Machine Learning Engineer Nanodegree

Capstone Project

Jörg Strebel
November 27th 2019

I. Definition

Project Overview

As people age, they inevitably retire from their jobs to live off their savings. If you google for “retirement investing today”, you will find around 143 million related web pages on the WWW. So the topic is very relevant to many people. In Germany, a regular employee has to spend a certain percentage of his salary on a state-run insurance, but they can also opt to save more and put the money into financial products to secure their financial needs during retirement. One possible way to invest money for retirement is to participate in the stock market. This option has become more important over the last 10 years, as the fixed-interest investment opportunities have largely vanished or yield unsatisfactory profits.

If a person then chooses to use the stock market for long-term retirement investment, they need to have a sound and secure investment strategy, i.e. how much to invest, when to buy, what to buy (or sell). Ideally, the strategy leads to a long and steady increase of the value of the portfolio, so that the money is available when they retire. Now the problem arises, how the individual can come up with such a strategy.

How to do retirement investing has been a serious research topic over the last decades (e.g. Farhi and Panageas: Saving and investing for early retirement: A theoretical analysis, Gerber and Shiu: Investing for Retirement, Bierwirth: INVESTING FOR RETIREMENT: USING THE PAST TO MODEL THE FUTURE). I also have a personal motivation to investigate this problem, as I am an active investor at German, European and US stock exchanges (mainly ETFs). If this project is successful, it will be very helpful for my investment decisions.

Problem Statement

This capstone project aims to solve the problem of finding a good long-term investment strategy under budget, time and risk constraints on ETFs suitable for retirement investing. Currently, investors that follow a passive approach usually have simple strategies which might leave money on the table, and investors that follow an active approach need a lot of time to inform themselves and make trading decisions. An investor usually faces a budget and time constraint, as they invest only a limited amount of money per time period (e.g. per month) and they would like to minimize the overall transaction fees; also, the investor face a risk constraint, as they cannot afford to lose their retirement savings.

The assessment of the algorithm to find such a strategy (i.e. the learning part of an agent) is straightforward, as you can reuse the existing financial KPIs like returns, profitability, volatility etc. Both the inputs (trading actions) and the outputs (financial outcomes) are quantifiable, measurable and replicable (for historic stock data).

The problem itself has already been researched scientifically (e.g. Reinforcement Learning for Finance and Snow - Machine Learning in Asset Management).

As a solution, I would like to suggest a reinforcement-learning based agent (DeepRL agent), that learn a good trading strategy. The agent would decide when to trade and how much to buy from the three given ETFs. It is also supposed to stay within the given budget constraints. The agent learns a trading strategy based on historical stock data and then runs daily and outputs the amounts to buy or sell from the funds in the portfolio. The goal is not to generate sustainable or supreme α (Definition), but to automate the investment decision, free the investor of manual labor and take advantage of mid-term market developments. The project does not focus on long-term stock forecasting using statistical methods (e.g. like in Theory of Long-Term Stock Forecasting), but on reinforcement learning-based solutions as a novel way to solve this problem.

Metrics

The evaluation happens on a subset of the available historic data, the test set. The two investment strategies are evaluated using the stock data in this test set. The stock data contains one time series of prices per day per fund, so there are no categorical attributes and no risk of imbalanced classes. All three funds time series are available on the complete time period under investigation.

The metric to compare both strategies will be the total value increase after the test period, i.e. how much has the total value of the portfolio increased using either one of the trading strategies. You can then calculate the percentage increase based on the initial budget. A similar approach is followed in Stock prediction models. The total value is the Euro value of the portfolio at the last day of the test period plus the remaining budget. This metric is aimed at the investor for comparing the performance of different agents.

The reward function is different from this metric; their purpose is to guide a learning agent to the best strategy and to assess learning progress. That’s why the reward function includes the total value increase and both penalties and transaction cost. This metric is aimed at the agent.

In the end, there are two parallel metrics; one for a human investor and one for the agents. Both metrics are very general, their values are highly correlated and both are applicable to any sort of agent. The reward function is part of the task environment, and as every agent interacts with the same task environment, each agent is judged by the same reward function. So the total reward allows a fair comparison between different agents.

II. Analysis

Data Exploration

Stock Data and US interest rates

Source of the data

Any trading strategy has to be backtested using historic stock data; this project is using the historic stock data of the following three indices with daily data granularity:

- S&P 500 index:
  - https://finance.yahoo.com/quote/%5EGSPC/history?p=%5EGSPC,
  - https://www.ariva.de/ishares_s-p_500_ucits_etf-fonds/historische_kurse
- MSCI World Index:
  - https://finance.yahoo.com/quote/X010.DE/history?p=X010.DE,
  - https://www.ariva.de/comstage_msci_world_trn_ucits_etf-fonds
- STOXX EUROPE 600 ETF: https://www.ariva.de/db_x-tr-stoxx_europe_600_etf_inhaber-anteile_1c-fonds/historische_kurse

The three index funds are ideal financial products for retirement investing, as they offer very high diversification, and thus very low risk of wild swings or long-term losses. The three ETFs are also globally diversified: one from the US, one from Europe and one covering global enterprises.

The Websites provide the data in one CSV format for each index fund, with the columns: date, open price, high price, low price, close price. These are all real-valued columns, so there is no categorical data and no classes. Hence there can be no imbalance in the data sets. All funds data consists of time series with a daily granularity and hence, they feature the same number of data points. The data set does not show individual stock trades (buy/sell actions), only the closing price, so it is not imbalanced in terms of trade decisions.

The number of data points depends on the number of trading days at the stock exchange. The file resources/wkn_ETF110_historic.csv gives an example of the data; I plan to use the close price which is available for the complete time period. For a 10-year time period, we have roughly 2170 records for daily stock prices per fund. Please see the following sample data for MSCI World Index:

Date	Open	High	Low	Close
11.10.2019	60,9378 €	60,9378 €	60,9378 €	60,9378 €
10.10.2019	60,1189 €	60,1189 €	60,1189 €	60,1189 €
09.10.2019	59,7625 €	59,7625 €	59,7625 €	59,7625 €

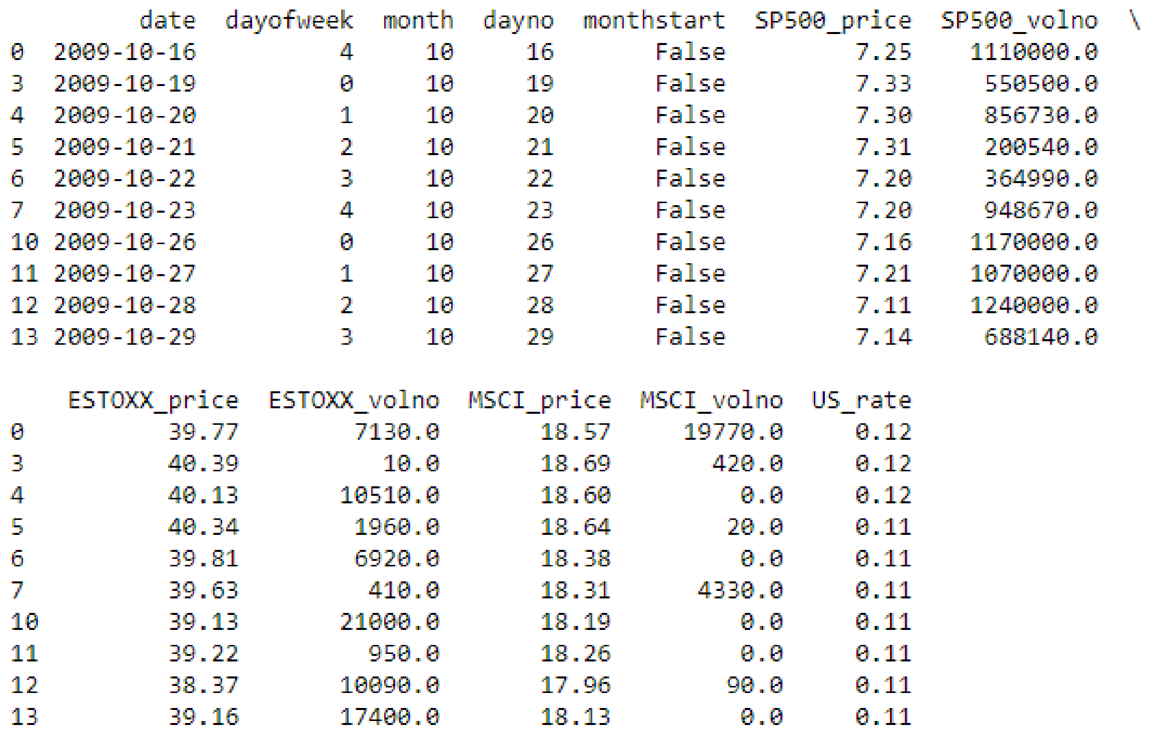
Ideally, the historic data would span 10 years of stock data, with the last 2 years as test data.

Other inputs consist of the following financial parameters:

- available monthly savings of investor in €

•available initial budget in € (no credit will be considered)
•transaction fees for stock transactions (no savings plan).

Please see the following chart for the first ten lines in my financial data set. The price columns are closing prices of the corresponding ETF:

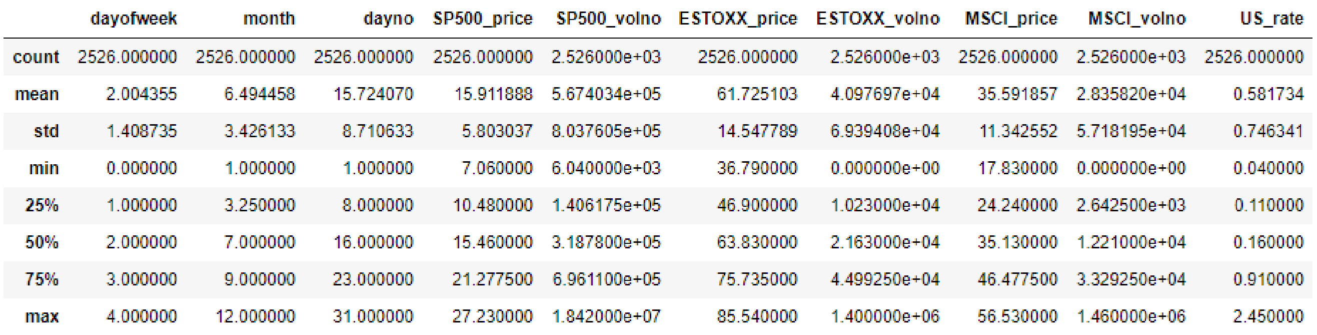


```
          date  dayofweek  month  dayno  monthstart  SP500_price  SP500_volno  \
0   2009-10-16          4     10     16       False         7.25    1110000.0
3   2009-10-19          0     10     19       False         7.33     550500.0
4   2009-10-20          1     10     20       False         7.30     856730.0
5   2009-10-21          2     10     21       False         7.31     200540.0
6   2009-10-22          3     10     22       False         7.20     364990.0
7   2009-10-23          4     10     23       False         7.20     948670.0
10  2009-10-26          0     10     26       False         7.16    1170000.0
11  2009-10-27          1     10     27       False         7.21    1070000.0
12  2009-10-28          2     10     28       False         7.11    1240000.0
13  2009-10-29          3     10     29       False         7.14     688140.0

    ESTOXX_price  ESTOXX_volno  MSCI_price  MSCI_volno  US_rate
0          39.77        7130.0       18.57     19770.0     0.12
3          40.39          10.0       18.69       420.0     0.12
4          40.13       10510.0       18.60         0.0     0.12
5          40.34        1960.0       18.64        20.0     0.11
6          39.81        6920.0       18.38         0.0     0.11
7          39.63         410.0       18.31      4330.0     0.11
10         39.13       21000.0       18.19         0.0     0.11
11         39.22         950.0       18.26         0.0     0.11
12         38.37       10090.0       17.96        90.0     0.11
13         39.16       17400.0       18.13         0.0     0.11
```

Basic statistic information

The following table shows the basic statistic information of the financial data after cleansing.



	dayofweek	month	dayno	SP500_price	SP500_volno	ESTOXX_price	ESTOXX_volno	MSCI_price	MSCI_volno	US_rate
count	2526.000000	2526.000000	2526.000000	2526.000000	2.526000e+03	2526.000000	2.526000e+03	2526.000000	2.526000e+03	2526.000000
mean	2.004355	6.494458	15.724070	15.911888	5.674034e+05	61.725103	4.097697e+04	35.591857	2.835820e+04	0.581734
std	1.408735	3.426133	8.710633	5.803037	8.037605e+05	14.547789	6.939408e+04	11.342552	5.718195e+04	0.746341
min	0.000000	1.000000	1.000000	7.060000	6.040000e+03	36.790000	0.000000e+00	17.830000	0.000000e+00	0.040000
25%	1.000000	3.250000	8.000000	10.480000	1.406175e+05	46.900000	1.023000e+04	24.240000	2.642500e+03	0.110000
50%	2.000000	7.000000	16.000000	15.460000	3.187800e+05	63.830000	2.163000e+04	35.130000	1.221000e+04	0.160000
75%	3.000000	9.000000	23.000000	21.277500	6.961100e+05	75.735000	4.499250e+04	46.477500	3.329250e+04	0.910000
max	4.000000	12.000000	31.000000	27.230000	1.842000e+07	85.540000	1.400000e+06	56.530000	1.460000e+06	2.450000

“dayoftheweek” goes from 0-4 as Saturday and Sunday are no trading days at the stock exchange. “month” goes from 1-12, and “dayno” goes from 1-31 as expected. We have 2526 records in total after data cleansing.

For more details, please see the Jupyter notebook of this project.

Abnormalities in the data

Missing values are present in the data, as the stock exchange is not in session everyday, so we are lacking data for the weekends. This affects all ETF price and volume data.

For detecting outliers in the data, the Tukey's Method will be used: An outlier step is calculated as 1.5 times the interquartile range (IQR). A data point with a feature that is beyond an outlier step outside of the IQR for that feature is considered abnormal.

There 192 outliers for the S&P 500 traded volume, 231 outliers for the EuroSTOXX volume, 219 outlier for the MSCI Word Index volume. There are 245 outliers for the US rate.

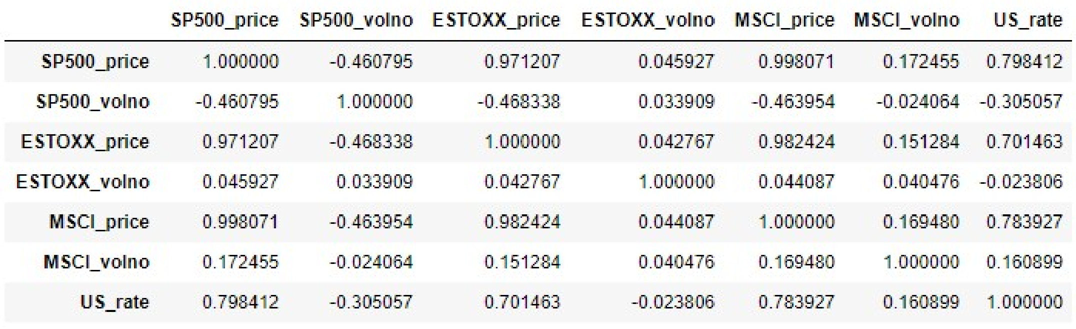
None of these outliers are critical, as the traded volume depends on a variety of factors; all three ETFs are very liquid assets on the XETRA stock exchange, so we can exclude a systematic error here. Moreover, the data is automatically collected by the stock exchange, which rules out errors as well.

The US rate is set by the US Treasury and varies over time, as we'll see below. All records will be retained.

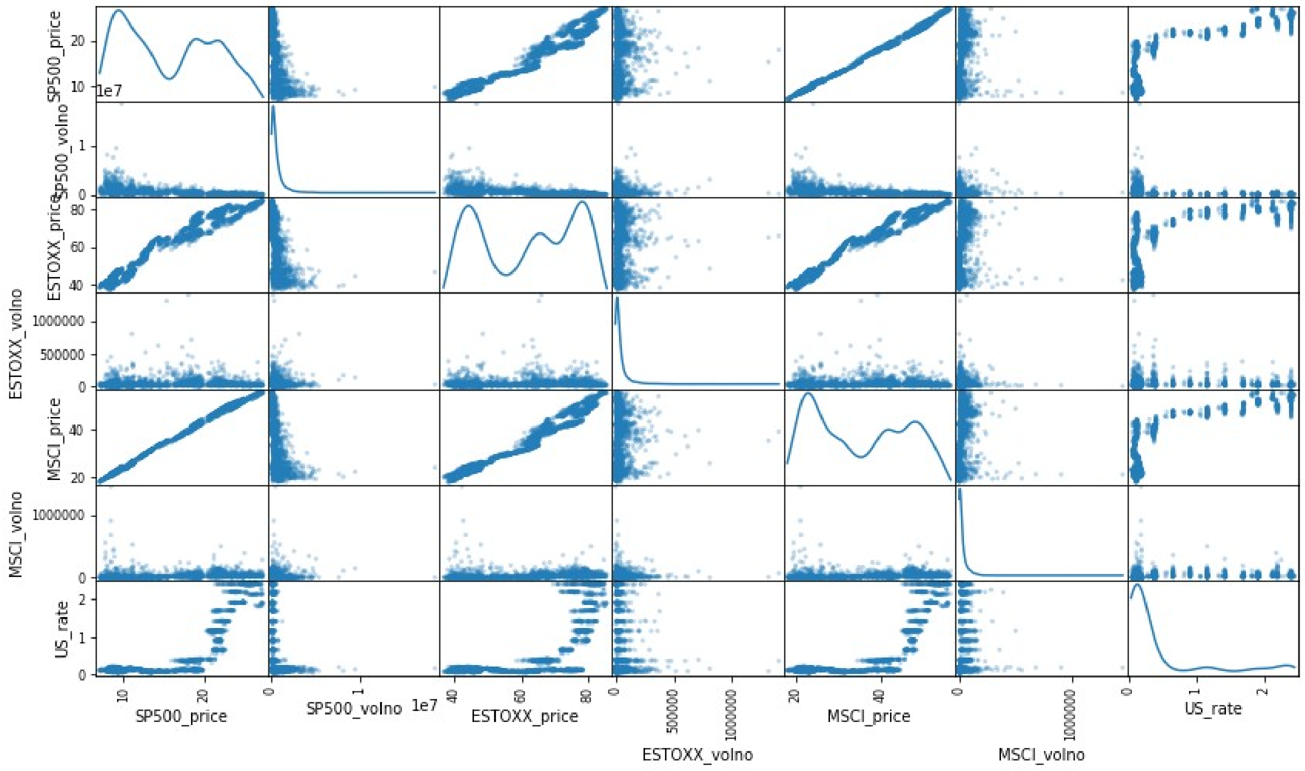
Strong Correlations are present: all three stock prices are heavily correlated over time, which is partly understandable, as the S&P500 ETF and the MSCI World ETF share a number of stocks (global enterprises i.e. Facebook, Alphabet etc.). This also goes for the EuroSTOXX and the MSCI World ETF.

The correlation of the S&P500 with the EuroSTOXX only goes to show how globalized the economy has become. Big enterprises from Europe make business on the same economic playing field as their American counterparts.

These correlations are the reason why both agents currently only trade the S&P500 – there is little reason to trade three ETFs in parallel, as they do not offer any diversification potential (they all move about the same at the same time).



	SP500_price	SP500_volno	ESTOXX_price	ESTOXX_volno	MSCI_price	MSCI_volno	US_rate
SP500_price	1.000000	-0.460795	0.971207	0.045927	0.998071	0.172455	0.798412
SP500_volno	-0.460795	1.000000	-0.468338	0.033909	-0.463954	-0.024064	-0.305057
ESTOXX_price	0.971207	-0.468338	1.000000	0.042767	0.982424	0.151284	0.701463
ESTOXX_volno	0.045927	0.033909	0.042767	1.000000	0.044087	0.040476	-0.023806
MSCI_price	0.998071	-0.463954	0.982424	0.044087	1.000000	0.169480	0.783927
MSCI_volno	0.172455	-0.024064	0.151284	0.040476	0.169480	1.000000	0.160899
US_rate	0.798412	-0.305057	0.701463	-0.023806	0.783927	0.160899	1.000000



Agent States and Actions

The agent state and the actions are also main input data source for a reinforcement learning-based agent. The state is defined according to https://teddykoker.com/2019/06/trading-with-reinforcement-learning-in-python-part-ii-application/ as a numpy-array with the stock prices, the budget and the number of shares per ETF:

$$x_t = [price_{t-M}, \ldots, price_t, r_{t-M}, \ldots, r_t, N_t, B_t, Bm_t]$$

price = share price in time window

r = US interest rate in time window

N = Number of shares in portfolio

B = Budget available

Bm = Monthly budget available

Example data:

```
[      0.         0.         0.         0.         0.         0.         0.
      0.         0.         7.25       0.         0.         0.         0.
      0.         0.         0.         0.         0.         0.12       0.
 120000.      1000.    ]
```

You can see that the prices and interest rates are initialized with 0, except for the price at time slot 1 (7.25€) and the interest rate (0.12). Number of shares is 0 and the available budget is the starting budget (120000€) and the available monthly budget is the starting value (1000€).

The actions a_t consist of floating point values, which indicate the amount of stocks to sell or buy at time t, where buying is indicated by a positive value, and selling by a negative value. The actions are limited by high and low limits (defined in task.py).

Those limits are:

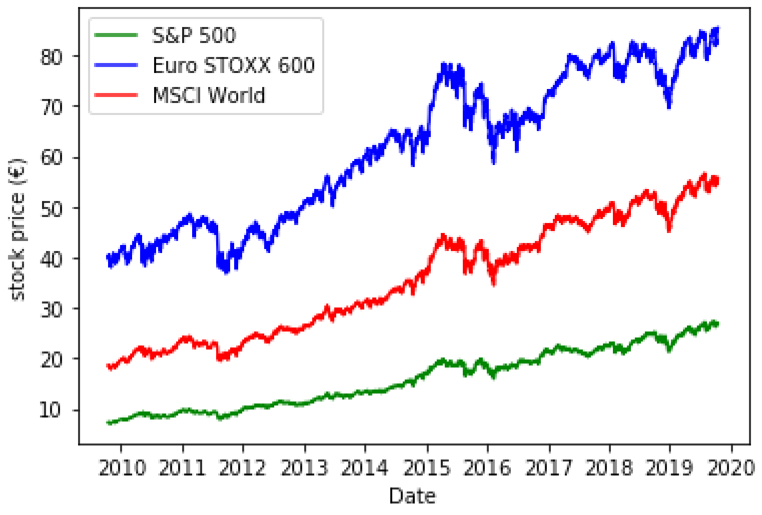
$$\frac{\pm Bm}{max(p_{1..t})}$$

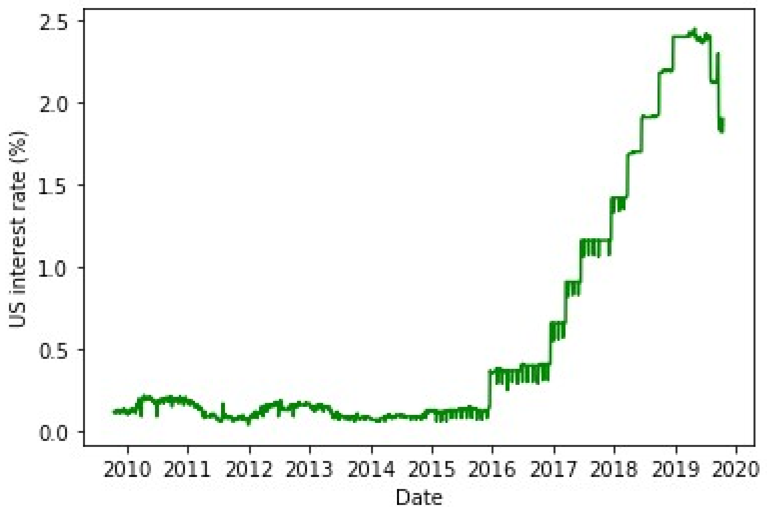
So the agent can only buy or sell the number of stocks that are within its monthly budget.

Exploratory Visualization

The following two visualizations show the time series of both the stock prices and the US interest rate. They show the market environment, in which the agents operate. The price variances massively affect the trading decisions, and the variances in interest rate lead to a varying motivation to buy shares.

Especially in the stock prices basically only knew one direction over the last ten years: up. So the agents will likely more buy than sell.





Algorithms and Techniques

The DeepRL agent has to solve the prediction and the control problem of reinforcement learning, i.e. it has to learn the value functions from the interaction with the environment and and it has to derive an optimal policy from these value functions.

Q-Learning is a way to learn the optimal action value function Q using an off-policy temporal difference algorithm. The iterative improvement of Q is defined by :

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \Big].$$

(Source: [2, p. 153]) S are states, A are actions, R are rewards, α and γ are learning parameters. Q can then be used to derive a policy, i.e. choosing A from S by using an ε-greedy approach. When this algorithms runs for several episodes, both the policy and the value functions should move toward their optimal values (generalized policy iteration).

This approach has a drawback, as it cannot cope with continuous states and actions and it is an indirect approach, as we first need to estimate Q and then try to derive a good policy from it; Q can be very complex, so it might make sense to learn the policy directly (see policy gradients below).

Q-Learning can be extended with function approximation to also cover continuous states and actions and hence continuous value functions q(s,a,w). The learning algorithm for the control problem is very similar to the one for the original Q-learning, but the update rule for the parameter w is new (Source: [3]):

$$\Delta w = \alpha \left(R + \gamma max_a \hat{q}(S', a, w) - \hat{q}(S, A, w) \right) \nabla_w \hat{q}(S, A, w)$$

R are rewards, S’ is the next state.

In order to learn the optimal policy π* directly, we can use policy gradients to estimate the underlying function parameters θ, i.e. we do a function approximation of policy function for π(s,a,θ) . In order to derive the optimal policy, we define an objective function J(θ) which is the expected value of the reward function given that policy. The gradient of J(θ) then gives us the update rule for θ, which in turn determines the policy:

$$\nabla_\theta J(\theta) = \nabla_\theta E_\pi [R(\tau)] = E_\pi [\nabla_\theta (\log \pi(s, a, \theta)) R(\tau)]$$

$$\Delta\theta=\alpha\nabla_\theta(\log\pi(s,a,\theta))R(\tau)$$

(Source: [3]) α is the step size in this context. R is the score / reward function and will be replaced by the action-value function q in the future. The Actor-Critic Method brings back the estimated action value function $\hat{q}$ as a better score function for the policy update.

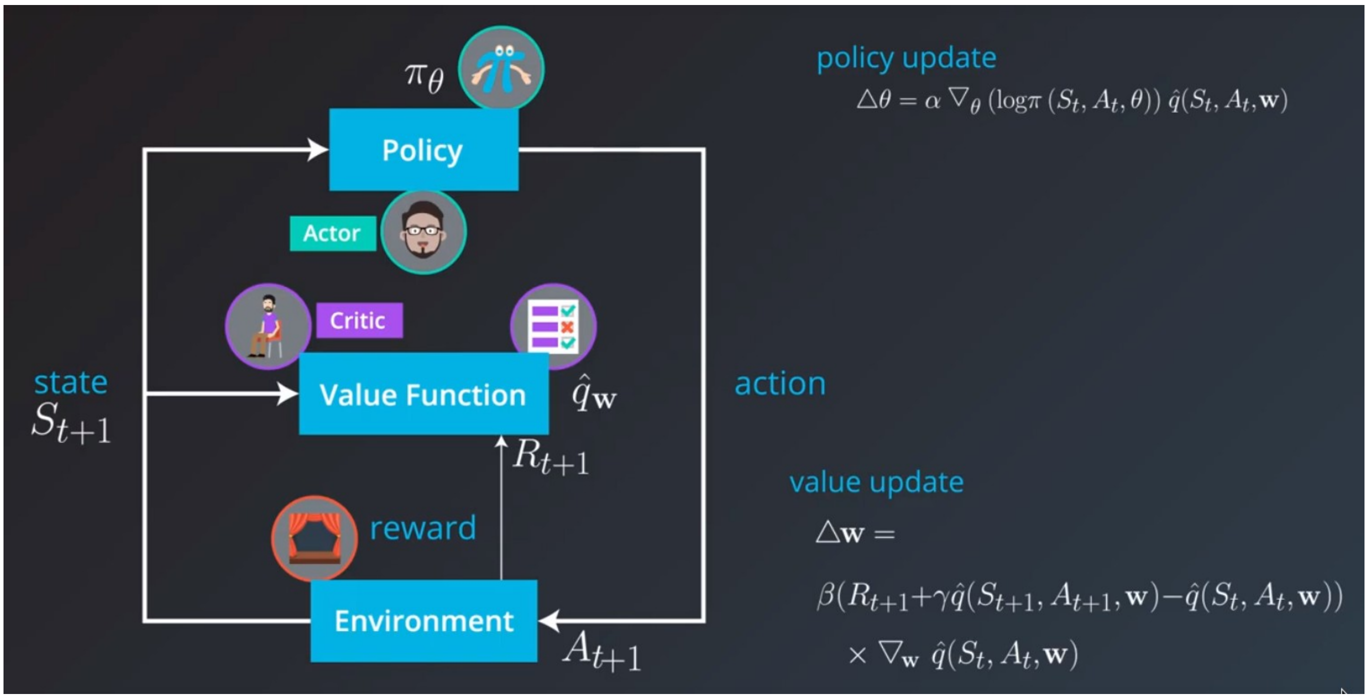
$$\Delta\theta=\alpha\nabla_\theta(\log\pi(s,a,\theta))\hat{q}(s,a,w)$$

In turn, we have a second function approximator for the action value function $\hat{q}$ with its own update rule (in close resemblance to the Q-learning update rule and the temporal difference learning method):

$$\Delta w=\beta(R+\gamma\hat{q}(S',A',w)-\hat{q}(S,A,w))\nabla_w\hat{q}(S,A,w)$$

β is the learning rate for value updates; S', S are adjacent states and A', A are adjacent actions. θ is the parameter of the policy π(s,a,θ) (actor), w is the parameter for the action value function $\hat{q}$ (critic).

The whole learning process is shown in the below picture (Source: [3]). The actor uses the policy to determine next actions, and the policy is changed by the input of the critic, who in turn learns to evaluate these actions. Both function approximations are implemented using deep neural networks (actor.py, critic.py).


π_θ
Policy
Actor
Critic
Value Function
$\hat{q}_{\mathbf{w}}$
state
S_{t+1}
R_{t+1}
reward
Environment
A_{t+1}
action
policy update
$\Delta\theta = \alpha\,\nabla_\theta\,(\log\pi\,(S_t, A_t, \theta))\,\hat{q}(S_t, A_t, \mathbf{w})$
value update
$\Delta\mathbf{w} = \beta(R_{t+1}+\gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w})-\hat{q}(S_t, A_t, \mathbf{w}))\times\nabla_{\mathbf{w}}\,\hat{q}(S_t, A_t, \mathbf{w})$

The neural networks used are regular, dense feed-forward multi-layer perceptron networks which are trained by backpropagation. The DeepRL agent uses one input layer, two dense hidden layers and an output layer. Backpropagation is a learning algorithm, that uses gradient descent on the loss function and changes the weights in the neural network to minimize the loss; the loss function calculates the difference between the true and the estimated action value function (critic) or policy (actor) with respect to the function parameter w/θ. As we don't have the true action values, we use temporal difference learning TD(0) as a substitute for the loss signal.

For my DeepRL agent, the learning algorithm is deep deterministic policy gradients (DDPG), based on source code from the Udacity quadcopter project. This algorithm fits nicely, as it can process a continuous state and action space and is deterministic in nature.

DDPG:

- deep – it uses deep neural networks for function approximations.
- Deterministic – it returns discrete values, no distributions
- policy gradient – it uses policy gradients.

The critic uses Q-learning for its value function, but DDPG uses a slightly different update rule for the actor:

$$\nabla_\theta J \approx E_s[\nabla_a Q(s,a|w)\nabla_\theta \pi(s|\theta)]$$ [1, p. 3]

So the actor network directly acts on the gradients of the action-value function Q. The critic knows, in which direction Q changes beneficially, and this gradient is passed on to the actor. The loss function in the actor neural network is defined using these action value (Q value) gradients:

```
# Define loss function using action value (Q value) gradients
action_gradients = layers.Input(shape=(self.action_size,))
loss = K.mean(-action_gradients * actions)
```

„These gradients will need to be computed using the critic model, and fed in while training. Hence it is specified as part of the "inputs" used in the training function“[4]:

```
self.train_fn = K.function(
      inputs=[self.model.input, action_gradients, K.learning_phase()],
        outputs=[],
        updates=updates_op)
```

“The critic has a neural network, whose output is the Q-value for any given (state, action) pair. However, we also need to compute the gradient of this Q-value with respect to the corresponding action vector, needed for training the actor model. This step needs to be performed explicitly, and a separate function needs to be defined to provide access to these gradients:”[4]

```
# Compute action gradients (derivative of Q values w.r.t. to actions)
action_gradients = K.gradients(Q_values, actions)

# Define an additional function to fetch action gradients (to be used by actor model)
self.get_action_gradients = K.function(
        inputs=[*self.model.input, K.learning_phase()],
        outputs=action_gradients)
```

Wang et al. - Deep Q-trading show how to use Q-learning for trading decisions and they found that the accumulated wealth over n days in the past was a good candidate for a reward function. So this project will adopt this approach.

The training process follows the example of Stock prediction models and Teddy Koker's Blog entry; its parameters look as follows:

- •The window size determines, what temporal section of the values is considered by the agent; for example, the agent might only be trained on the last two weeks of closing values. The idea is to have a sliding window over the fund price time series.
- •The neural networks are trained with the same input data in a number of iterations, so that convergence is reached. If the total reward after each iteration does not change significantly anymore, the training can be stopped early.

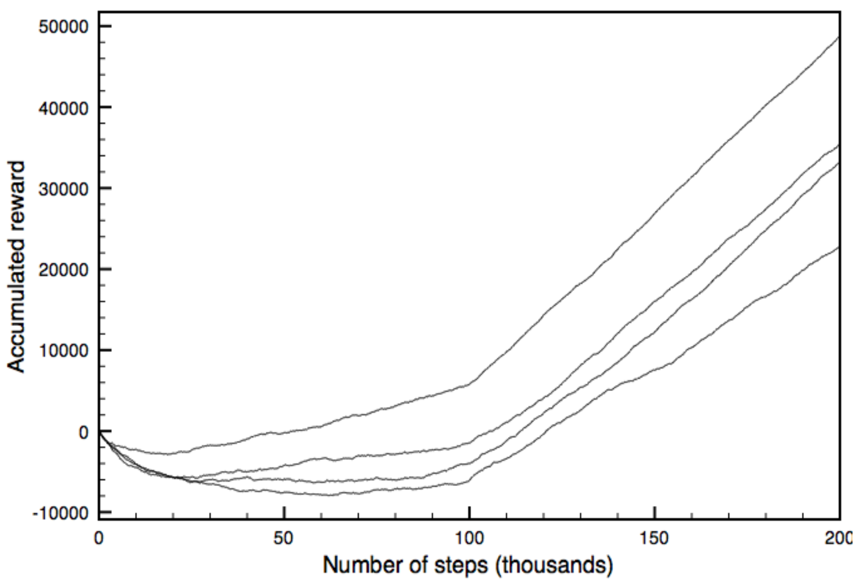
It must be noted that the action space is continuous as the agent can sell and buy real-valued amount of shares (under a budget constraint). The agent is supposed to detect/learn the more favorable action - that's the goal of reinforcement learning. So the agent might learn a highly imbalanced strategy, i.e. it might buy more than it might sell or hold. When the agent sees climbing share prices, it will be more inclined to buy stock, as it expects more reward from it. In a bull market, the actions will be imbalanced towards buying. The actions of the agent are part of the input training set of the critic neural network. To counter the effect of imbalanced and self-reinforcing strategies, two measures are taken:

- •a noise function is introduced, that leads the agent to try out random / exploratory behavior
- •the experience replay buffer is queried in a random fashion so that temporal correlations are broken.

Benchmark

The benchmark strategy for my proposed RL trading algorithm would be a passive dollar-cost averaging strategy (Link) in combination with a buy-and-hold strategy (Link). In essence, the investor buys each month investment products for a constant amount of money and then holds these products indefinitely. Such a strategy would typically be found with long-term investors building a retirement portfolio. At the end of the trading period, the strategy yields a portfolio of funds and the remaining budget, which can be added together to give the total value in €. This value is the benchmark for my DeepRL agent to beat; my agent will also produce a portfolio of funds with a total value. Wang et al. - Deep Q-trading also use a buy-and-hold strategy as a benchmark for their q-learning trading system. This strategy minimizes risk, but it may also overlook opportunities and does not take advantage of the market direction. The benchmark strategy is implemented in the same programming framework as the DeepRL agent and it is compared on the same data from the same financial products over the same time period.

In order to compare multiple strategies, the project follows the same approach as the quadcopter project and measures the learning process by tracking the development of the total reward. This is a visualization of the learning curve of the agent. Poole and Mackworth 2017 in their book on artificial intelligence also recommend this approach. They use a chart of the cumulative reward over time.



Exemplary Cumulative reward Plot

"One way to show the performance of a reinforcement learning algorithm is to plot the cumulative reward (the sum of all rewards received so far) as a function of the number of steps. One algorithm dominates another if its plot is consistently above the other. […] There are three statistics of this plot that are important:

- The asymptotic slope shows how good the policy is after the algorithm has stabilized.
- The minimum of the curve shows how much reward must be sacrificed before it starts to improve.
- The zero crossing shows how long it takes until the algorithm has recouped its cost of learning."

III. Methodology

Data Preprocessing

The financial data comes as CSV files, so we need to tell pandas how to read them: we define a column schema on the files, let Pandas parse the dates, and define the decimal separator.

The column "vol" is not a number, but an abbreviated string, e.g. "8.0K". We need to turn this string into a floating point number, so the "K" and "M" chars were replaced with 1000.0 and 1000000.0.

We are relying on the closing price of the ETFs, so we remove the records where the closing price has missing values.

Outliers are not removed, as there are no reasons to doubt the data.

As all three financial time series for S&P500, EuroSTOXX and MSCI World come with their own date column, we need to make sure that we use the same calendar and align on common dates. So I created a common calendar and joined all three data sets on that calendar. As there is no trading happening on weekends and public holidays, this join operation created a number of "empty" data records. I removed these data records in a separate step.

Implementation of the DeepRL agent

Task environment

The task environment (defined in task.py) defines the goal and provides feedback to the agent. In this project, the class Task can be seen as the bank or broker of the agent, i.e. a facility by which the agent interacts with the stock market and receives information from the stock market. The task environment also enforces the budget constraints (like a bank or a broker) and tracks the portfolio of the agent. The task also creates and tracks the agent state.

The following parameters are set in the task environment:

- Starting budget: 120000€
- Monthly budget: 1000.00€
- Transaction cost: 10.00€ per transaction
- Penalty for wrong actions: 5000.00€ per violation (the penalty is only added to the reward function and not deducted from the budget)

Please note: both agents (benchmark and DeepRL) see the identical task environment, so they play by the exact same rules. This goes for the test and training task environment!

Reward function

The reward function consist of three pieces:

- the total value created up to the current time step i.e. the current value of the portfolio plus the funds in the bank account.
- The penalties incurred for either wrongful selling or buying without funds. The agent needs to learn to stay within the financial constraints and to sell only what is available in the portfolio
- the transaction costs, i.e. the fees incurred for placing orders at a broker or stock exchange.

As I only want to track the value generated beyond the starting budget, I subtract it from the total value. The resulting figure is then transformed using np.tanh() in order to generate a nice number in [-1;1]. A scaling factor makes sure that the np.tanh() function is stretched, so that there is always a sufficient gradient.

Here is the resulting function:

```
totalvalue = self.get_total_value()
totalvalue = totalvalue - self.start_budget + penalties + tcosts
reward = np.tanh(self.rewardscale*totalvalue)
```

Neural Network Architecture

I will explain the DeepRL agent in further detail in the following paragraphs. The learning algorithm has the following tunable parameters:

- parameters of learning process: no. of episodes, runtime

- parameters of noise process: μ, σ, θ
- parameters of DDPG agent: γ, τ
- parameters of replay buffer: batch size, buffer size
- parameters of neural network: learning rate

Based on my experiences in the Udacity quadcopter project, I used the configuration suggested in [1]. Here are the parameters of the network model:

- parameters of learning process:
  - no. of episodes: 2000
  - parameters of noise process:
    - μ=0.0
    - σ=0.2
    - θ=0.15
- parameters of DDPG agent:
  - γ=0.99
  - τ=0.001
- parameters of replay buffer:
  - batch size=64
  - buffer size=100000
- parameters of neural network:
  - learning rate=0.01 for Adam optimizer.

The neural network architecture looks as follows:

- Actor:
  - layers: 1x input layer, 2x hidden dense layers, output layer
  - sizes: hidden layers with 400,300 units
  - ReLU as activation function for hidden layers
  - Sigmoid activation function for output layer (to encode actions)
  - After each hidden layer, I introduced a BatchNormalization layer. Each hidden layer featured a kernel regularizer to reign in model complexity.
  - no dropout layers were used, as they did not prove helpful.
- Critic: the neural network looks identical to the actor network, except for the actions/states split.
- The replay buffer was set at 1*10^6 elements. This setting is also based on [1].

Problems in the implementation:

I encountered the following problems in the implementation:

- A design issue was how to define the permissible limits on the actions of the agent (i.e. task.action_low, task.action_high in task.py)? The DDPG algorithm is dependent on these two limits: we need a fixed interval, as the DDPG agent outputs a number in [0..1], which is then mapped on the interval [$\text{action}_{\text{low}}$...$\text{action}_{\text{high}}$]. In theory, the agent could sell and buy much more – as an action, it could sell every share in the portfolio, and it could buy as many shares as possible within its (monthly) budget constraints. As a solution, I set reasonable default values for [$\text{action}_{\text{low}}$...$\text{action}_{\text{high}}$], which fit in the budget constraints and are not too constraining for the agent.
- Q-Learning is brittle. For some runs, the agent would no explore enough and thus remain in the starting state for episodes. On the second run, the agent would just explore fine.

- As the task environment uses penalties to enforce the budget limits, it is very important for an effective agent to stay within these limits; if the agent goes beyond, the penalties get added to the reward function and in some cases, the action cannot be executed. To make it easier for the agent, I added the currently available total and monthly budget to the agent state, so that it can always use it for learning. Nonetheless, these limits and avoiding penalties makes the learning task even harder.

Refinement

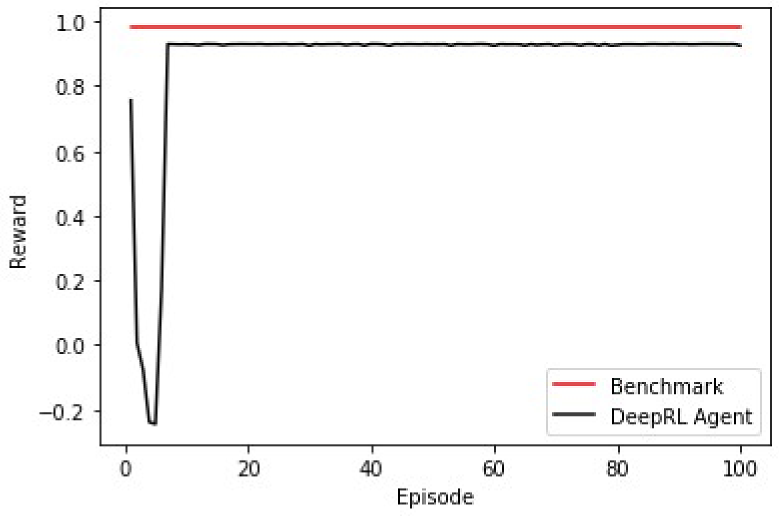
- Tweaking the reward function was one of the main refinement activities that I had to do in this project. According to the project proposal, I added penalties to the reward function for actions that are not executable, like selling stocks that are not in the portfolio. It took some training runs to tune the penalties to the right level (set at -5000.00€ now)
- After training the DeepRL agent for a while, I could see that it had a hard time trading three stocks at the same time, so I changed the code to make the agent focus on one ETF to ease the complex learning process. This refinement is just fair, as the benchmark agent also only buys one stock type (S&P 500).

The introduction of penalties and transaction costs lead to a reduced trading activity by the DeepRL agent; before, it traded wildly and relied on the task environment to correct or reject invalid trading actions. After the refinement, the agent traded more carefully and avoided trading at all under some conditions. This refinement was necessary to foster learning.

The limitation to one tradable ETF also led to a better learning behavior. Before, the DeepRL agent wildly combined trades. The possibility to trade three ETFs in parallel added an allocation decision with combinatorical complexity into the learning task. After the refinement, the agent only needs to decide on the level and timing of investment for one ETF, which is a simpler task, so the learning progress was more stable, and the actions were more purposeful.

IV. Results

Model Evaluation and Validation



Above you will see a chart with two lines, which show the rewards plot for the two agents over the course of 100 training episodes.

The benchmark agent is still ahead, but the DeepRL agent took about 10 episodes for exploration and then it exhibited a very similar performance in terms of cumulative rewards. So the final performance is on par with an established investment approach.

The transaction costs and the penalties are added in each step as needed, but they are not cumulative (please see definition of the reward function above). The reward function allows for a fair comparison of both agents, as their performance and learning progress is measured by the same reward function. If the benchmark agent behaves stupidly and violates budget constraints, it will suffer from the same penalties that the DeepRL agent would get under the same conditions. If the DeepRL agent exactly copied the trading actions of the benchmark agent, it would receive the identical rewards. Both agents act within the same task environment and the task environment determines the penalties and rewards. Nota bene: In the above scenario, the benchmark agent's settings are fine-tuned to avoid penalties, while the DeepRL agent has to painfully learn to avoid those. So, the DeepRL agent is likely to perform worse in the beginning.

Justification

Here is the comparison of the training and test result of the two agents:

Agent	Phase	Time steps	No. of Shares in Portfolio	Total Value in €	Total reward
Benchmark	Training (best run)	1893	4541.35	165736.63€	0.98
	Test	631	722.80	122496.54€	0.13
DeepRL Agent	Training (best run)	1893	3661.29	157564.00€	0.93
	Test	631	509.72	122370.63€	-0.13

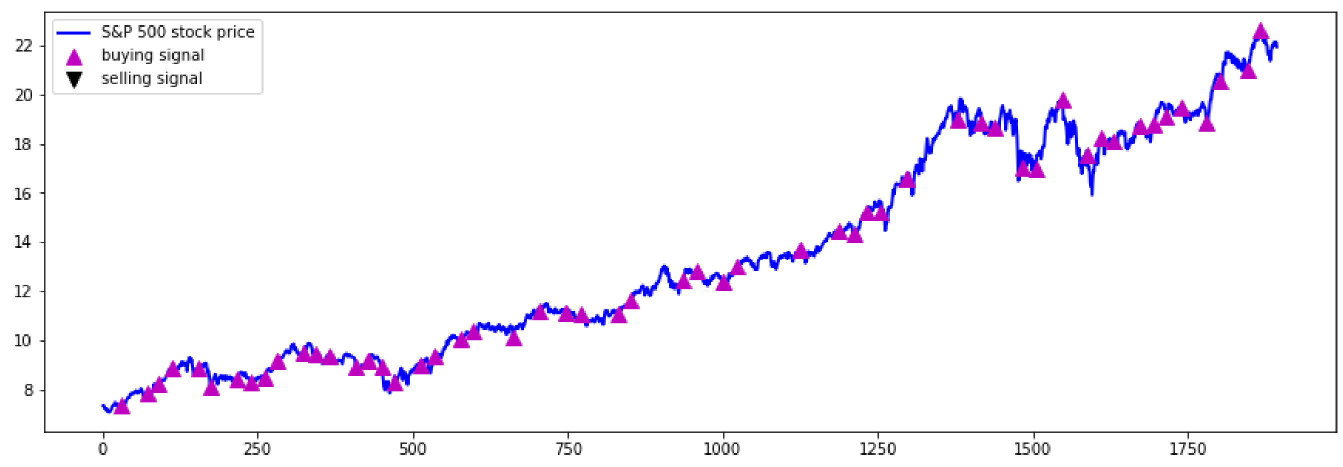
It is remarkable that the DeepRL agent showed an impressive test performance on data it has never seen before, only after 100 training episodes, although it has a pretty low total reward. This results is likely caused by a lot of penalties and transaction costs, that it gathered during the test phase. So its behavior can definitely still be optimized.

V. Conclusion

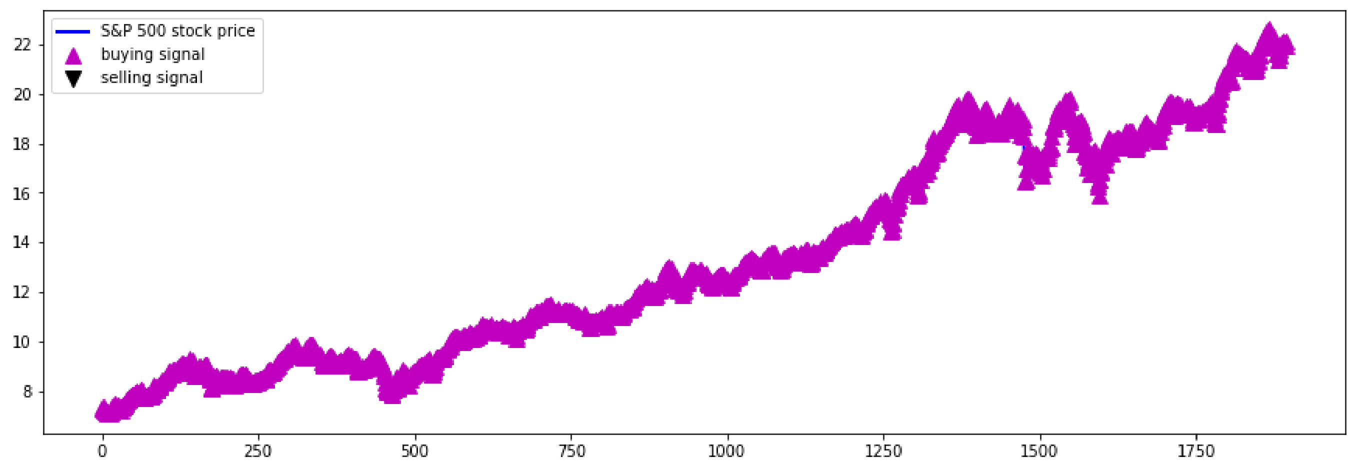
Free-Form Visualization

The two following charts show the trading actions during the best training episode of each agent. The x-axis is the time period and the y-axis is the stock price of the ETF. The violet markers indicate a buying action of a certain amount and the black marker indicate a selling action.

Here is the chart for the benchmark agent:



Here is the chart for the DeepRL agent:



It is remarkable that the DeepRL agent trades at almost every occasion and hence decides to suffer the consequences of penalties and transaction costs. However, this strategy better tracks the stock price of the ETF and produces a more stable return by using a higher sampling frequency of the stock market. But of course, there is probably some optimization potential by avoiding penalties and transaction costs.

Reflection

For the initial development of the DeepRL agent, the following workflow was applied:

1. manual data retrieval from the Internet
2. implement data loading, cleansing and pre-processing using Python pandas package
3. create test and training data set (in pandas)
4. implement DDPG agent based on Python source code in https://github.com/udacity/RL-Quadcopter-2

5. train the agent using the training data set and manually tweak network hyperparameters, network architecture and reward function
6. test the agent using the test data set and build visualization of the results (using pandas in a Jupyter notebook)

The hardest part of the project was to implement the two agents in the agent framework that was put forward in the Udacity quadcopter project. The DDPG agent source code has certain requirements concerning data structures which have to be met. Furthermore, the two agents need to exhibit the same class methods, so that a generic training framework can be used.

So far, the DDPG-based agent exhibits learning behavior, but there were several instances during the training and during the test where the agent showed non-explainable behavior (e.g. not trading for extended periods of time, or trading all the time). This happened despite the tweaking effort. The learning behavior was not stable during the training phase.

What I found amazing, was the obvious learning behavior that the agent exposed: the log files showed how the agent behaved more intelligently from episode to episode, as if it really understood how to improve. First, it started out with an erratic buying/selling behavior, but then it settled usually on one ETF, in which it invested continually, which is sensible given the transaction fees.

Improvement

One major improvement might be the usage of a more deterministic learning approach. Right now, we have a lot of parameters, a noise function and several complex neural networks which implement the learning algorithm. This system is powerful, but hard to understand and debug. Improvements are achieved by trial-and-error and not by careful engineering. So I would suggest to build a simpler model and study its learning progress. This might yield a better insight into the determinants of a good learning algorithm.

Another improvement would be possible for the benchmark agent; currently, it only implements a very simple buy-and-hold logic. This behavior mimics human behavior, but it could pay more attention to market trends and macroeconomic data to make more intelligent decisions.

A possible extension of the described proposal includes the automation of the workflow, such that the stock data is picked up daily, and the agent writes out the trading decision regularly to a file or a database. Another extension could be the automated retraining of the deep network models, e.g. every month. The retraining would ensure that the agent catches up on current trends of the stock market. Wikifolio, a German company, uses a social media platform to share investment portfolios of platform members and offers the chance to investors to put their money in one of these portfolios. If a trading agent was successful, it could be used there as well.

References

[1] Lillicrap, Timothy P., et al., 2015. Continuous Control with Deep Reinforcement Learning on page 11 (Link to PDF file).

[2] Sutton, Richard S. and Barto, Andrew G. 2018. Reinforcement learning – An introduction. 2nd ed. MIT Press.

[3] Udacity Nanodegree, Section Deep Q-Learning.

[4] Udacity Nanodegree, Section Reinforcement Learning.