Reinforcement learning for markets

Ayush Garg, Virgile Troude, Martin Fiebig, Simon Hasenfratz

ETH Zurich

December 6, 2019

Abstract

We have implemented a market setting with agents that participate as buyers and sellers in a double auction game. In every round of the game, buyers can bid a price b_i for a given good, while sellers can offer a price a_j for the same good. Agents are assumed to have some additional information to their avail in every time step, which helps them to determine an optimal bid or ask price and improve over time. In every round, a given matching mechanism tries to match potentials buyers with sellers at a price $d_{j,i}$. Agents that are matched will leave the game thereafter, while the remaining agents bid for the next round, always trying to leverage the available information by maximizing their Q-values. In this report we show how agents improve their returns over time by playing the game and learning by reinforcement. Furthermore, we illustrate that factors like opportunity costs or supply and demand are driving market forces and describe the outcome of various experiments with different market settings and reward functions.

Contents

1 Introduction

In this project we have implemented a market setting with agents that participate as buyers and sellers in a double auction game. In every round of the game, buyers can bid a price b_i for a unit of a given good, while sellers can offer a price a_j for a unit of the same good. Agents are assumed to behave rationally, meaning that their possible bid and ask prices are bound to constraints like budget or production costs. On top of that, agents are assumed to have some additional information to their avail in every time step. This information helps them to determine an optimal bid or ask price and improve over time. In every round, a given matching mechanism tries to match potentials buyers with sellers at a price $d_{j,i}$. Agents that are matched will leave the game thereafter, while the remaining agents bid for the next round, always trying to leverage the available information by maximizing their Q-values.

The report is structured as follows. Section 2 gives a brief overview of the basic market setting we have implemented, describing agents, the information setting, and the matching mechanism. In section 3 we introduce the principles of reinforcement learning and present our Q-learning mechanism. In section 4 we show how agents can improve their rewards over time by learning from playing the game, and we test different market settings and reward functions to extend our basic model. After the conclusion, we have appended some of the most relevant excerpts from the code we have used.

2 Market setting

2.1 Agent setting

The agents are modelled in a way that they all start with a random bid and ask price generated by $\mathcal{N}(100, 5)$. From the second round on, the agents start to increase or decrease their bids and offers, by taking the action with the maximal Q-value (see Section 3). We assume that agents can't arbitrarily change their bids and asks from one round to the next, but that they can only adjust their prices in discrete steps s by either increasing or decreasing their previous price by $k \cdot s$ for some $k \in N$. Furthermore, we incentivize agents to trade by assigning them a negative reward for every round they stay in the game. Our initial set of parameters- referred to as the basic setting- can be found below:

2.2 Information setting

We assume that each agent knows all the offers from the agents of different role that happened in the previous step. For any buyer $i \in I$ and any seller $j \in J$ we therefore assume that the observed information $o_{*,t}$ is

$$o_{i,t} = \{a_{j,t-1}\}_{j \in J}$$

and

$$o_{j,t} = \{b_{i,t-1}\}_{i \in I}$$

This situation is known as opposite side information setting. Based on this information we calculate Q-values for every agent in every round. These Q-values determine the actions taken by agents.

Parameter	Initial value
Number of buyers i	50
Number of sellers j	50
Number of rounds per game K	5
Number of games G	200
Price distribution first round	$\mathcal{N}(100, 5)$
Budget for buyers	120
Production cost for sellers	80
Maximal increase in price	10
Maximal decrease in price	10
Step size for price changes	2
Punishment for staying in the game	20

Table 1: Initial parameter values (basic setting)

2.3 Matching mechanism

The original set of agents $I \cup J$ is potentially reduced in every round, as buyers and sellers leave the double action once they were matched. The remaining buyers and sellers stay in the game and submit their offers until they are matched as well. We implemented the following matching mechanism:

- The list of remaining buyers and sellers gets permuted randomly after everyone has submitted their offers
- We start with the first remaining buyer and skim through the list of all remaining sellers
- We match buyer i with seller j if $0 < b_i - a_j < 5$. This condition means that buyers need to pay at least the ask price, but are disincentivized to bid absurd price
- The deal priceis sampled from a uniform distribution $d_{i,j} \sim \mathcal{U}(a_j, b_i)$ for every match

3 Learning mechanism

3.1 Principles of reinforcement learning

Reinforcement learning (RL) is a method for solving optimization problems that involves an actor or agent that interacts with its environment and modifies its actions, or control policies, based on stimuli received in response to its actions. It is motivated by natural learning mechanisms as seen in biological organisms who adjust and optimize their behaviour and actions based on the reward and punishment stimuli obtained from the environment around them [LVS12].

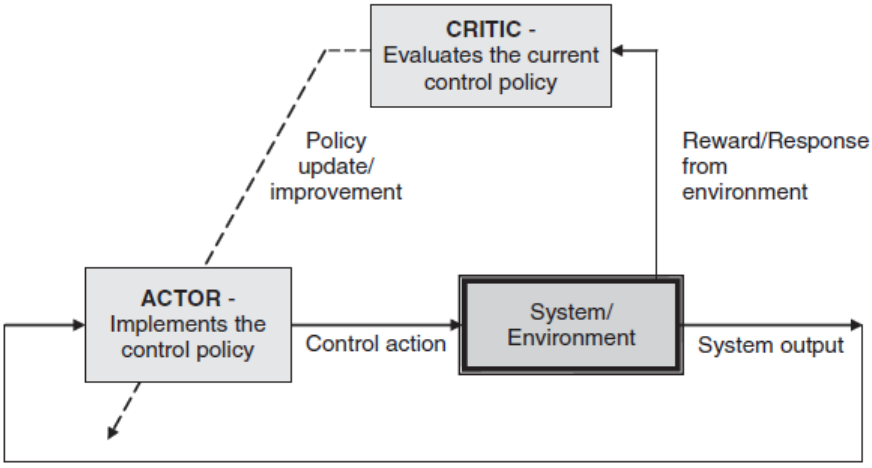


Figure 1: Reinforcement learning structure [LVS12]

Markov Decision Processes (MDP) provide a natural framework for analysing RL problems. An MDP consists of a set of variables (X,U,P,R) where X is a set of states, U is a set of actions, P refers to the transition probability from one state in X to another, if some action in U is taken. Formally,

$$P : X \times U \times X \to [0, 1]$$

$$P^u_{xx} = Pr\{x'|x, u\}$$

R is a reward function

$$R : X \times U \times X \to \mathbf{R}$$

R^u_{xx} gives the expected immediate reward when the MDP goes from state x to x' by taking action u. The basic problem for MDP is to find a mapping $\pi : X \times U \to [0, 1]$ that gives for each state x and action u the conditional probability $(x, u) = Pr(u|x)$ of taking action u given the MDP is in state x. This mapping will be called a (action) policy. Define a performance index as follows:

$$J_{k,T} = \sum_{i=0}^{T} \gamma^i r_{k+1} = \sum_{i=k}^{k+T} \gamma^{i-k} r_i,$$

where $0 \leq \gamma \leq 1$ is a discount factor. T is called the planning horizon of the decision problem. In our case we'll let T be infinity as the solutions for $T = \infty$ are well studied and simple to implement. Hence

$$J_k = \sum_{i=0}^{\infty} \gamma^i r_{k+i} = \sum_{i=k}^{\infty} \gamma^{i-k} r_i.$$

Define now the value function:

$$V^\pi(x) = E_\pi\{J_k|x_k = x\} = E_\pi\{\sum_{i=k}^{\infty} \gamma^{i-k} r_i|x_k = x\}.$$

The value function satisfies the Bellman equation:

$$V^{\pi}(x) = \sum_{u} \pi(x,u) \sum_{x'} P^{u}_{xx'}[R^{u}_{xx'} + \gamma V^{\pi}(x')].$$

If the Markov Chain is ergodic, this is equivalent to:

$$V^{*}(x) = \max_{u} \sum_{x'} P^{u}_{xx'}[R^{u}_{xx'} + \gamma V^{\pi}(x')].$$

and the optimal control policy for state=x is:

$$u^{*} = arg\max_{u} \sum_{x'} P^{u}_{xx'}[R^{u}_{xx'} + \gamma V^{\pi}(x')].$$

3.2 Q-learning

The main problem of finding the optimal policy is made easy by considering a Q-function that takes the state and action as input and outputs a value that is maximized by the optimal policy. Formally:

$$\pi^{*} = argmax_{a} Q(s,a)$$

$$V^{*}(s) = max_{a} Q(s,a)$$

Hence if we can approximate the Q-function then the above two equations directly give the optimal policy and optimal reward. The Q function is approximated by the following trial and error method:

$$Q(s,a) = r + \gamma max_{a'} Q(s',a') \tag{1}$$

For small problems with finite number of states and actions, the Q function can be computed as a look-up table but for problems where the state is continuous like for the prices, the Q-function can be approximated by a neural network as shown below.

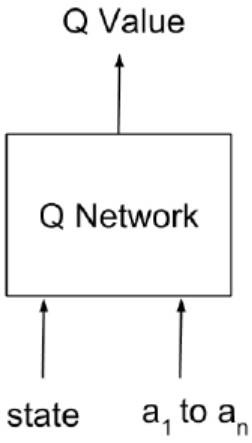


Figure 2: Q network is called n times, once for each action [Ati18]

Hence as the market process starts and continues the Q-network is trained repeatedly for each buyer and seller. Here the state includes the bids of the opposite side and the actions are quantized to be in (-10,-8,...0,...,8,10). Hence we iterate the equation (1) by training the network as and when new rewards are available. It can be proved that the Q learning update equation (1) converges to the optimal Q-value using the Banach fixed point theorem [LVS12] [SB18].

In our case, the Q function is approximated by a deep neural network with 3 hidden layers of hidden-unit sizes (50,30,10) respectively. These are preceded by an input layer that has as many units as there are sellers/buyers + two extra units for price and action. The output is a single unit layer with no activation which provides the final Q value for the given bid setting, price and action.

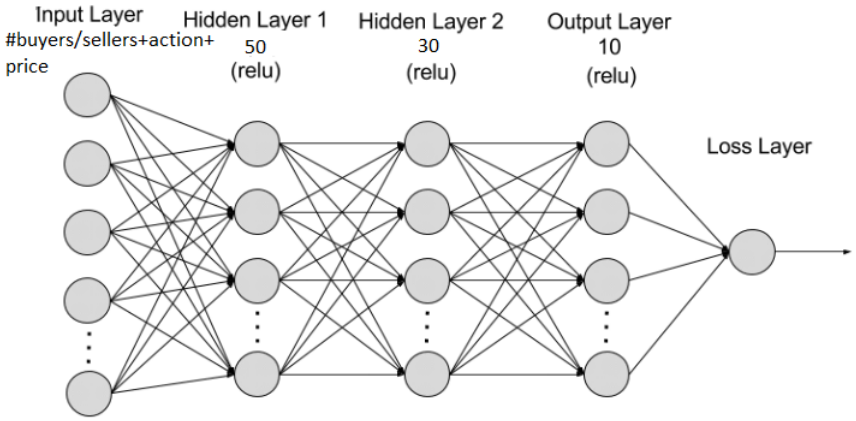


Figure 3: The Q function approximator network

4 Results and Discussion

4.1 Results in the basic setting

Starting with our basic setting as described in table 1, we have found that agents learn and adapt over time and are able to significantly improve their returns:

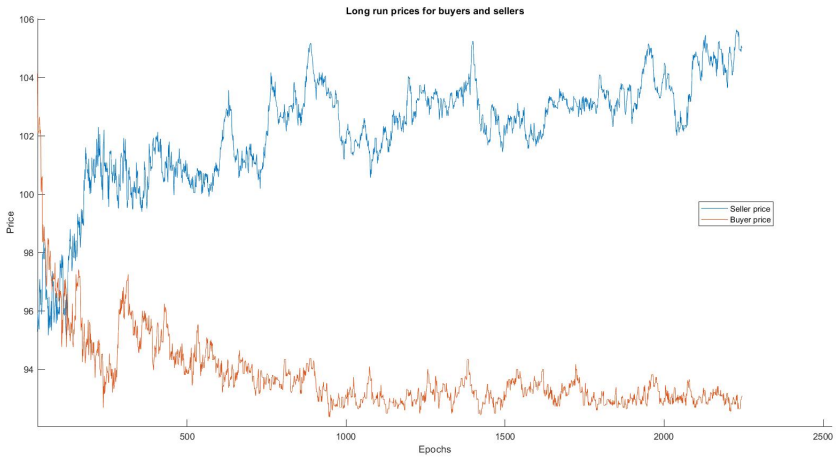
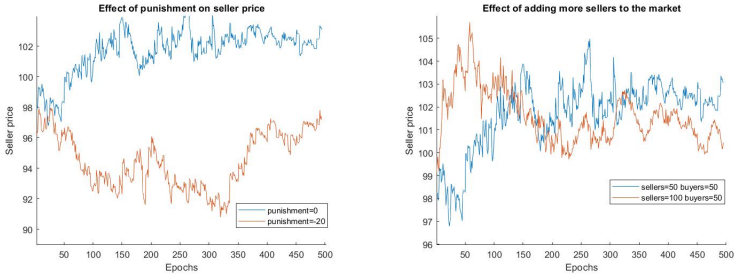


Figure 4: Agents trade at more favorable prices over time based on Q-learning

As expected, observed market prices can depend heavily on external market forces such as opportunity costs or supply and demand:



(a) Increased punishment for sellers leads to lower prices

(b) A surplus of sellers also pushes prices down

Figure 5: Driving forces behind observed market prices

In fact, opportunity costs have a material impact on prices:

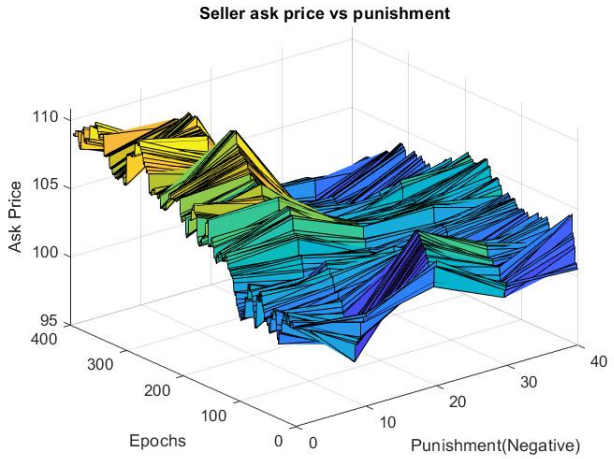


Figure 6: Punishments for sellers substantially push down ask prices

4.2 Testing different market settings and rewards

A lot of different experiments can be done by using the Q-learning algorithm and different versions of the market setting presented previously. In our experiments have chosen to see what happens when there are more or less buyers than sellers. At the same time, we have investigated what happens when we change the reward function.

In our experiment we consider a maximal budget for the buyers. The sellers know the budget of the buyers and so the action spaces of all agents are restricted. Furthermore, we discretized the action space so that each agent can only chose a price in the following list: $0, 10, 20, ..., 100$. An overview of the used parameter values used can be found in the table 2 below:

Parameter	**Initial value**
Number of buyers i	$50\,\&\,10$
Number of sellers j	$50\,\&\,10$
Number of rounds per game K	5
Number of games G	200
Price distribution first round	$\mathcal{N}(50, 5)$
Budget for buyers	100
Production cost for sellers	50
Step size for price	10
Punishment for staying in the game	20

Table 2: Parameter values for the experiment

We then compare three different settings for the reward function:

- **Constant reward**: The reward is the same for both sellers and buyers and it is given by:

$$\text{reward} = \|\text{punishment}\| \tag{2}$$

In this case we expect an equilibrium such that the price of the buyer is above the one of the seller.

- **Positive linear reward**: The reward function increases and decreases linearly with respect to the deal price for both buyers and sellers:

$$r_{buyer} \propto budget - p_{deal} \qquad r_{seller} \propto p_{deal} \tag{3}$$

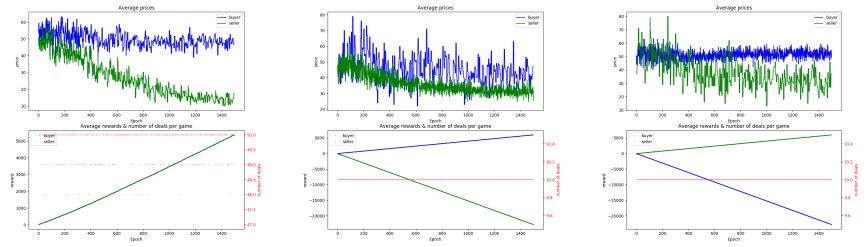
For both buyers and sellers the rewards stay positive if they succeed to find a deal. We expect that buyers try to drive the price down to obtain a better reward where sellers are more inclined to follow the demand of the buyers in order not to get punished.

- **Possibly negative linear reward**: Keep the same reward for buyers but change the reward of the sellers to the following:

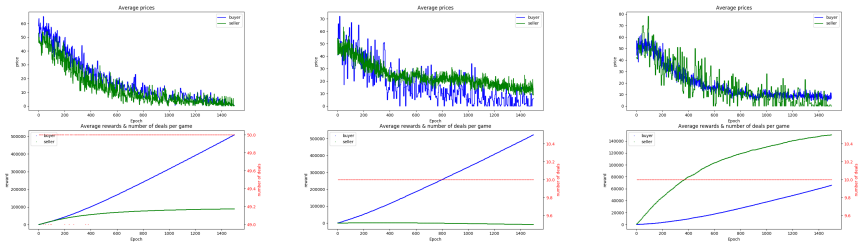
$$r_{seller} \propto \frac{budget}{production\ cost}(p_{deal} - production\ cost) \tag{4}$$

With this reward function, sellers are punished when the price is lower than the production cost and so we are expecting that the mean ask price amongst sellers is going to increase compared to the previous setting. In this case, it is expected that sellers succeed in pushing up prices such that there is an equilibrium just above the deal price.

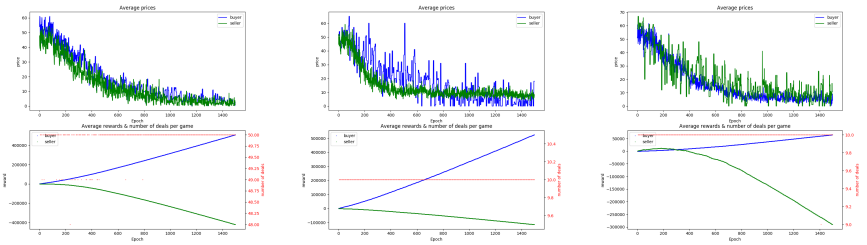
The goal of the experiment is to observe if agents succeed to play the game (i.e. by obtaining a deal), and at which price they converge (if there is convergence).



(a) Constant reward with n_{buyers}=$n_{sellers}$=50 (b) Constant reward with n_{buyers}=10 and $n_{sellers}$=50 (c) Constant reward with n_{buyers}=50 and $n_{sellers}$=10



(d) Positive linear reward with n_{buyers}=$n_{sellers}$=50 (e) Positive reward with n_{buyers}=10 and $n_{sellers}$=50 (f) Positive reward with n_{buyers}=50 and $n_{sellers}$=10



(g) Possibly negative linear reward with n_{buyers}=$n_{sellers}$=50 (h) Possibly negative reward with n_{buyers}=10 and $n_{sellers}$=50 (i) Possibly negative reward with n_{buyers}=50 and $n_{sellers}$=10

Figure 7: Average bid price of the buyers (in blue) and of the sellers (in green) for the different reward functions when the number of agent is the same (left column) or when there are less (center column) or more (right column) buyers than sellers. The number of deals made per game is annotated in red.

4.3 Discussion

In figure 7 we observe that for each reward function when there is less agent of a type than an other, their deviation of the average bid price is bigger than the other type and compared to the case when there is the same number of agent on both sides, the deviation with respect to the equilibrium is larger.

If you consider that sellers win if they succeed to sell their products at a price higher than the production cost of the asset, they actually never win here. Even if we decide to assign a negative reward when they sell cheaper than the production cost, the deal price is never ”fair” and they sell at a loss.

In the scenarios we have examined, considering different kinds of reward

functions, the sellers always lose. This relates to real life phenomena such as the interaction between farmers and distributors: most of the time farmers have to sell their products to distributors at a loss, and governments need to offset this by providing subsidies.

5 Conclusion

In our project we used reinforcement learning such that each agent can learn along time how to play the game that we implemented and improve themselves. We chose some settings to study the collective behavior of each type of agent (sellers and buyers) when we change the reward function. In our game sellers are always disadvantaged compare to buyers. It is easy to modify the code to obtain other kind of results. For example, we have only considered constant and linear reward functions but it is possible to change the reward such that it changes non-linearly with the deal price, the cost and the budget. Also, it is possible to observe what happens if each agents of the same type does not have the same objective.

To go further, it would be interesting to observe what happens when the buyer become a seller after a deal and vice versa and their reward is their wealth at each round.

6 Appendix

6.1 Matching function

```
import numpy as np

class Match:
    def __init__(self,n_sellers,n_buyers):
        self.n_sellers=n_sellers
        self.n_buyers =n_buyers

    def make_matches(self,Sellers,Buyers,
        previous_buyers_bids,previous_sellers_bids):

        seller_perm=np.random.permutation(self.n_sellers)
        buyer_perm =np.random.permutation(self.n_buyers)
        punishment=-10

        sum=0
        count=0

        for seller_id in seller_perm:
            if Sellers[seller_id].get_status()==True:
                for buyer_id in buyer_perm:
                    if Buyers[buyer_id].get_status() ==
                        True:
```

```
                        # and Buyers[buyer_id].get_price
                            () - Sellers[seller_id].
                            get_price() <= 2
                        if Sellers[seller_id].get_price()
                           <=Buyers[buyer_id].get_price()
                            :
                            deal_price=np.random.randint(
                                Sellers[seller_id].
                                get_price(),Buyers[
                                buyer_id].get_price()+1)
                            seller_reward=deal_price-80
                            buyer_reward=1/(deal_price
                                -80+1)

                            Sellers[seller_id].
                                update_qvalue(
                                seller_reward,
                                previous_buyers_bids)
                            Buyers[buyer_id].
                                update_qvalue(buyer_reward
                                ,previous_sellers_bids)

                            Sellers[seller_id].bail_out()
                            Buyers[buyer_id].bail_out()

                            print("Deal made between
                                seller {} and buyer {}".
                                format(seller_id,buyer_id)
                                )
                            print("Deal made at {} for
                                seller_price {} and
                                buyer_price {}".format(
                                deal_price,Sellers[
                                seller_id].get_price(),
                                Buyers[buyer_id].get_price
                                ()))

                            sum=sum+deal_price
                            count=count+1

                            break

        for seller_id in seller_perm:
            if Sellers[seller_id].get_status()==True:
                Sellers[seller_id].update_qvalue(
                    punishment, previous_buyers_bids)

        for buyer_id in buyer_perm:
            if Buyers[buyer_id].get_status()==True:
                Buyers[buyer_id].update_qvalue(punishment
```

```
                    , previous_sellers_bids)

        if count:
            return (sum/count,count)
        else:
            return (0,0)
```

6.2 Value function

```
from keras.layers import Dense,Input,Flatten
import numpy as np
from keras.models import Model
from keras import initializers

class Qvalue:
    def __init__(self,n_inputs):
        self.n_inputs = n_inputs
        self.model=self.create_model()

    def create_model(self):

        input_shape=(self.n_inputs,1)
        n_hidden1=50
        n_hidden2=30
        n_hidden3=15
        n_outputs=1

        inputs= Input(shape=input_shape)
        x=Flatten()(inputs)
        x=Dense(n_hidden1,activation='relu',
            kernel_initializer=initializers.Zeros())(x)
        x=Dense(n_hidden2,activation='relu',
            kernel_initializer=initializers.Zeros())(x)
        x=Dense(n_hidden3,activation='relu',
            kernel_initializer=initializers.Zeros())(x)
        output=Dense(n_outputs)(x)

        model = Model(inputs=inputs, outputs=output)
        model.compile(loss='mean_squared_error',
            optimizer='adam', metrics=['accuracy'])
        return model

    def train(self,action,price,opposite_bids,target):
        X=np.concatenate(([action],[price],opposite_bids)
            ).T
        X = np.reshape(X, (1, self.n_inputs, 1))
        self.model.train_on_batch(X,[target])

    def get_Qvalue(self,action,price,opposite_bids):
        input=np.concatenate(([action],[price],
```

```
            opposite_bids)).T
        input=np.reshape(input,(1,self.n_inputs,1))

        Qval=self.model.predict(input)

        return Qval
```

References

[Ati18] R. Atienza. *Advanced Deep Learning with Keras*. Packt Publishing, 2018.

[LVS12] Frank L. Lewis, Draguna L. Vrabie, and Vassilis L. Syrmos. *Optimal Control*. John Wiley & Sons, Inc., 2012.

[SB18] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2018.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

REINFORCEMENT LEARNING FOR MARKETS

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
HASENFRATZ	SIMON
GARG	AYUSH
TROUDE	Virgile

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date

ZURICH, 06.12.2019

Signature(s)

S. Hasenfratz

Ayush

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.