Developing a Self-Learning Trading Agent using Reinforcement Learning

Capstone Project

Machine Learning Engineer Nanodegree

Ralph Fehrer, October 30th, 2016

1. Definition

1.1. Project Overview

The goal of this project is to develop a self-learning trading agent which uses *Q-learning* as a specific *reinforcement learning* technique to make autonomous trading decisions, based on market states and expected wins/losses. The evaluation of this *Q-learning trader* is focused on two complementary questions:

Is a Q-learning trading agent able to learn a profitable trading policy in a simplified day trading setting? And can a human trader learn anything from the policy the agent has learned?

The agent developed in this project can be considered as an artificial technical day trader, that tries to exploit intraday trends to make small, profitable trades several times per day, with the goal of maximizing the net profit over many trading days.
Due to the highly complex nature of market data, the lack of reproducible benchmarks and since reinforcement learning is still an active research area, it would be unreasonable to expect a profitable automated day-trading system as a result of this project. The presented approach should instead rather be considered as an experimental setting, analyzing one possible application of reinforcement learning to the trading domain.

1.2. Problem Statement

The Q-learning trading agent is provided with market state data several times per trading day by a market environment. Based on this market state and previous experience which is represented by a *Q-table*, the trading agent autonomously makes a trading decision. The portfolio value resulting from this trading decision is then used to update the *Q-table* in a learning step.

Based on this general description, the following major- and sub-elements can be specified:

1. (Market) environment
- Market data component that provides historical price data
- Technical analysis component to calculate the market state based on the historical price data
- Order component which allows the agent to enter and close positions
- Position component to calculate the current value (or reward) of a position

2. Trading agent

- *Q-table* mapping each possible state/action pair to a value that represents the expected win or loss if the agent takes action *a* in state *s*
- Logic which selects a trading action from a set of allowed actions at every update step
- Algorithm to update the *Q-table* based on the current state, the chosen trading action and the portfolio value

In order to simplify the complex task of developing a *Q-learning* trading agent, several constraints are specified: The trading agent acts in a day trading setting, i.e. any open position is automatically closed at the end of a trading session. Only one asset is traded with position sizes of 100% of the portfolio value, either long or short. The state transition at any update step is exclusively determined by the *position* sub state, according to the transition chart in figure 1.

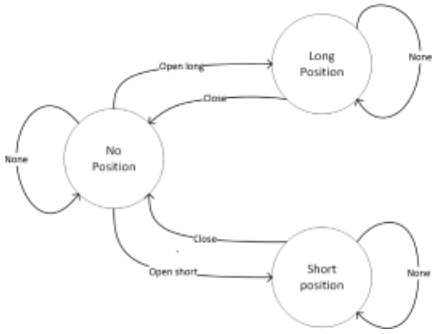

Figure 1: Trading state transitions

It is important to emphasize that the implemented trading approach is completely unhedged and - in consequence of the 100% shorted positions - highly vulnerable to extreme losses. However, as stated above, it is not the intention to develop a profitable day-trading strategy in this project, but to investigate the application of *Q-learning* to trading in a simplified experimental setting.

1.3. Metrics

The key metric to measure the performance of the self-learning trading agent is the total reward, i.e. the position values accumulated over all update steps over all trading days. The total reward achieved by the *Q-learning agent* is then compared to the performance of an agent that just randomly picks an action at every update step without learning anything.

A second type of metrics used in this project is represented by the three technical indicators from which the market state is derived at each update step.
The first technical indicator is the Relative *Strength Index (RSI)*, a *momentum oscillator* measuring the velocity and magnitude of directional price movements.
The second indicator is the *Moving Average Convergence/Divergence* (*MACD*), consisting of three time series calculated from historical price data: The actual *MACD*, a signal series and a divergence series, which is the difference between the first two. The *MACD* itself is the difference between a fast and a slow *exponential moving average* and the *signal* series is an *exponential moving average* of the *MACD*.

As a third indicator *Bollinger Bands* are used, which consist of an n-period moving average (MA), an upper band at k times an n-period standard deviation above the moving average ($MA + k\sigma$) and a lower band at k times an n-period standard deviation below the moving average ($MA - k\sigma$).
It should be noted that the choice of indicators is mainly based on their relatively frequent usage in trend-based day trading systems. The set of indicators, just as the constant state derived from them are more or less arbitrary design decisions and would be a good target for experimentation.

2. Analysis

2.1. Data Exploration

The market data used in this project is provided by *Quantopian*, an open web-platform for (aspiring) Quants (https://www.quantopian.com/?signout=true). According to their contract with the original data vendors, the marked data can only be obtained and used within the *Quantopian* research environment. The data used in this project is *SPY* close data, with a resolution of 1 minute, for the one year period between Febuary 2015 and February 2016.

2.2. Exploratory Visualization

The *SPY* price data is obtained through the *get_pricing* API method in the *Quantopian* research environment. This function returns a *pandas.Panel*-object with columns depending on the requested data and a datetime-index (For more details, see https://www.quantopian.com/help). The price data is then fed to several *talib* functions, to calculate the *MACD*, *RSI* and *Bollinger Bands* indicators (https://mrjbq7.github.io/ta-lib).
The following table contains a quick summary of the *SPY* and indicator data.

	count	mean	std	min	max
SPY	98310	204.457093	7.690784	181.040000	213.720000
macd_raw	98277	-0.001262	0.117318	-3.628914	1.049916
macd_signal	98277	-0.001258	0.110243	-2.948773	0.905086
macd_hist	98277	-0.000004	0.035887	-1.581692	0.701948
bb_upper	98299	204.650753	7.615652	181.560667	213.833976
bb_middle	98299	204.457503	7.689022	181.307492	213.626250
bb_lower	98299	204.264254	7.768967	177.773093	213.510606
rsi	98303	50.263723	16.317072	1.290005	98.380043

For a quick plausibility check of the data count, the average number of trading days per year (~251) can be multiplied by the number of minutes per trading day, which is about 390. The result of 97890 expected data points is close enough to the actual count of 98310.
The difference between the amount of price data and indicator data can be explained by the fact that all indicators involve the calculation of *moving averages*, which require a certain amount of historic price data that is not available at the beginning of a time window.
The min and max values of the upper and lower *Bollinger Bands* vary around the mean of the price data, which is consistent with their mathematical definition of a *k-times n-period standard deviation above/below* the prices moving average.
As a percentile metric, the *RSI* moves in a range between 0 and 100, where the actual min/max values of 1.29/98.38 represent extreme conditions which could be regarded as potential trading signals. The

median of 50.26 is almost perfectly centered in the middle of the *RSI* range and confirms its character as an *oscillator*.

Figure 2 combines price and indicator time series for one particular day in a single plot.

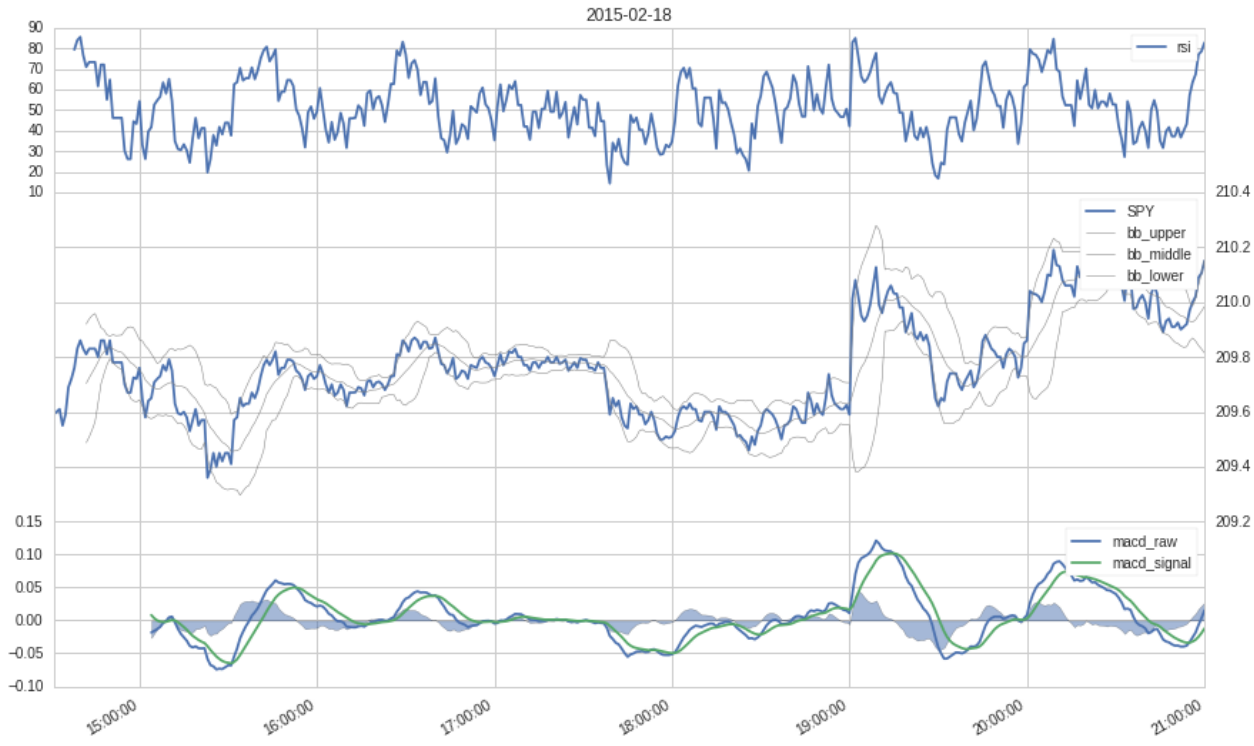

Figure 2: price and indicators

The *RSI* is often used to detect potential trend changes, where a value above 70 can be interpreted as *overbought* and a value below 30 as *oversold,* signaling imminent trend changes. Figure 2 contains several overbought and oversold signals.
The *MACD* consists of the MACD and the signal series and a filled bar curve, representing the difference between these two. A crossing of the *signal* and the *MACD* may indicate a trend reversal, where a positive difference could be an indication of an up- and a negative difference an indication of a down trend. The absolute value of the *MACD* is sometimes used as trend confirmation signal, such that long trades are only entered when this value is equal to or greater than zero.
Other trend confirmation signals can be derived from the Bollinger bands, such as the slopes of the upper and lower bands. The breakout of the price above the upper or below the lower band may be interpreted as a trend reversal signal.

Based on this discussion, the market state presented to the *Q-Learning Trader* is defined as follows:

complete_state=
['rsi>70','rsi<30','macd_hist_positive','macd_hist_negative','upper_bband_up','lower_bband_down']

It should be noted, that these indicator signals are used and combined in many different ways in various trading systems. The constructed state is therefore only an initial suggestion and it is part of this project to see, if the agent is able to autonomously detect profitable trading opportunities based on it.

2.3. Algorithms and Techniques

Reinforcement learning deals with a subset of a more general type of decision problems known as *Markov decision processes (MDPs)*.

MDPs are defined by the *5-tuple*

$$(S, A, P_a(s, s'), R_a(s, s'), \gamma)$$

Where

- S is a finite set of states
- A is a finite set of actions
- $P_a(s, s')$ is the probability that $s' \epsilon S$ is the next state if action $a \epsilon S$ is taken in state $s \epsilon S$
- $R_a(s, s')$ is the immediate reward after transition from s to s'
- γ is a discount factor regulating the tradeoff between immediate rewards and future rewards
-

Reinforcement Learning (RL) tries to solve *MDPs* whose transitions probabilities and / or rewards are not known in advance. At the core of each *RL* algorithm is some type of *value function* mapping states or state/action pairs to numerical values, which quantify the advantage of being in a specific state, or of taking a specific action in a specific state. This *value function* is constantly updated based on the rewards an *RL*-agent receives during learning and represents the agent's *policy*.

Q-learning is a special *RL* – technique using a table to represent the *value function* $Q(s, a)$, where the value in each cell quantifies the expected reward for taking action a in state s. The table-based representation makes *Q-learning* a particularly transparent and comprehensible approach and lends itself nicely to a more detailed inspection of the learned policy, which is one of the intensions of this project.

The general algorithm for updating and learning $Q(s, a)$ can be stated as follows:

Initialize $Q(s, a), \forall s \in S, \forall a \in A(s)$ *arbitrarily, and* $Q(terminal) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)$
    $S \leftarrow S'$
  until S is terminal

Where

- An *episode* is one repetition of the same learning task, starting in a specified start state and ending in a terminal state
- *ε-greedy* is a method to control the exploration/exploitation tradeoff
- α is the learning rate which can be set to any value between 0 and 1 and controls the step size in the direction of the gradient at any update step.
- γ is the *discount factor* which can be set to a value between 0 and 1. A value of 0 leads to an extremely myopic agent focusing only on the immediate reward to a current action, while a larger value also discounts future rewards to the results of the current update step.

The general algorithm was slightly adapted due to some specifics of the problem domain and restriction of the environment. The version implemented for this project can be stated as follows:

Repeat (for each trading day until 30 minutes before trading session end):
  prevS ← None, prevA ← None
  Repeat (each minute):
    R ← Reward
    S ← State
    If S not in Q-table, Initialize $Q(S, a), \forall a \in A(S)$
    Choose A from S using policy derived from Q (ε-greedy)
    Take action A
    If previousState is not None and prevAction is not None
      $Q(prevS, prevA) \leftarrow Q(prevS, prevA) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)$

The best value combination for α and γ was determined by a grid search algorithm (see refinement and evaluation). An *ε-greedy* policy was realized by implementing the *choose_action* method such that – statistically - at every tenth update step instead of selecting the action that maximizes $Q(S, a)$ for the current state S, an action is just uniformly drawn from the set of allowed actions $A(S)$. This approach ensures that the agent keeps exploring even in later episodes, which intuitively seems to be the right choice given the highly dynamic and insecure market environment.

2.4. Benchmark

Main purpose of this project is to answer the question, if a *Q-learning* trading agent is able to learn a trading policy in a simplified day-trading setting. The performance of the learning agent is therefore not compared to a common market benchmark, but to a *monkey trader* which by design cannot learn anything at all. This *monkey trader* acts in exactly the same environment and follows the same state transition rules as the *Q-learning* agent. The only difference is, that the *monkey trader* at any update step just uniformly draws from the set of allowed actions.

3. Methodology

3.1. Data Preprocessing

Depending on the time window, the price time series from which the indicator time series are calculated may contain some non-numerical values (*NaN*s). Most of the implementations used in this project are robust towards *NaN*s but it could be observed, that the *Quantopian* algorithm runtime crashes in some cases due *NaNs*. This issue was reported to *Quantopian* and is currently investigated. To be on the safe side, price data is sanitized by using a *backfill* policy for *NaNs*, such that any *NaN* is replaced by the next valid numerical value in the time series. For a large time series consisting of minute resolution price data with some isolated *NaNs*, the impact of this rather coarse policy is negligible.
While the *NaN*s contained in the price data series are random and potentially dangerous, *NaN*s in the indicator time series are systematic: All indicators use moving averages, which require a certain amount of historical data for their calculation. It is therefore not only expected but also correct that the first entries in the indicator time series consist of *NaN*s.

3.2. Environment implementation

The market environment was implemented using *API*-method provided by the *Quantopian* research environment, which is based on the *IPython Notebook* patform.
The central function of the market environment is *handle_data* , which is called every minute of a trading day by the *Quantopian* trading algorithm engine. In this method, the most recent historical data is retrieved and passed to the *talib's* implementations of the *MACD*, *RSI* and *Bollinger Bands* indicators. Based on the indicators values the state is calculated and passed to the agent. In a last step, the market and indicator data is saved in a *pandas.DataFrame* for later analysis.

To facilitate experimentation with different *reinforcement learning* techniques and to encourage further research, a base abstraction was implemented for self-learning trading agents interacting with the *Quantopian* environment. This base abstraction defines the main learning sequence comprising reward retrieval, action selection and value function update and implements the main environment-dependent functionality, such as *get_reward* and *act*. The implementation of the concrete learning algorithms is left to specific agent classes deriving from the base abstraction.
The class diagram in figure 3 illustrates the most important aspects of the agent's class design.

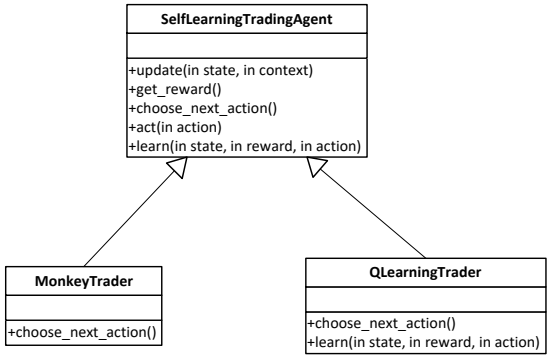

Figure 3: Trading agents class design

Agent and environment are coupled by the agent's *update* method, which is invoked in the *handle_data* function. Figure 3 illustrates the main update sequence executed each minute during a trading day.

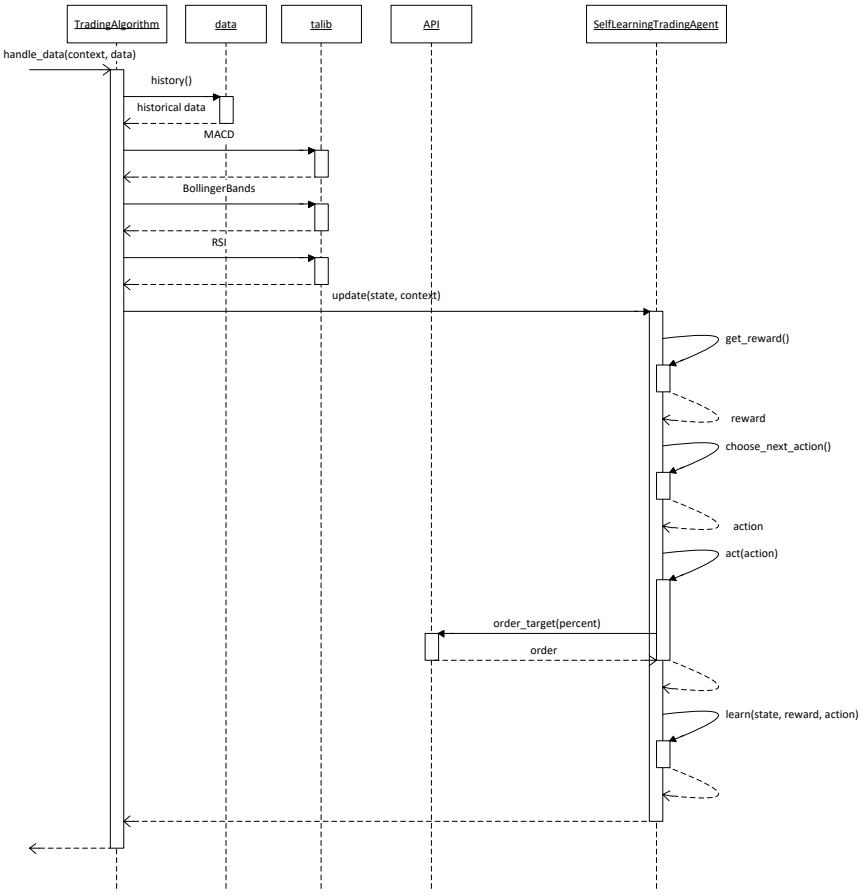

Figure 4: Main update sequence

3.3. Refinement

The implementation was refined in a two-step optimization process. The first and relatively coarse step consisted in iterating through the *power set* of the complete state definition, to find the subset of states producing the best results. In a second step, the *Q-learning* parameters α and γ where fine-tuned by a grid search algorithm, to find the best parameter combination based on the set of states found in step one.

4. Results

4.1. Model Evaluation and Validation

The *MonkeyTrader* and the *QLearningTrader* were both run with exactly the same historical data, to compare their performance. Figure 3 illustrates the *MonkeyTrader's* results for the first and the final trading days.

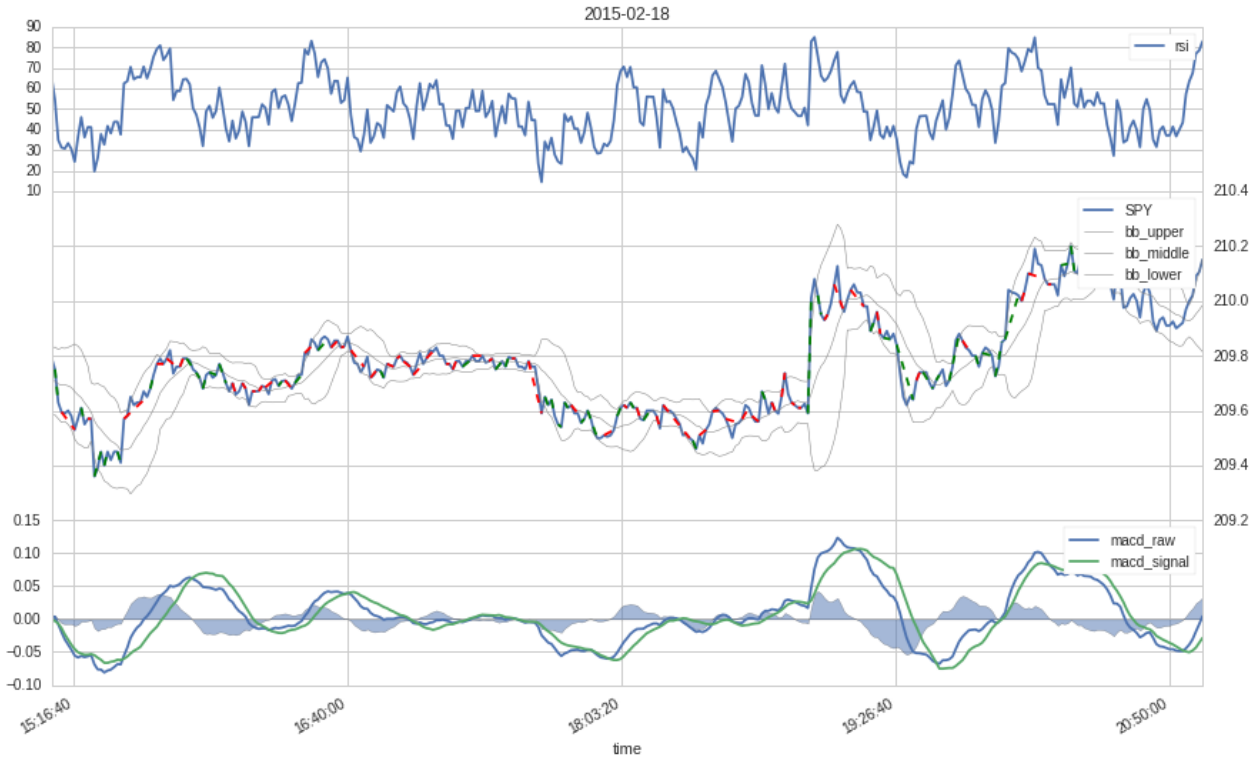

Figure 5: Monkey trader, 1st trading day

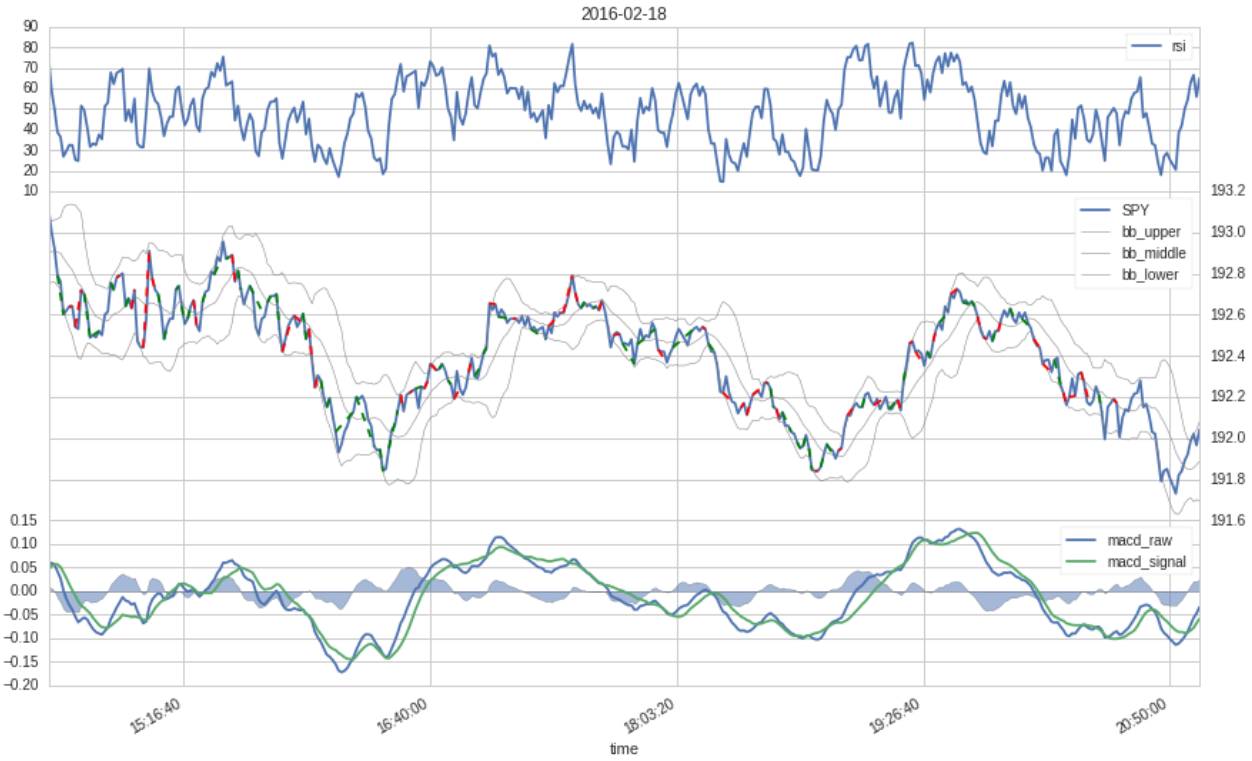

Figure 6: Monkey trader, 2nd trading day

The realized trades are represented by dashed lines, were red indicates a losing and green a wining trade.

The *monkey trader* obviously prefers short term trades, which can be explained by the fact that an existing position is closed with a probability of 0.5 at every update step. The trading behavior of the *monkey trader* seems to remain consistent throughout the complete time window and with a total reward of -787.69954172 it loses a considerable amount of money. Both observations can be expected from a trading agent that cannot learn anything.

The setting for the *Q-learning* trader was optimized in a two-step process, where the first step consisted in an iteration through the power set of the complete state definition, to find the best state combination. The results for the best state combination are summarized in the following table:

Best state combination	**Total reward benchmark** $(\boldsymbol{b})$	**Total reward QLearner** $(\boldsymbol{q})$	$\left(\frac{\boldsymbol{q}-\boldsymbol{b}}{\lvert\boldsymbol{b}\rvert}\right) * \mathbf{100}\%$
rsi>70, lower_bband_down	-787.69	-39.87	95%

An improvement of almost 100% is already a strong indication for the ability of the *Q-Learning* trader to learn a trading policy that is significantly better than the completely random policy of the *monkey trader* benchmark.

Based on the best combination of states, the α and γ parameter of the *Q-learning* trader were fine-tuned in a grid search process. The final results are reported in the table below.

Best parameter combination for best state combination	**Total reward benchmark** $(\boldsymbol{b})$	**Total reward QLearner** $(\boldsymbol{q})$	$\left(\frac{\boldsymbol{q}-\boldsymbol{b}}{\lvert\boldsymbol{b}\rvert}\right) * \mathbf{100\%}$
α=0.9, γ=0.7	-787.69	9.22	101%

The results show that the performance of the *Q-learning* trader could be further improved by a fine-tuning of the two central parameters.

A visual analysis of the *Q-learning* agent's trading behavior may provide more information about the learned trading policy. The following two figures illustrate the first and the last trading days.

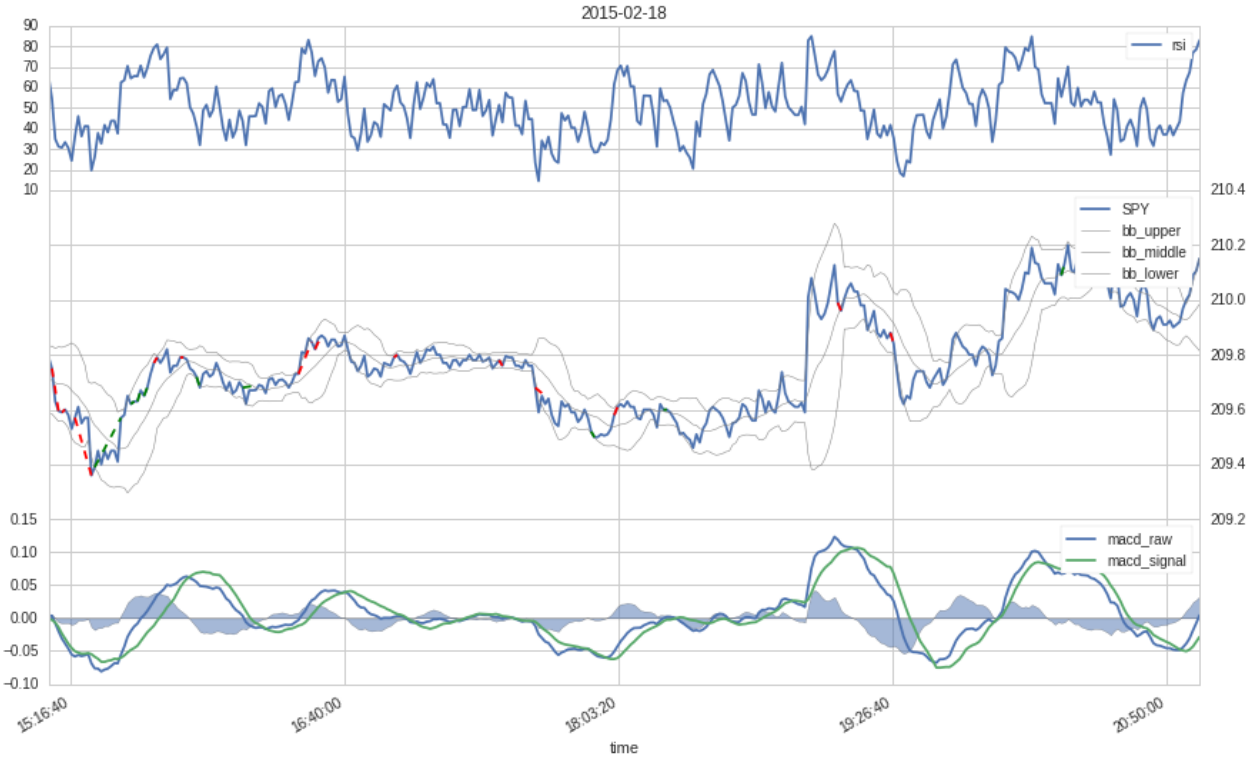

Figure 7: Q-learning trader, 1st trading day

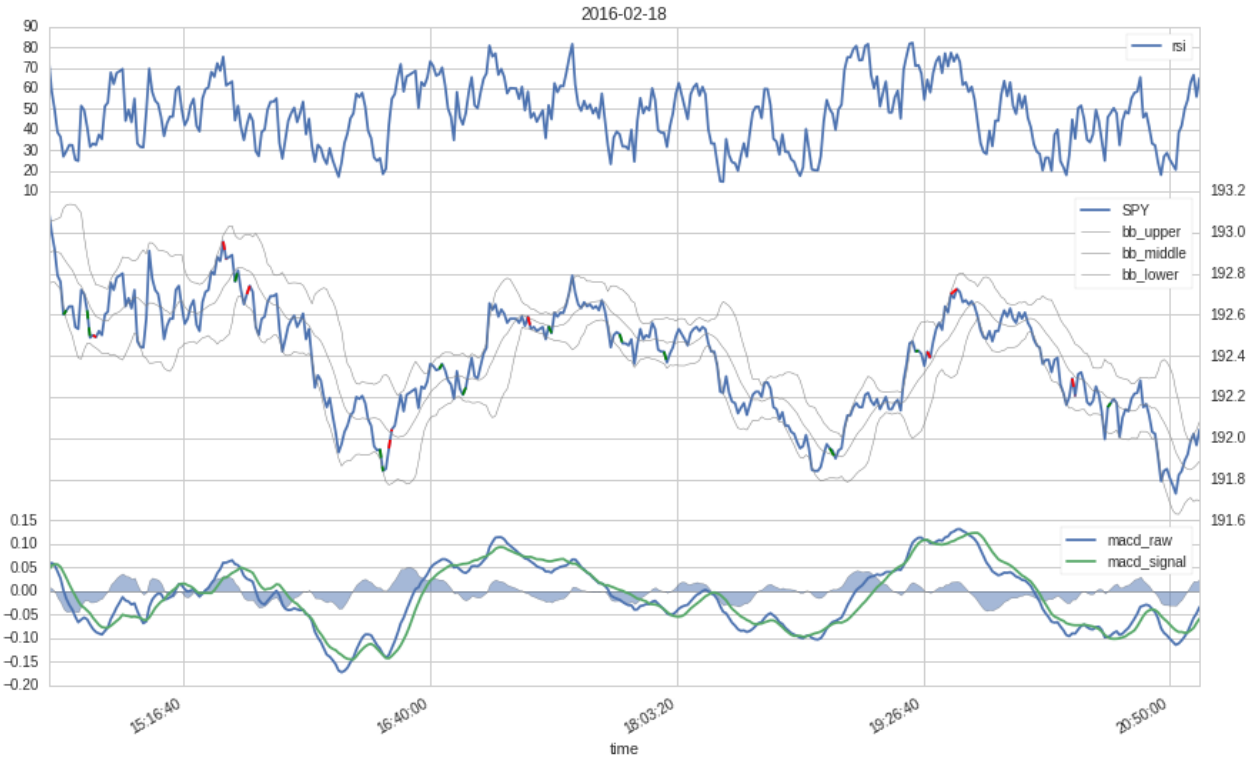

Figure 8: Q-learning trader, 2nd trading day

A direct comparison of the *Q-learning* and the *monkey trader* shows, that the *Q-learning* agent trades much less than the *monkey trader*. The trade length is however comparable, which is a little surprising given the relatively high value of γ.

4.2. Justification

A day trading setting such as the one used in this project lends itself nicely to a Q-learning implementation: the episodes are naturally defined by the individual days, state and reward can easily be derived from an existing market environment and the actions are limited to a small set of trade types. The *Q-learning* trader implemented in this project, represents one possible approach and performed considerably better than the benchmark. Although the setup and the absolute results may not justify the use of the *Q-learning* as a day trading system, the implementation provides ample opportunity for further experimentation and research.

5. Conclusion

5.1. Free-Form Visualization

The *Q-table* of the *Q-learning* trader represents its trading policy. A more detailed inspection of the *q-table* may therefore help to gain a deeper insight into what the *Q-learning* trader has actually learned. Figure 9 illustrates the *Q-table* in a heatmap representation.

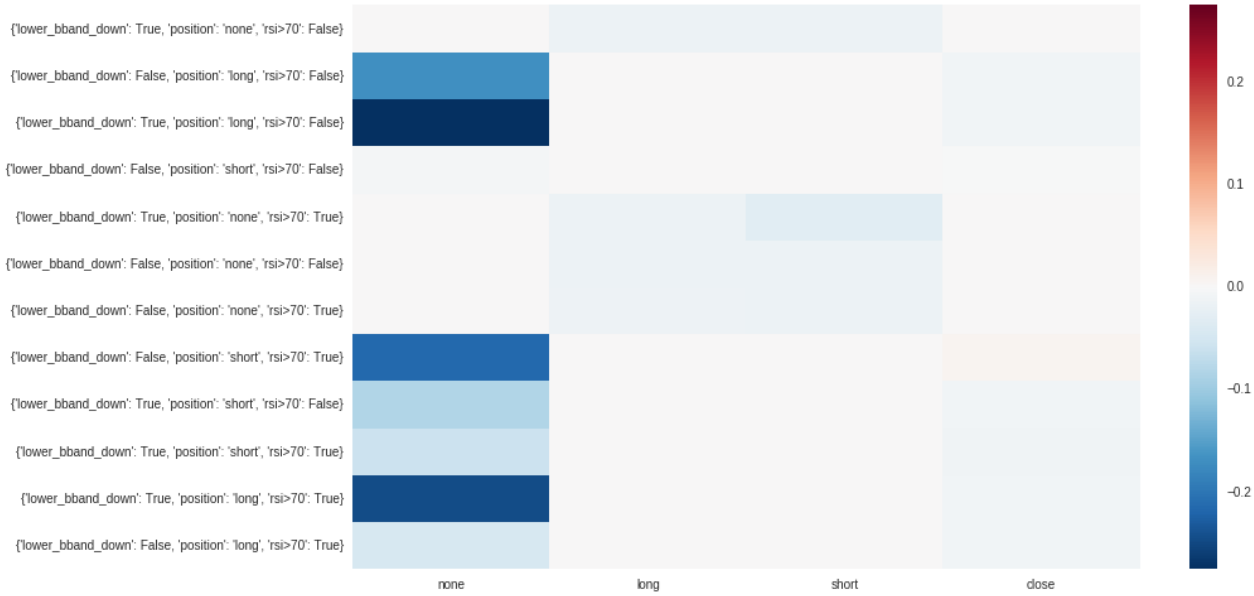

Figure 9: Heatmap representation of the Q-table

The heatmap of the *Q-table* reveals, that the signal level is generally very low. This is consistent with the short-term trades favored by the agent, but may also be attributed to the generally inconsistent nature of market signals.

Another interesting observation is that almost all state/action pairs are mapped to values ≤ 0 . In terms of a policy, this means that the agent was not able to really identify profitable setups based on the states and achieves its performance mostly by avoiding trades. The only state/action pair with a slightly positive value is *lower_bband_down=False,position=short,rsi>70=True / close*, but the value of 0.01 cannot be considered a strong signal.

The highest absolute values were assigned to pairs containing the *none* action. One possible explanation for this can be found in the design of the state machine, which allows the *none* action in all states, while the other three actions are only allowed in some states. The *none* action is thus selected more often than the others and gets more updates. The higher values for state/action pairs that are selected more often may also imply, that the general signal level could be increased by running the algorithm on more episodes.

5.2. Reflection

The task of developing a reinforcement learning trading agent posed several interesting challenges.
The first difficulty was to find related work, that could be used for benchmarking and/or orientation. Several papers exist applying different approaches of reinforcement learning to the trading and investment domain. However, the solutions and algorithms introduced by these papers vary widely and are often rather involved and difficult to reproduce. The design and implementation for this project was therefore largely developed from scratch, using some code snippets from a previous project.
Another difficult task was to find a good market environment with sufficient and reliable market data and with a simulation and backtesting infrastructure to run the *Q-learning* algorithm over a large number of episodes. Although the *Quantopian* research environment proved to be a very good choice in this regard, several aspects had to be adapted and extended, such as the plotting functionality to illustrate and analyze the trades.

5.3. Improvement

There are several potential optimization targets in the *Q-learning* algorithm and the market environment.
One obvious issue is the preference for short-term trades and the correlated low signal level of the *Q-table*. Longer trades and potentially higher signals may be achieved by tweaking the reward system, such that e.g. the rewards realized by a *close* action receive a higher weight. This would at the same time lead to a more realistic scenario, since the realized wins or losses are what really matters in day trading.
Another potential optimization target is the state space. The state used in this project was based on a completely arbitrary design choice and experienced traders would very likely work with different indicators and signal combinations. Other types of metrics could be factored in, such as *fundamental* data or statistical features. The only two constraints for state constructions are that the state has to be discrete and that the state/action space must not be too large, to avoid the *curse of dimensionality*.