENTIATION, Q-LEARNING, DQN.

The object of the research is the stock market, in particular cryptocurrency exchanges. The subject of research is algorithmic trading based on internal and external indicators.

The objective of the work is to test the hypothesis about the profitability of the RL strategy in the presence of a wide range of data, over a long time interval, taking into account commission, rounding, and price shifts.

Development methods: artificial neural networks, reinforcement learning, fractional differentiation, stationarity, and unit root tests. Development tools: PyCharm development environment, Python 3 programming language.

Work results: analyzed data available in analytical services, selected useful indicators, downloaded data set, implemented and investigated fast fractional differentiation, standardized the data, defined the neural network architecture and hyperparameters, developed an environment for training agents, tested the convergence and profitability of the agent before and after the introduction of constraints.

ABBREVIATIONS

RL	–	Reinforcement Learning
ADF-GLS	–	Augmented Dickey-Fuller test with Generalized Least Squares
KPSS	–	Kwiatkowski–Phillips–Schmidt–Shin
ReLU	–	Rectified linear unit
AR(F)IMA	–	Autoregressive (fractionally) integrated moving average
VE	–	Value error
(I) FFT	–	(Inverse) fast Fourier transform
API	–	Application programming interface

INTRODUCTION

Assessment of the current state of the development object

RL has existed since the end of the last century, therefore there are quite a lot of works related to stock trading. In contrast, cryptocurrency trading gained popularity in the mid-2010s, leading to fewer works with varying approaches, data, restrictions, or even goals. Some works focus on trading a single asset, while others aim to build a portfolio of assets.

Relevance of the work and reasons for its implementation

Cryptocurrency exchanges have opened up opportunities for speculation to the general public. No intermediaries (brokers) are needed to work with them, as trades can be made in fractional amounts and exchanges automatically provide loan money for short positions. A substantial portion of assets on exchanges is highly liquid, enabling frequent order submissions that are highly likely to be fulfilled. As a result, both long-term trading and high-frequency trading are possible.

Decisions must be made for trading. This can be done randomly, or by "figures" on the graph, or by the position of the stars in the sky. The key factor is whether it leads to profitability. Practical experience demonstrates that various internal and external indicators can be effectively utilized for decision-making. For instance, trading volume can be classified as an internal indicator, while the number of coins mined by miners can be considered an external indicator.

In fact, the data is a multidimensional time series. This may encourage the use of forecasting techniques, but this intermediate step between data and decision degrades the accuracy of the model. In this situation, it is more appropriate to determine the performed action directly from the data.

The main obstacle to using supervised learning, i.e. classification and regression, is the labeling of the data. Without labels, it is impossible to define the objective function: we could build a decision tree – a classifier that matches states with optimal actions, but to do so, we need to know or be able to determine which actions are optimal. Unsupervised learning, which extracts patterns (clusters, components, maps) from the data, is also not applicable because it is not clear how to transform information about patterns with unknown properties into optimal action. This is where reinforcement learning (RL) becomes useful.

RL incorporates future rewards when evaluating a state or a state-action pair. Additionally, through Temporal Difference Learning (TD), it is possible to learn after each decision. Hence, it is crucial to examine the capabilities of RL promptly.

Objective and tasks of the work

The objective of the work is to test the hypothesis about the profitability of the RL strategy in the presence of a wide range of data, over a long time interval, taking into account commission, rounding, and price shifts. To achieve this goal, the following tasks have been set:

- analyze data available in analytical services
- select useful indicators
- load data
- standardize data
- define neural network architecture and hyperparameters
- develop an environment for training agents
- test the convergence and profitability of the agent before and after the introduction of constraints

Object, methods, and means of research and development

The stock market, as an object, is mimicked by the developed OpenAI gym [1] environment for training agents. DQN [2] [3] acts as an agent. Software implementation of data downloading and processing, agent and environment is done in Python 3 using such libraries as: TensorFlow [4], Keras [5], keras-rl [6], gym [1], NumPy [7], SciPy [8], arch [9], scikit-learn [10], pandas [11], matplotlib [12].

SECTION 1. THEORETICAL BASIS OF REINFORCEMENT LEARNING [13]

1.1. The *k*-armed "Bandit" problem

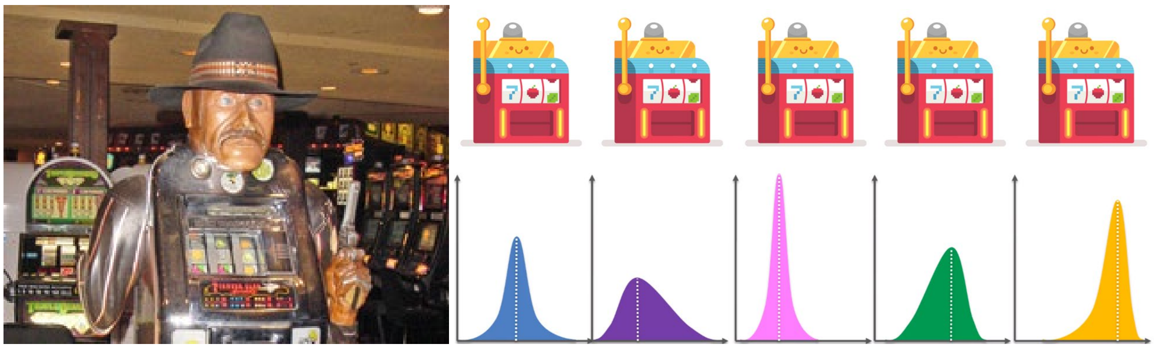

Figure 1 – Illustration of the problem

"Bandit" is a slot machine in the form of a bandit. Players insert a coin into the machine's slot and pull the hand, resulting in a reward.

The *k*-armed "Bandit" problem is a mathematical representation of the real-world reward maximization problem. In this problem, there are *k* machines, each with its own fixed reward distribution. At each step, the challenge is to select a machine without prior knowledge of its distribution.

The profitability of the action is estimated by the following tabular method:

A_t — (action) action in the moment t, scalar

R_t — (reward) reward in the moment t, scalar

$Q_t(a)$ — value function

$q_*(a) = \mathbb{E}[R_t | A_t = a]$ — true value

$$Q_t(a) \doteq \frac{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a} \cdot R_i}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}} = \frac{\text{sum of reward when action } a \text{ selected prior moment } t}{\text{sum of times when action } a \text{ selected prior moment } t}$$

Table 1 – Example of $Q_t(x)$ calculation when $A = \{x, y\}$

$A_1 = x,\ R_1 = 2$	$A_2 = y,\ R_2 = 0$	$A_3 = x,\ R_3 = 4$	$A_4 = y,\ R_4 = 1$	…
$Q_1(x) = 0$	$Q_2(x) = \frac{2}{1} = 2$	$Q_3(x) = Q_2(x) = 2$	$Q_4(x) = \frac{2+4}{2} = 3$	…

Then, knowing the value of various actions, it remained to choose the action with the greatest value. This approach is called greedy:

$$A_t = \underset{a}{\operatorname{argmax}}\, Q_t(a)$$

But it lacks exploration, that is why we use ε-greedy selection:

$$A_t \overset{\text{def}}{=} \begin{cases} \underset{a}{\operatorname{argmax}}\, Q_t(a)\,, & \text{with probability } 1 - \varepsilon \\ \text{random action,} & \text{with probability } \varepsilon \end{cases}$$

Essentially, we determine the average reward for each action. We can calculate the average iteratively:

$$Q_{n+1} = \bar{R}_n = \frac{R_1 + R_2 + \cdots + R_n}{n} = Q_n + \frac{1}{n}(R_n - Q_n)$$

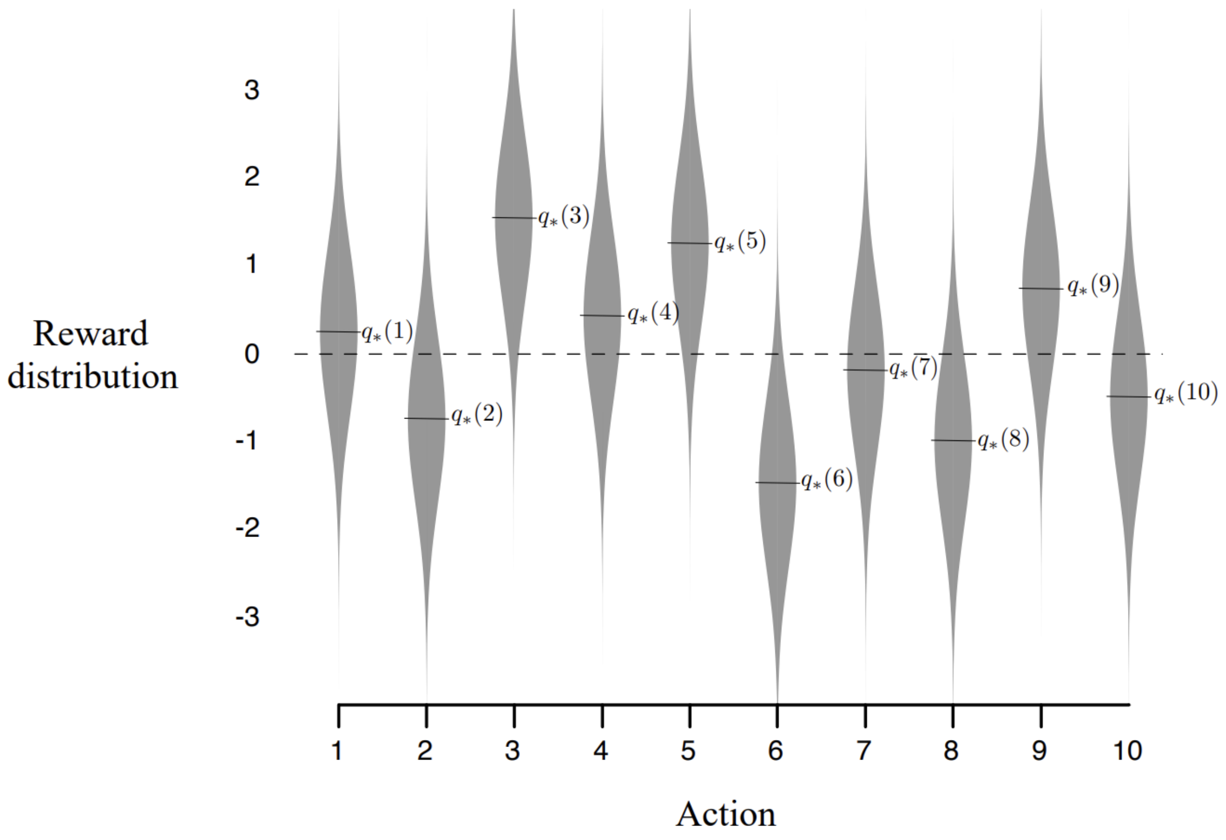

Figure 2 – Example of "10-armed Bandit" [1]. The third action is optimal

[1] The mean coincides with the peak value of the density, which is generally true only for unimodal symmetric distributions.

1.2. Non-stationary problems

Problems in which the value of actions changes over time are called non-stationary. In such tasks, the value function should gradually reduce the influence of old rewards. To achieve this, the multiplier before the brackets is replaced by a constant value $\alpha \in (0, 1)$:

$$Q_{n+1} := Q_n + \frac{1}{n}(R_n - Q_n) \rightarrow Q_{n+1} := Q_n + \alpha(R_n - Q_n)$$

In fact, it is an exponential moving average.

1.3. Problems with states

$S - (state)$ state, arithmetic vector

$\pi_t(a) = \pi(a|S_t) -$ policy, the prob. of selecting action a in state s

$Q_t = Q(S_t, A_t) -$ value of action

$V_t = V(S_t) = \pi_t Q_t -$ value of state

$S_0,\ A_0,\ R_1,\ \ldots,\ S_{T-1}, A_{T-1}, R_T -$ trajectory

In problems involving states, every action is accompanied by a reward and a transition to a new state . As a result, our understanding of expected income changes, as it should include not only immediate rewards but also future rewards. To address this, the reward R_t is replaced by the aggregate reward $G_t = R_{t+1} + R_{t+2} + \cdots$, which recursively appears as $G_t = R_{t+1} + G_{t+1}$. Additionally, a discount factor $\gamma \in (0,1)$ is introduced to diminish the impact of future rewards: $G_t = R_{t+1} + \gamma G_{t+1}$.

$$V_t := V_t + \alpha(R_{t+1} + V_{t+1} - V_t)$$

These methods are known as Monte Carlo methods. They have a drawback: a predefined trajectory is required to calculate G_t, starting from the end of the trajectory ($G_{T-1} = R_T, G_{T-2} = R_{T-1} + \gamma G_{T-1}, \ldots$).

Temporal Difference Learning (TD) is significantly more effective. Instead of relying on the aggregate reward G_{t+1}, TD leverages an approximation V_{t+1}. This enables the value function to be updated at each step.

1.4. Methods of the approximate solution

So far, we have focused on tabular methods, which are suitable for tasks with small state spaces. However, in scenarios where states are represented by large vectors, tabular methods are not applicable. Instead, approximators such as regression models, artificial neural networks, or even functional series are used.

Let us consider the transition from tabular methods to neural networks. First of all, an error is introduced:

$$\overline{\mathrm{VE}}(\mathbf{w}) \doteq \sum_{s\in\mathcal{S}} \mu(s)\big[V(s) - \hat{V}(s,\mathbf{w})\big]^2,\ \mu - \text{state distibution}$$

Then the gradient descent looks like this:

$$\mathbf{w}_{t+1} := \mathbf{w}_t - \frac{1}{2}\alpha\nabla\big[V(S_t) - \hat{V}(S_t,\mathbf{w}_\mathrm{t})\big]^2$$

$$= \mathbf{w}_t + \alpha\big[V(S_t) - \hat{V}(S_t,\mathbf{w}_\mathrm{t})\big]\nabla\hat{V}(S_t,\mathbf{w}_\mathrm{t})$$

$$= \mathbf{w}_t + \alpha\big[U_t - \hat{V}(S_t,\mathbf{w}_\mathrm{t})\big]\nabla\hat{V}(S_t,\mathbf{w}_\mathrm{t})$$

Where U_t (U – unbiased) is an unbiased estimate of $V(S_t)$ ($\mathbb{E}[U_t|S_t = s] = V(s)$). For example, in Monte Carlo methods, U_t is represented by G_t – the expected value of the aggregate reward, which is unbiased by definition. However, when utilizing TD, certain concessions are made by employing $\hat{G}_t = R_{t+1} + \gamma\hat{V}(S_{t+1},\ \mathbf{w}_\mathbf{t})$, which depends on $\mathbf{w}_\mathbf{t}$. This makes the estimate biased, and results in the methods being semi-gradient.

One of the varieties of TD algorithms is Q-learning. This is the TD algorithm, where $\pi(a|S_{t+1})$ is a greedy strategy – the same as $V(S_{t+1}) = \max_a Q(S_{t+1}, a)$.

Finally, gradient descent for Q-learning looks like this:

$$\mathbf{w}_{t+1} := \mathbf{w}_t + \alpha\left[R_{t+1} + \gamma\max_a \hat{Q}(S_{t+1}, a,\ \mathbf{w}_\mathbf{t}) - \hat{Q}(S_t, A_t, \mathbf{w}_\mathrm{t})\right]\nabla\hat{Q}(S_t, A_t, \mathbf{w}_\mathrm{t})$$

SECTION 2. DATA

2.1. Data description

Let us characterize the data used before each decision:

The first dimension of the data corresponds to moments of time and has a constant step. Exchanges and analytics services that offer data via API typically offer data with a step ranging from minute to day. Obviously, as the step increases, the random component increases. Therefore, to minimize uncertainty, the smallest available steps (e.g. 1 min) should be used.

The second dimension of the data contains indicators. They are often neglected, using only price. Therefore, 190 indicators of various cryptocurrency exchanges were downloaded from analytical services such as Bybt: input/output volumes, reserves, number of transfers, etc.

Among the indicators available for download, those that are linear combinations of others were removed, while products and fractions, on the contrary, were left. This selection is based on the nature of neural networks, which use linear combinations of features and cannot approximate the product without a sufficiently large number of layers.

Since different indicators have different time intervals, they were resampled to the minute level using forward filling. Consequently, a data set (a single table) with a volume of 10 GB and 8 million rows was obtained.

Table 2 – Example of forward filling

datetime	reserve_usd_bitmex	reserve_usd_bitmex_ffill
2014-09-23 07:00:00+00:00	2.01281	2.01281
2014-09-23 07:01:00+00:00	nan	2.01281
…	…	…
2014-09-23 07:59:00+00:00	nan	2.01281
2014-09-23 08:00:00+00:00	1.16098	1.16098
2014-09-23 08:01:00+00:00	nan	1.16098
…	…	…

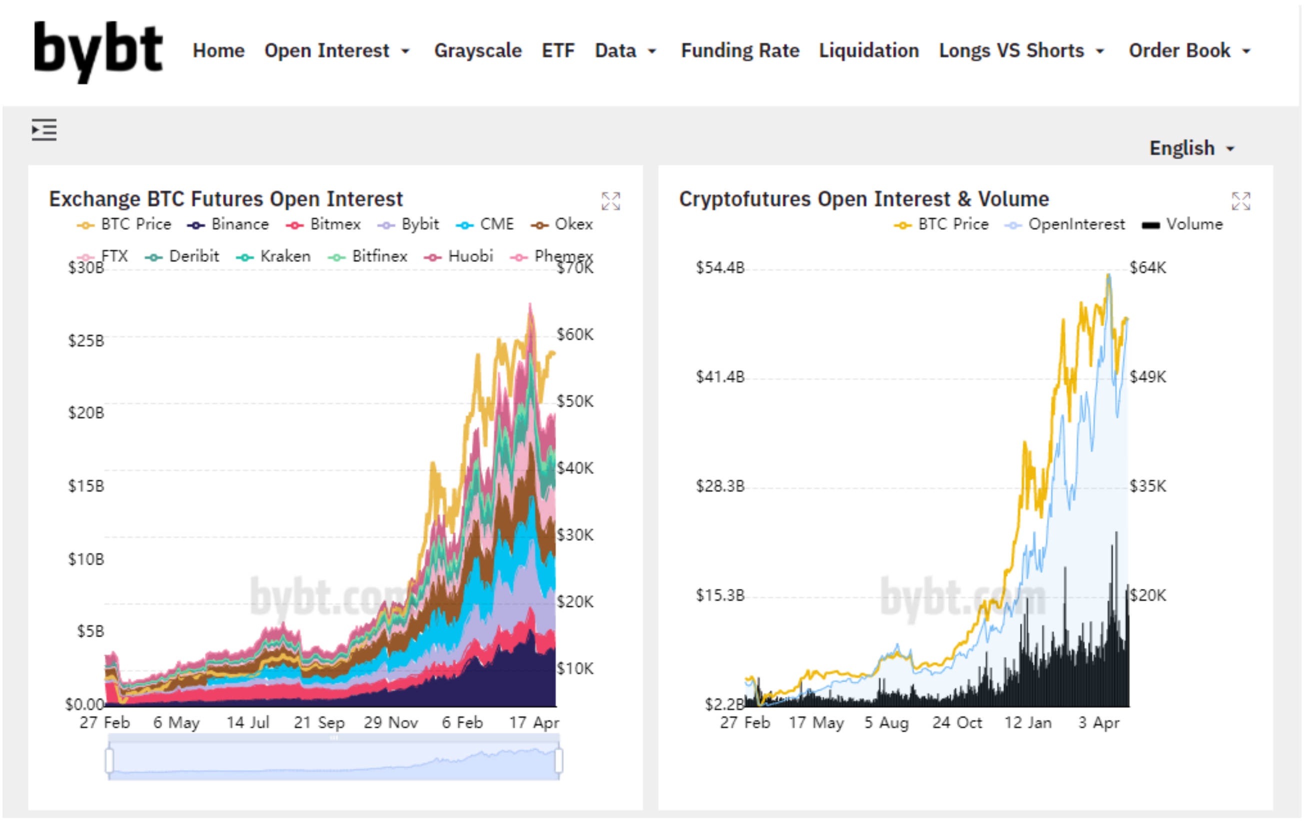

Figure 3 – Futures information platform Bybt

Together with the indicators, data from the exchange was downloaded: price, data for calculating the liquidation price, separate buy/sell volumes, which are not provided by the exchange and require local calculation from the buy/sell data sets.

2.2. Data preparation

2.2.1. Fractional differentiation

To prevent distribution shift between real data and training/validation data, and to improve predictions in general, it is necessary to ensure stationarity of the time series. Stationarity can be achieved through differentiation, which involves subtracting the t'th element from the (t-1)'th element. However, this method has the drawback of losing valuable information. Therefore, it is recommended to use fractional differentiation [14]. This approach uses the expansion of the lag operator L ($Lx_t = x_{t-1}$) into binomial series (a power series with binomial coefficients), enabling differentiation by a fractional order $d \in (0,1)$:

$$\nabla^d x_t \overset{\text{def}}{=} (1-L)^d, \text{ then}$$

$$\nabla^d x_t = \sum_{k=0}^{\infty} \binom{d}{k} (-L)^k = 1 - dL - \frac{1}{2} d(1-d)L^2 + \dots \approx$$

$$\approx \sum_{k=0}^{|x|} w_k x_{t-k}\,, \text{where } w_0 = 1, w_k = -w_{k-1} \frac{d-k+1}{k}$$

2.2.2. Convolutions

The last sum is the convolution of two arrays. If we calculate it as a sum, it will take about two minutes per column for the obtained data set. This is unacceptably high, so convolution was used with expansion into a Fourier series [8] [15] (in the future, this will reduce the processing time from two months to seven hours).

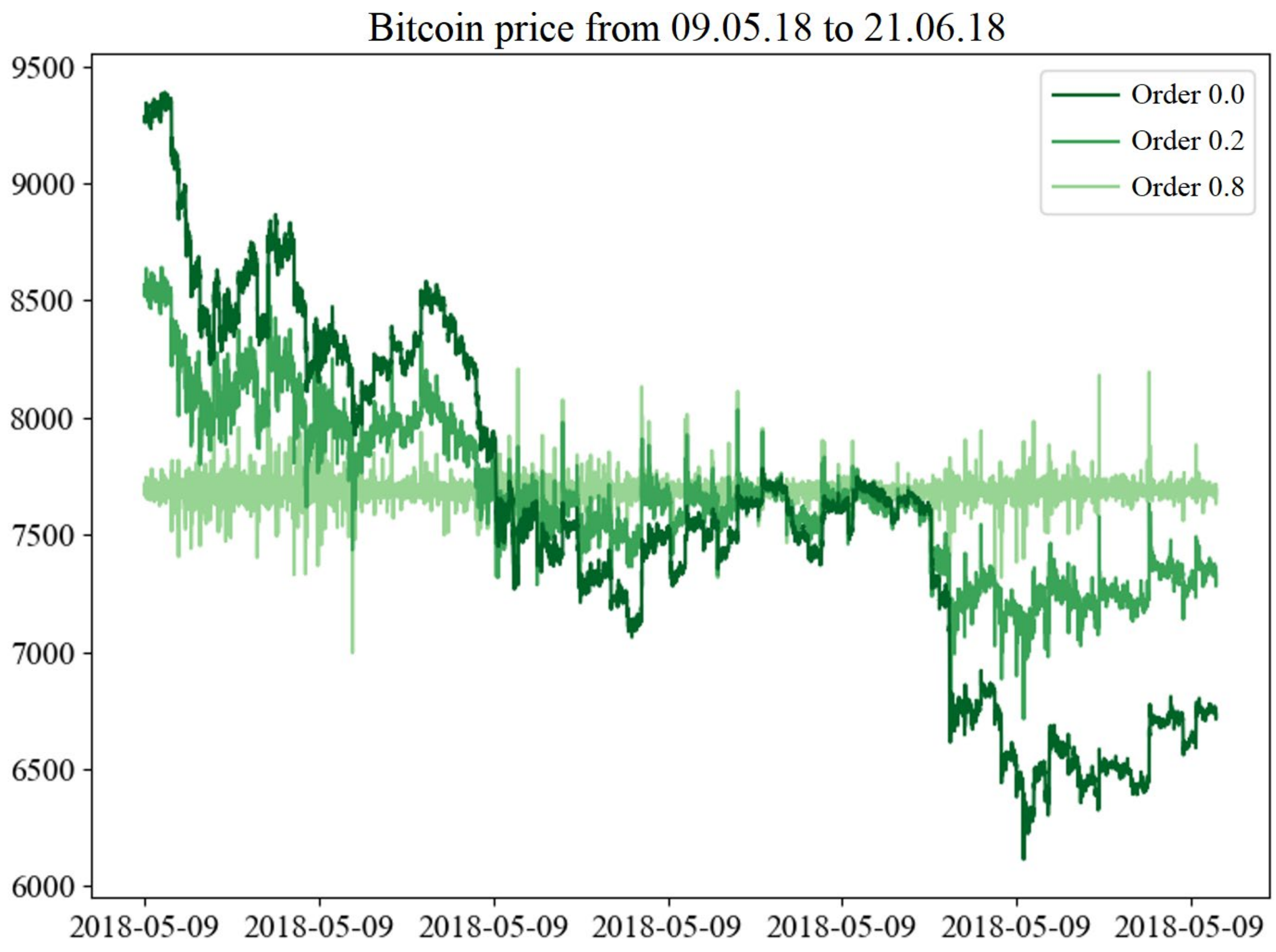

Figure 4 – Example of fractional differentiation

The convolution with FFT looks quite simple:

```
ifft(fft(w) * fft(x))
```

2.2.3. Minimum order of differentiation

The smaller the order of differentiation, the more useful information is preserved in the data. To determine which order leads to stationarity and which does not, we can use the bisection method, or even better, the more powerful method of Brent [16], to find the minimum order.

The GLS test was used to check for stationarity [17]. It is parametric and suffers from false positives, so it is often used in tandem with KPSS [18], which is a nonparametric method but suffers from false negatives. Therefore, none of them can give an unequivocal answer about stationarity.

An additional restriction was imposed on boundaries in which the order of differentiation was searched, which is not noticed in ready-made libraries (fracdiff, numpy-fracdiff, fractional-differentiation-time-series). In fact, we cannot use the entire interval $(0,1)$ because we replaced the infinite series that should converge to 0 with a finite series that converges to positive ε, which increases as the order approaches 0 (Fig. 5). Because of this, an excess positive shift may occur with unknown consequences. Therefore, it is necessary to determine the minimum boundary for the search for order. This can be done by introducing ε_{max}, for example, by three-sigma rule:

```
e_max = 0.0027  # 1 - 3 sigma
min_order = scipy.optimize.brentq(
    lambda order:
        np.cumsum(get_weights(order, weights_num=len(df)))[-1] - e_max
    a=0, b=1,
)
```

Code 1 – Finding the minimum boundary of the unshifted order using the Brent optimization method from the SciPy library [8]

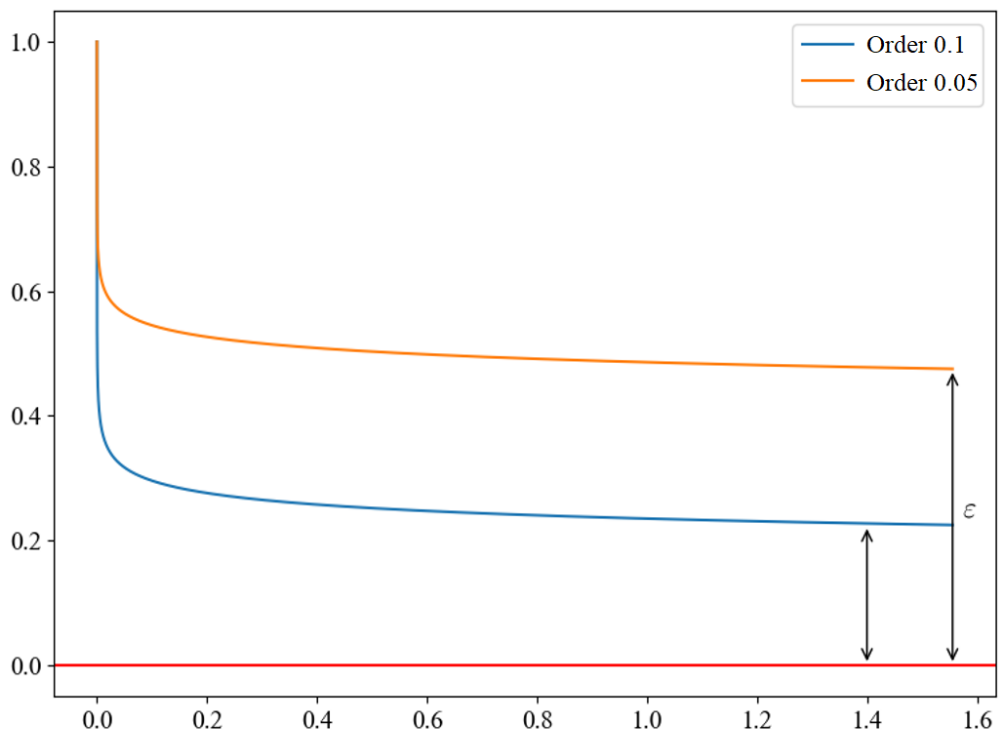

Figure 5 – Cumulative sum of fractional differentiation weights

During the tests, models with a constant regression model were used, because the shift, which is later removed during standardization, is present in all time series. To obtain comparable results, the search for a sufficient order of differentiation was conducted with a constant number of lag components. Because in the arch econometric package [9], KPSS selects the number of lags depending on the data values. This can lead to unreasonably large numbers of lags. Therefore, it is chosen as the minimum non-zero value at zero and one orders of differentiation.

```
def min_not_0(a, b):
    return 0 if a == b == 0
             else (max(a, b) if a == 0 or b == 0
                             else min(a, b))

lags_kpss = min_not_0(
    arch.KPSS(column, trend='c').lags,              # order 0
    arch.KPSS(numpy.diff(column), trend='c').lags   # order 1
)
diff_order_kpss = scipy.optimize.brentq(
    lambda order: arch.KPSS(
        lib.frac_diff_fft(column, order=order),
        trend='c', lags=lags_kpss
    ).pvalue - 0.0027,
    a=min_order, b=1
)
```

Code 2 – Search for the order of differentiation

Due to the reversed alternative hypothesis and the low threshold significance, the result obtained with KPSS should be considered necessary, and with ADF-GLS – sufficient. However, there are often cases where ADF-GLS asserts weak stationarity of the series without any differentiation, while KPSS requires strong differentiation. This may be due to heteroskedasticity or structural breaks in the data. An example of a discrepancy in test results can be seen in Table. 3. We cannot exclude these columns (116/141) due to the significant loss of the amount of information, we cannot differentiate either. Therefore, it is advisable to differentiate by the larger of the two obtained orders.

Table 3 – Test results for variance ratio, unit root, and stationarity

Indicator	KPSS Lags	KPSS Order	DFGLS Lags	DFGLS Order
transactions_count_outflow_bitmex	650	1	0	0
mpi	684	1	0	0
exchange_whale_ratio_spot_exchange	654	0.713	0	0
fund_flow_ratio_all_exchange	684	0.561	0	0
close	210	1	134	1
volume_buy	589	1	0	0
stablecoin_supply_ratio	685	0.81 5	0	1
thermo_cap	54	1	0	0.637
…				

Together with the unit root and stationarity tests, the variance-ratio test [19] was conducted. It checks whether the time series is a random walk, and its statistic allows us to characterize the unit root.

Table 4 – Results of the variance ratio test

Indicator	Variance ratio	Significance
close	1.024	0
volume_sell	0.643	0
reserve_usd_derivative_exchange	0.999	0.472129
reserve_usd_bitmex	1	0.785474
inflow_top10_all_exchange	1	0
…		

A statistic close to 1 corresponds to the typical case of a unit root. A statistic between 0 and 1 indicates that after the perturbation, the magnitude value is between the initial and peak values. A statistic greater than 1 indicates that after the perturbation, the value of the quantity becomes even larger.

According to this test, most of the indicators (137/141) have a typical unit root, also most are not random wanderings (121/141).

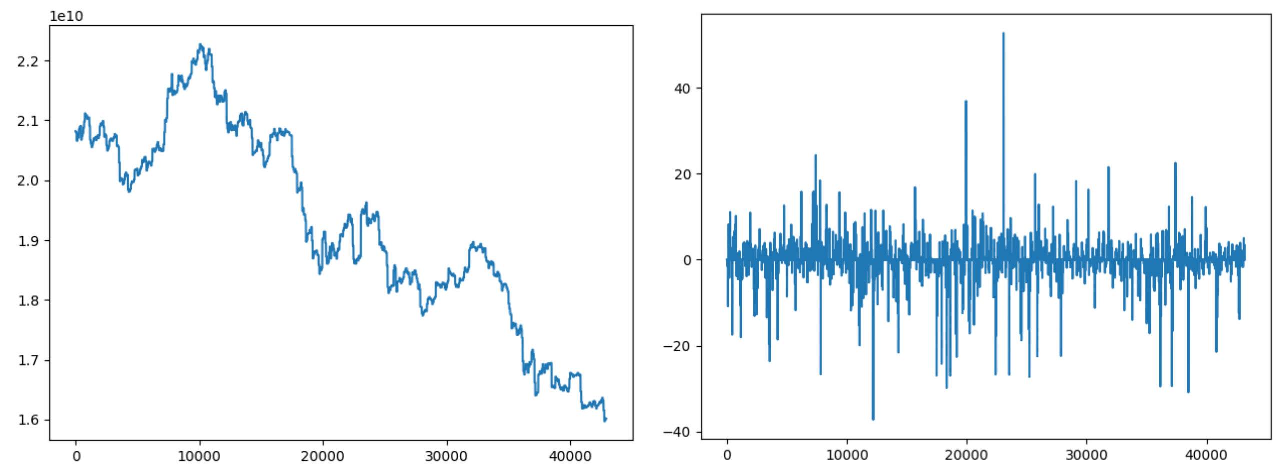

Figure 6 – Data of aggregate reserve of exchanges before/after preprocessing. The left time series is non-stationary and has a trend, the right one is stationary

After stationarization, the time series were standardized, for faster learning and smaller neuron weights.

SECTION 3. ENVIRONMENT AND AGENT

3.1. Architecture of the neural network

Unlike computer vision problems, where each pixel alone does not carry useful information, we have data in which each indicator can potentially be significant. It is also known that wide neural networks have good memory abilities. Therefore, we should have a sufficiently wide neural network, for example, as wide as the number of inputs (or proportionally).

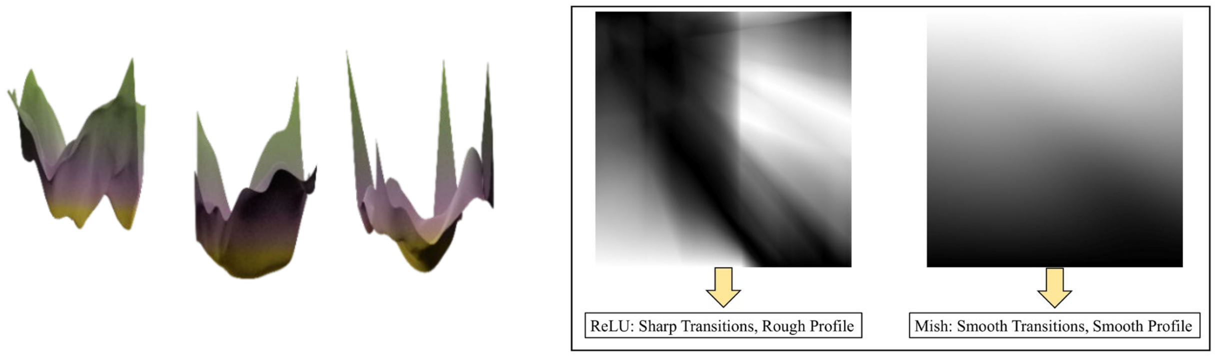
ReLU: Sharp Transitions, Rough Profile
Mish: Smooth Transitions, Smooth Profile

Figure 7 – Comparison of output surfaces [20]
1: ReLU/Swish/Mish, 2: ReLU/Mish

Mish was used as the activation function [20]. It gives a smoother surface of the loss function. ConcreteDropout [21] was used for regularization, which has an adaptive rate of neuron shutdown.

A Chi uniform distribution was used [22] to initialize layers with Mish, and a Glorot uniform distribution was used for the last layer with linear activation [23]. It has been established that these distributions prevent gradient vanishing [22] [23].

3.2. Implementation of the environment

There are many ready-made environments: in the Tensortrade library, in the FinRL library, gym-anytrading, TradingGym, trading-gym. But none of them are at the same time: tested for edge cases, and fast enough, and such that includes a commission and takes into account the deviation of the closing price from the possible execution price. Therefore, it was decided to develop our own environment based on OpenAI Gym.

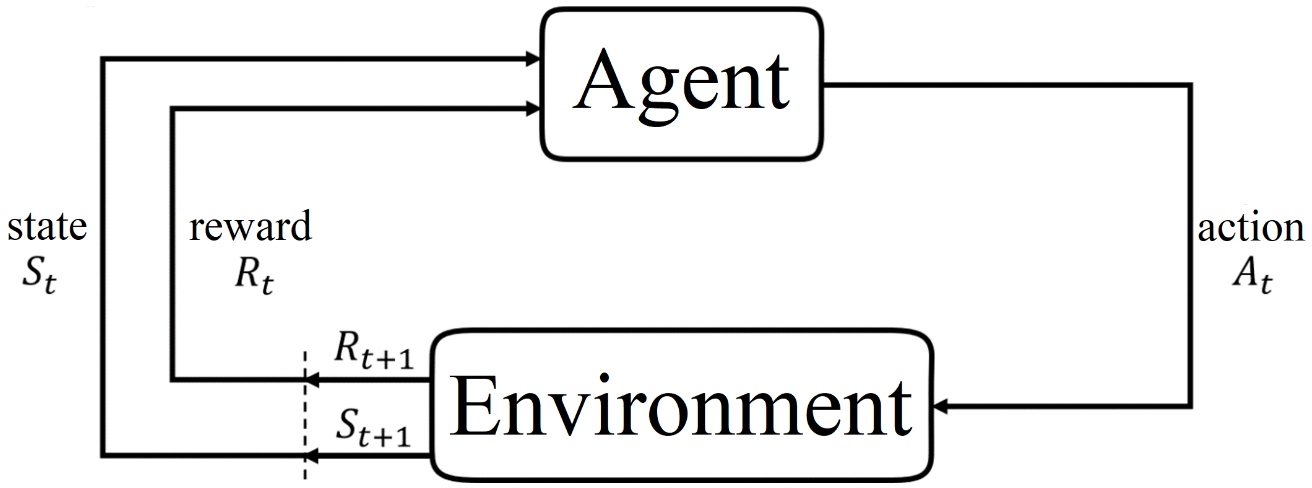

Figure 8 – A typical diagram of the agent's interaction with the environment

The created environment completely copies the behavior of the Bitmex exchange, all roundings and commissions:

```python
def value_diffs(self) -> np.ndarray:
    if self.buys:
        enter_values = np.fromiter(
            (lib.contract_to_xbt(
                # buy correction:
                price=buy_price + self.corr_buy,
                multiplier=-self.instrument.multiplier
            ) for buy_price in self.buys),
            # rounding:
            dtype=np.int,
        )

        one_buy_exit_value = lib.contract_to_xbt(
            # sell correction:
            price=self.close[self.t] - self.corr_sell,
            multiplier=-self.instrument.multiplier,
        )

        # comissions:
        return (
            (1 - self.position.commission) * enter_values -
            (1 + self.position.commission) * one_buy_exit_value
        )
    else:
        return np.array([])
```

Code 3 – Calculation of differences in the volume of buys and sells

It is also highly productive due to:

1) using structured arrays numpy.recarray, instead of typical for data analysis, pandas.DataFrame tables, which slow down calculations due to a large number of Python abstractions
2) using lazy generators
3) using view instead of array copies
4) caching, for example when calculating the number of satoshis (1e-8 bitcoin) in one dollar:

```
@lru_cache(maxsize=None)
def contract_to_xbt(price, multiplier) -> int:
    return round(multiplier / price)
```

3.2.1. Observations

At each step, the environment returns indicator data along with a tuple characterizing the state of the position: (*value_diffs_mean*, *buys_age* / *max_buys_age*, *buys_age* > *max_buys_age*). The first value is calculated as the average of elementwise subtraction of buy volumes from the volume that will be released if the selling takes place at the current price. The second and third values show the agent how long he can stay in the position, otherwise, it would be an infinite sum game.

The way we standardize the position state is crucial. Because if we bring it to the wrong scale, then at the beginning the neural network will perceive it as more or less important than the given indicators. The last two values are very easy to bring to the segment [-1, 1] because their limits are known:

$$((buys\_age \,/\, max\_buys\_age) - 0.5) * 2$$

$$1 \text{ if } buys\_age > max\_buys\_age \text{ else } -1$$

The shift and scale of the first value are calculated as the average and standard deviation of the differences within a window that moves in time and covers a time interval as large as the maximum age of the position. For example, if *max_buys_age* = 2, then there will be two differences in each window:

values = [1, 2, 3, 4, 5]

value_windows = [[1, 2, 3], [2, 3, 4], [3, 4, 5]]

window_diffs = [[3 - 13 - 2], [4 - 2, 4 - 3], [5 - 3, 5 - 4]] =

= [[2, 1], [2, 1], [2, 1]]

powers_means = mean(*value_diffs*) = [2, 1]

value_diff_mean = mean(*powers*) = 1.5

value_diff_std = std(*diff_power_means* - *value_diff_mean*) =

= std([2 - 1.5, 1 - 1.5]) = std([0.5, -0.5]) = 0.5

3.2.2. Reward function

The simplest reward was used: the difference in buy/sell volumes.

3.3. Agent

There are many types of agents, the main list along with the description is given in the table:

Table 5 – Main agents. D/C – discrete/continuous spaces

Algorithm	Description	Strategy	Action space	State space	Operator
Q-learning	State–action–reward–state	Off	D	D	Q-value
SARS	State–action–reward–state–action	On	D	D	Q-value
DQN	Deep Q Network	Off	D	C	Q-value
DDPG	Deep Deterministic Policy Gradient	Off	C	C	Q-value
A2(3)C	(Asynchronous) Advantage Actor-Critic	On	C	C	Preference function

NAF	Q-Learning with Normalized Advantage Functions	Off	C	C	Preference function
TRPO	Trust Region Policy Optimization	On	C	C	Preference function
PPO	Proximal Policy Optimization	On	C	C	Preference function
TD3	Twin Delayed DDPG	Off	C	C	Q-value
SAC	Soft Actor-Critic	Off	C	C	Preference function

In this work, experiments were carried out with DQN [2] [3]. This is Q-learning enhanced by using a replay buffer that stores a transition tuple after each step (S, A, R, S'). Training with a replay buffer is not conducted from the last received tuple but from a set of random ones contained in the buffer. This facilitates greater generalization and makes the algorithm more efficient relative to the number of observations.

3.4. Limitation of hyperparameters

ε-greedy strategy with probability ε closes the position, which is equivalent to starting the environment not at zero, but at a random moment in time. This should have a good effect on generalization abilities, however, the average distance between starts should not be less than the maximum buy age. Therefore (3 is the size of action space {BUY, SELL, HOLD}):

$$\frac{\varepsilon}{3}\, max\_steps < max\_buy\_age$$

SECTION 4. TRAINING RESULTS

4.1. Results without shift and commission

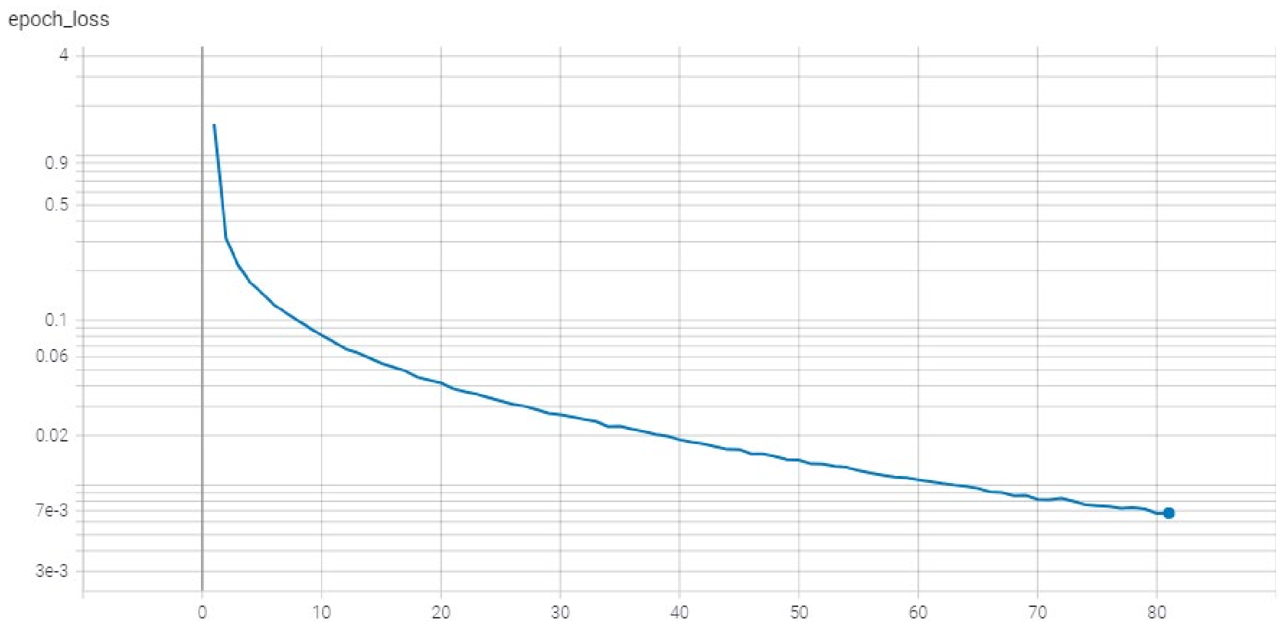

Figure 9 – Loss function (logarithmic scale)

Further, by revenue, we mean 1 dollar received to 1 invested, at the rate of $10'000/BTC.

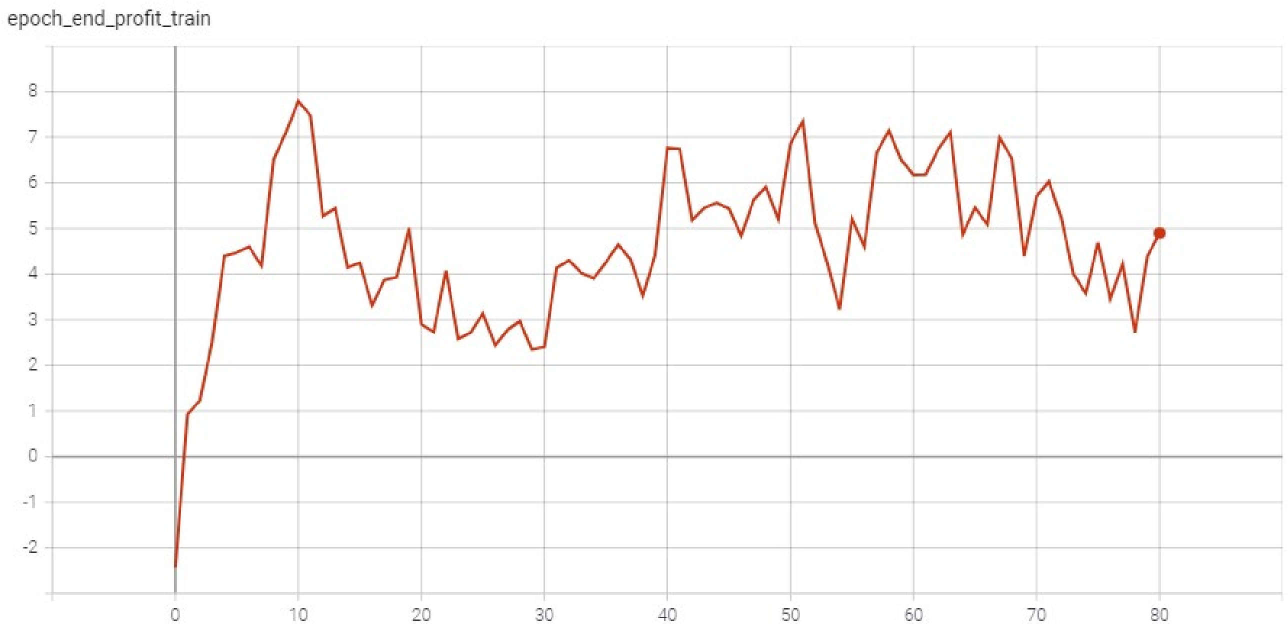

Figure 10 – Revenue of epoch during training

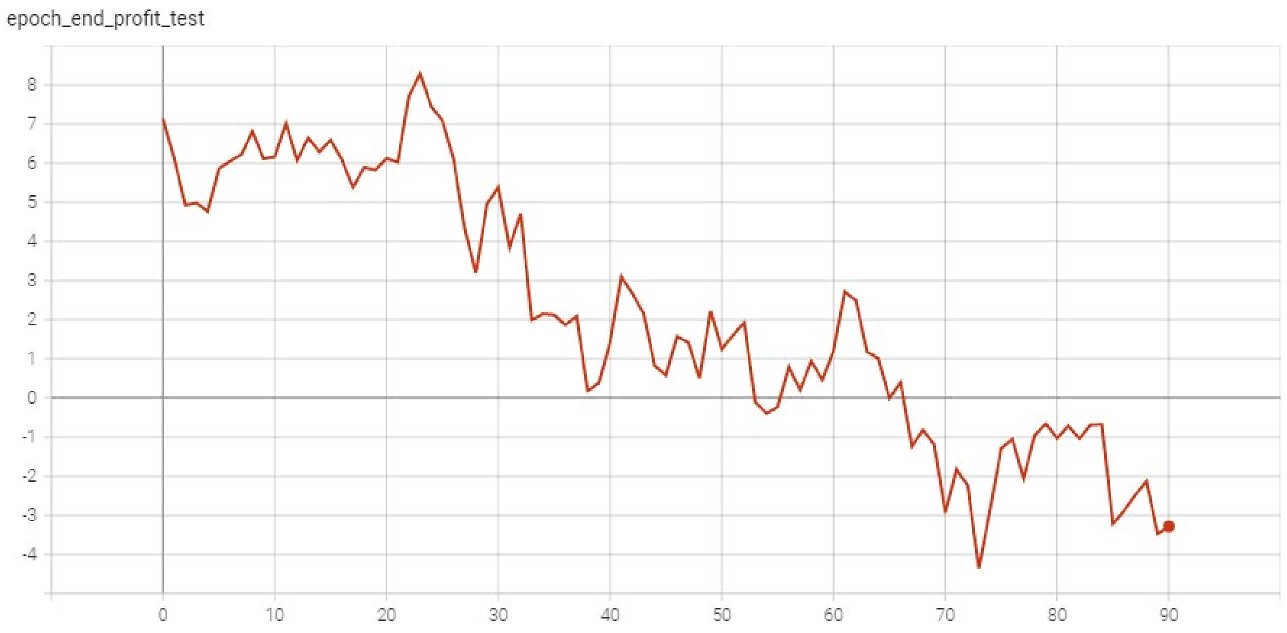

Figure 11 – Revenue of epoch during testing

4.2. Results with shift and commission

After training without shift and commission, the best weights were used to optimistically initialize the agent in the environment with shift and commission.

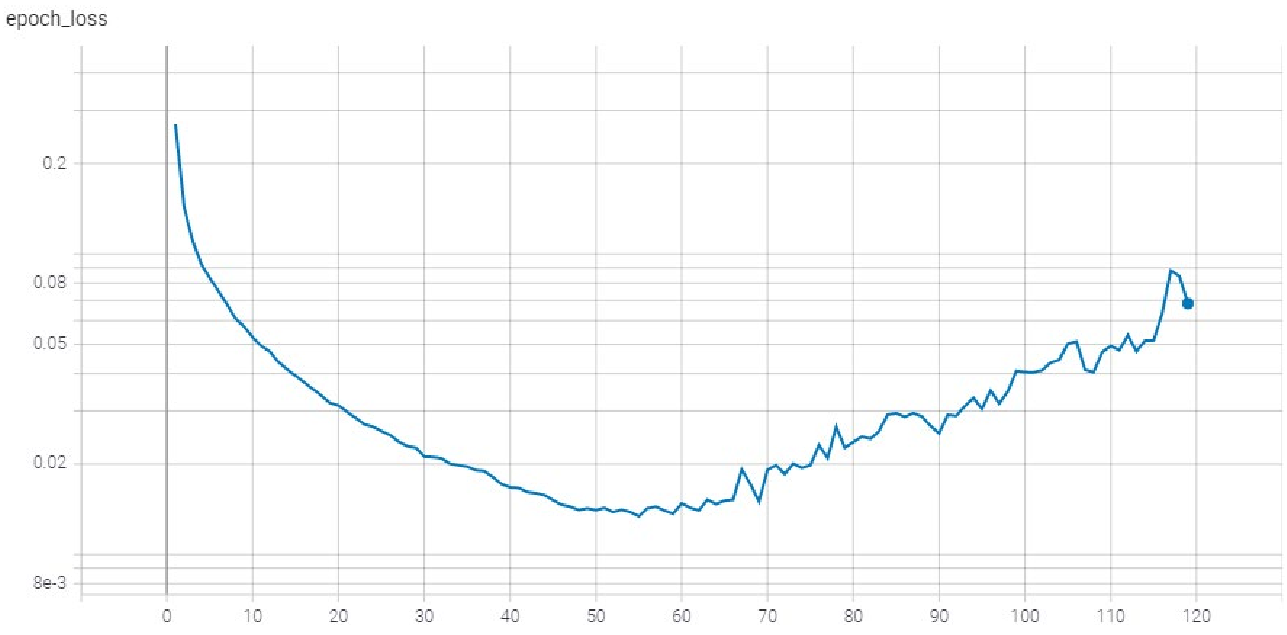

Figure 12 – Loss function (logarithmic scale)

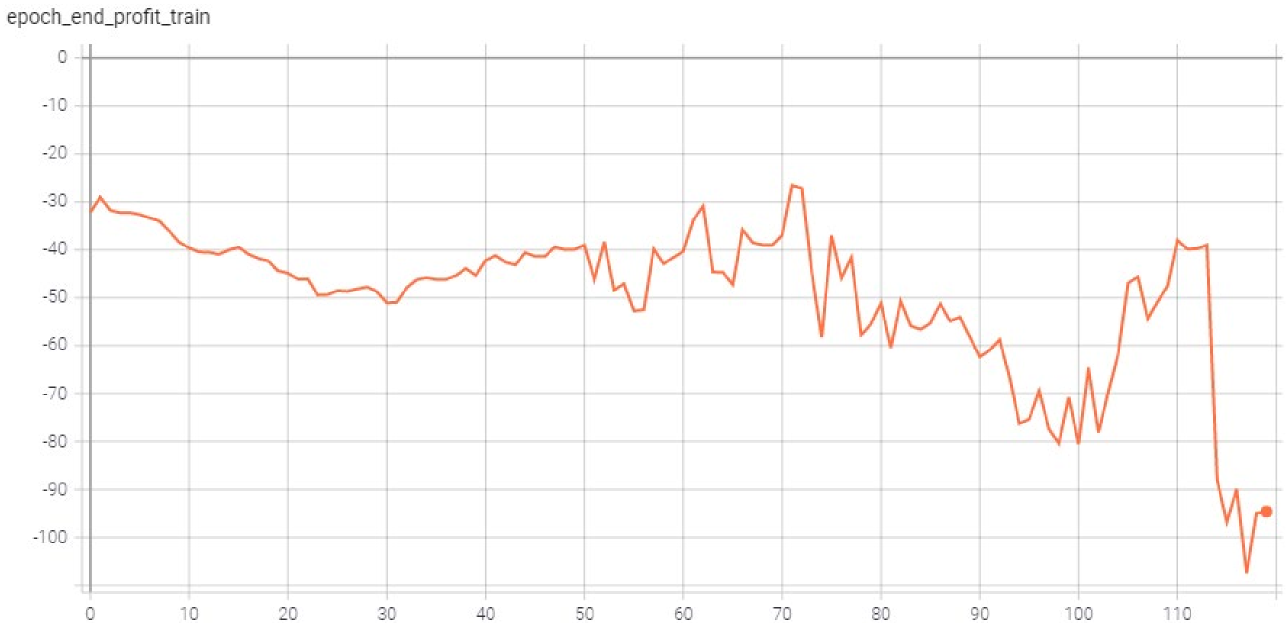

Figure 13 – Revenue of epoch during training (optimistic initialization)

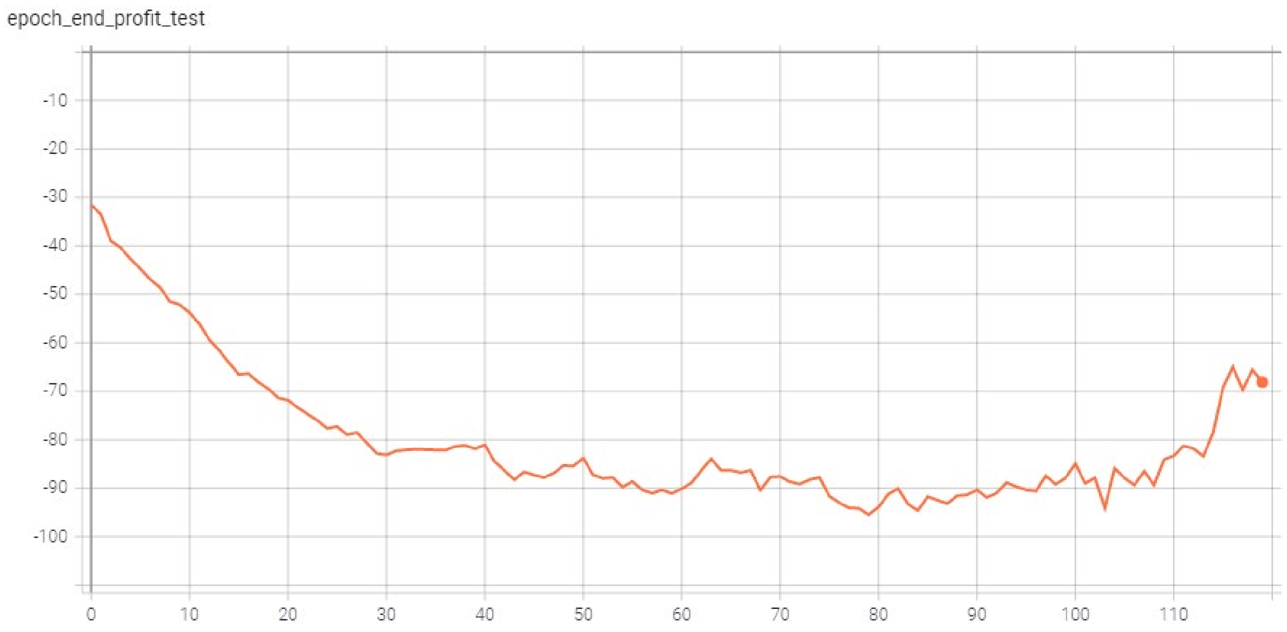

Figure 14 – Revenue of epoch during testing (optimistic initialization)

CONCLUSIONS

Due to the lack of a loss function during testing (validation), we can only draw conclusions from revenue. Results without shift and commission are difficult to interpret because it is not completely clear whether overfitting is the cause of the drop in revenue. However, the stable high profit during the first twenty epochs indicates the ability to work in an idealized environment.

We have a much worse result with the shift and the commission: the revenue is negative without any hint of growth.

Perhaps we should try to use the improved version of DQN – Rainbow [24]. Its biggest advantages are the use of a prioritized replay buffer [25] and multi-step rewards when updating the value function, which can help make more informed decisions. It is also advisable to replace the ε-greedy strategy with an upper confidence bound [26], this will allow taking into account the last time a random action was performed, for gradual rather than chaotic exploration.

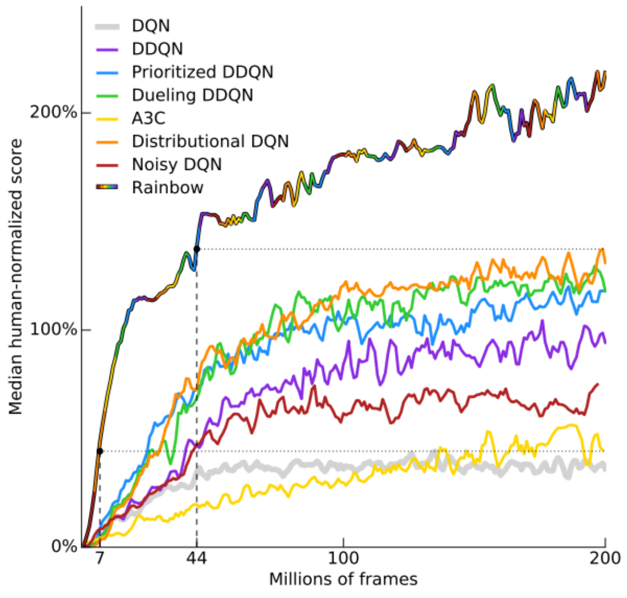

Figure 15 – Performance comparison of different versions of DQN [24] in the learning environment of Atari 2600 [27] arcade games

More radical would be to abandon reinforcement learning in favor of evolutionary algorithms: genetic programming, genetic algorithms, or neuroevolution [28].

REFERENCES

1. Brockman G., Cheung V., Pettersson L., Schneider J., Schulman J., Tang J., and Zaremba W. OpenAI Gym. 2016.
2. Mnih V., Kavukcuoglu K., Silver D., Graves A., Antonoglou I., Wierstra D., and Riedmiller MA, "Playing Atari with Deep Reinforcement Learning," // ArXiv, no. abs/1312.5602, 2013.
3. Mnih V., Kavukcuoglu K., Silver D., Rusu AA, Veness J., Bellemare MG, Graves A., Riedmiller MA, Fidjeland A., Ostrovski G., et al., "Human-level control through deep reinforcement learning," // Nature, No. 518, 2015. P. 529-533.
4. Abadi M., Barham P., Chen J., Chen Z., Davis A., Dean J., Devin M., Ghemawat S., Irving G., Isard M., et al. TensorFlow: A system for large-scale machine learning // OSDI. 2016.
5. Chollet F., others. Keras. 2015.
6. Plappert M. keras-rl. GitHub, 2016.
7. Harris CR, Millman KJ, van der Walt SJ, Gommers R., Virtanen P., Cournapeau D., Wieser E., Taylor J., Berg S., Smith NJ, et al., "Array programming with NumPy," // Nature, No. 585, September 2020. P. 357–362.
8. Virtanen P., Gommers R., Oliphant TE, Haberland M., Reddy T., Cournapeau D., Burovski E., Peterson P., Weckesser W., Bright J., et al., "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," // Nature Methods, No. 17, 2020. P. 261–272.
9. Sheppard K., Khrapov S., Lipták G., mikedeltalima, Capellini R., Hugle, esvhd, Fortin A., JPN, Li W., and et al., "bashstage/arch: Release 4.19," 2021.
10. Pedregosa F., Varoquaux G., Gramfort A., Michel V., Thirion B., Grisel O., Blondel M., Prettenhofer P., Weiss R., Dubourg V., et al., "Scikit-learn: Machine Learning in Python," // Journal of Machine Learning Research, No. 12, 2011. P. 2825–2830.

11. pandas development team T. pandas-dev/pandas: Pandas. Zenodo, 2020.
12. Hunter JD, "Matplotlib: A 2D graphics environment," // Computing in Science & Engineering, No. 9, 2007. P. 90–95.
13. Sutton R., Barto A., "Reinforcement Learning: An Introduction," // IEEE Transactions on Neural Networks, No. 16, 2005. P. 285-286.
14. Hosking JRM, "Fractional differentiation," // Biometrika, No. 68, 1981. P. 165-176.
15. Jensen AN, Nielsen M., "A Fast Fractional Difference Algorithm," // Econometrics: Mathematical Methods & Programming eJournal, 2013.
16. Brent R., "Table errata: Algorithms for minimization without derivatives (Prentice-Hall, Englewood Cliffs, NJ, 1973)," // Mathematics of Computation, No. 29, 1975. P. 1166.
17. Elliott G., Rothenberg T., and Stock J., "Efficient Tests for an Autoregressive Unit Root," Econometrica, No. 64, 1996, pp. 813-836.
18. Kwiatkowski D., Phillips P., Schmidt P., and Shin Y., "Testing the null hypothesis of stationarity against the alternative of a unit root: How sure are we that economic time series have a unit root?," // Journal of Econometrics, No. 54, 1992. P. 159-178.
19. Campbell J., Lo A., Mackinlay A., and Whitelaw RF The Econometrics of Financial Markets. Princeton University Press, 1996. P. 48–55.
20. Misra D., "Mish: A Self Regularized Non-Monotonic Neural Activation Function," // ArXiv, no. abs/1908.08681, 2019.
21. Gal Y., Hron J., and Kendall A. Concrete Dropout // NIPS. 2017.
22. He K., Zhang X., Ren S., and Sun J., "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," // 2015 IEEE International Conference on Computer Vision (ICCV), 2015. P. 1026-1034.
23. Glorot X., Bengio Y. Understanding the difficulty of training deep feedforward neural networks // AISTATS. 2010.

24. Hessel M., Modayil J., Hasselt HV, Schaul T., Ostrovski G., Dabney W., Horgan D., Piot B., Azar MG, and Silver D. Rainbow: Combining Improvements in Deep Reinforcement Learning // AAAI. 2018.
25. Schaul T., Quan J., Antonoglou I., and Silver D., "Prioritized Experience Replay," // CoRR, #abs/1511.05952, 2016.
26. Auer P., "Using Confidence Bounds for Exploitation-Exploration Trade-offs," // J. Mach. Learn. Res., No. 3, 2002. P. 397-422.
27. Bellemare MG, Naddaf Y., Veness J., and Bowling M. The Arcade Learning Environment: An Evaluation Platform for General Agents (Extended Abstract) // IJCAI. 2015.
28. Such FP, Madhavan V., Conti E., Lehman J., Stanley K., and Clune J., "Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning," // ArXiv, no abs /1712.06567, 2017.

APPENDIX A.
IMPLEMENTATION OF FRACTIONAL DIFFERENTIATION

```
from typing import Optional

import numpy as np
import scipy.signal
import scipy.special
import scipy.stats
from numba import jit


def get_weights(order: float, weights_num: int, mode: int = 0) -> np.ndarray:
    if mode == 0:
        k = np.arange(1, weights_num)
        weights = np.concatenate(([1], np.cumprod((k - order - 1) / k)))
    elif mode == 1:
        s = np.tile([1.0, -1.0], -(-weights_num // 2))[:weights_num]
        weights = s * scipy.special.binom(order, np.arange(weights_num))
    else:
        raise mode

    return weights


def get_weights_eps(order: float, max_weights_num: int, eps: float = EPS) -> np.ndarray:
    weights = [1.]
    for k in range(max_weights_num - 1):
        w = weights[-1] * -(order - k) / (k + 1)
        if abs(w) <= eps:  # <= 0
            break
        weights.append(w)
    return np.array(weights)


def num_from_threshold(weights, min_rel_imp):
    weight_sums = np.cumsum(np.abs(weights[::-1]))
    weight_sums = weight_sums / weight_sums[-1]
    take_num = len(weight_sums[(weight_sums >= min_rel_imp) & (weight_sums > 0)])
    return take_num


def diff_numba(array: np.ndarray, weights: np.ndarray):
    res = np.zeros_like(array, dtype=np.float64)
    shift = np.zeros(array.size * 2)
    shift[array.size:] = array
    for i, w in enumerate(weights):
        res += w * shift[array.size - i: 2 * array.size - i]  # ~ w * shift_right(array, i)
    return res
```

```
def frac_diff(array: np.ndarray, order: float, max_weights_num: Optional[int] = None,
              min_rel_imp: Optional[float] = None, all_weights=False, do_skip=True,
              numba_kwargs: Optional[dict] = None):
    assert isinstance(array, np.ndarray)
    assert len(array.shape) == 1, array.shape
    assert 0 <= order <= 1, order

    if numba_kwargs is None:
        numba_kwargs = {'parallel': True, 'nopython': True}

    if min_rel_imp is not None:
        assert 0 <= min_rel_imp <= 1, min_rel_imp
        weights = get_weights_eps(order, max_weights_num=array.size)
        if all_weights:
            skip = num_from_threshold(weights, min_rel_imp) - 1
        else:
            take_weights = num_from_threshold(weights, min_rel_imp)
            weights = weights[:take_weights]
            skip = take_weights - 1
    elif max_weights_num is not None:
        assert 1 <= max_weights_num <= array.size, (1, max_weights_num, array.size)
        weights = get_weights_eps(order, max_weights_num)
        skip = len(weights) - 1
    else:
        raise (max_weights_num, min_rel_imp)

    if skip / array.size > 0.1:
        print('Skip ratio', skip, array.size, skip / array.size)

    res = jit(**numba_kwargs)(diff_numba)(array, weights)

    return res[skip if do_skip else 0:]
```

```
def frac_diff_fft(array: np.ndarray, order: float, skip_threshold: Optional[float] = None, do_skip=True):
    assert isinstance(array, np.ndarray)
    assert len(array.shape) == 1, array.shape
    assert 0 <= order <= 1, order

    if skip_threshold is None:
        skip_threshold = n_sigma_p(n=3)
    weights = get_weights(order, weights_num=array.size)
    if order == 0:
        return array
    elif order == 1:
        return np.diff(array)
    else:
        c1 = np.cumsum(weights)
        c1 = c1 / c1.max()

        c2 = np.cumsum(c1)
        c2 = c2 / c2.max()

        c3 = np.diff(c2)
        c3 = c3 / c3.max()

        skip = len(c3[c3 > n_sigma_p(skip_threshold)])

        if skip > array.size * 0.2:
            print('Size', order, skip, array.size, array.size * 0.2)
            print('Consider decrease threshold')

    res = scipy.signal.convolve(array, weights, mode='full')[:array.size]
    # ~res = scipy.fft.irfftn(scipy.fft.rfftn(array) * scipy.fft.rfftn(weights))

    return res[skip if do_skip else 0:]
```