**Masterarbeit**

# Algorithmic Pricing and Tacit Collusion
## Cooperation in Multi-Agent Deep Reinforcement Learning

**Matthias Hettich**

# Contents

# List of Tables

# List of Figures

# List of Symbols

Miscellaneous

| | |
|---|---|
| $\doteq$ | equality relationship that is true by definition |
| $U\{\mathcal{A}\}$ | uniform distribution over set $\mathcal{A}$ |
| $Pr\{X = x\}$ | probability that a random variable X takes on the value x |
| $\mathbb{E}[X]$ | expectation of a random variable X, $\mathbb{E}[X] = \sum_x p(x)x$ |
| $\mathbb{R}$ | set of real numbers |
| $\in$ | is an element of |
| $\subset$ | subset of |
| $\lvert\cdot\rvert$ | number of elements |
| $\arg\max_a f(a)$ | a value of $a$ at which $f(a)$ takes its maximal value |

Finite Markov Decision Process

| | |
|---|---|
| $t$ | period, a discrete time step |
| $T$ | final period |
| $\gamma$ | discount parameter |
| $s$ | a state |
| $a$ | an action |
| $r$ | a reward |
| $S_t$ | state at time t, stochastically due to $S_{t-1}$ and $A_{t-1}$ |
| $A_t$ | action at time t |
| $R_t$ | reward at time t, stochastically due to $S_{t-1}$ and $A_{t-1}$ |
| $\mathcal{S}$ | set of all states |
| $\mathcal{A}$ | set of all actions |
| $\mathcal{R}$ | set of all rewards |
| $\pi$ | policy or strategy (decision-making rule) |
| $\pi(a \mid s)$ | probability of taking action $a$ in state $s$ under policy $\pi$ |
| $\pi_*$ | optimal policy |
| $F$ | state transition probability |
| $F(s', r \mid s, a)$ | probability of transition from state $s$ and taking action $a$ to state $s'$ with reward $r$ |
| $q$ | action-value |
| $q_\pi(s, a)$ | value of taking action $a$ in state $s$ under policy $\pi$ |
| $q_*(s, a)$ | value of taking action $a$ in state $s$ under the optimal policy |

Q-learning

| | |
|---|---|
| $Q(s, a)$ | array estimate of optimal action-value $q_*(s, a)$ |
| $\varepsilon$ | probability of taking a random action in an $\varepsilon$-greedy policy |
| $\alpha$ | learning rate |
| $Y_t$ | learning target |

Deep Q-Network

| | |
|---|---|
| $Q(s, a, \theta)$ | nonlinear function approximation of optimal action-value $q_*(s, a)$ |
| $\hat{Q}(s, a, \hat{\theta})$ | target network with older weights $\hat{\theta}$ |
| $\theta$ | neuronal network weights |
| $C$ | periods between target network updates |
| $\Delta_\theta f(\theta)$ | gradient with respect to $\theta$ |
| $L(\theta)$ | loss function |
| $\alpha$ | learning rate |
| $B$ | replay buffer |
| $E$ | experience stored in replay buffer |
| $\beta$ | size of the replay buffer |
| $\omega$ | size of the minibatch |
| $r(\pi)$ | average reward under policy $\pi$ |
| $\bar{R}$ | estimation of average reward under optimal policy $r(\pi_*)$ |
| $\lambda$ | step size parameter for the average reward |

Economic Environment

| | |
|---|---|
| $n$ | number of firms in market |
| $r$ | profit of a firm (corresponds to the reward $r$) |
| $p$ | price for a product (corresponds to the action $a$) |
| $p^M$ | monopoly price without competition |
| $p^N$ | static Nash equilibrium price under perfect competition |
| $o_i(p_{j\in n\neq i})$ | optimal response of firm i to prices of the other firms |
| $c$ | marginal costs |
| $g$ | quality of a product |
| $d_i(p_{j\in n})$ | demand for a product i |
| $\mu$ | horizontal product differentiation |
| $\Delta$ | average profit gain (collusion measure) |
| $m$ | number of discrete price steps in action set $\mathcal{A}$ |
| $\eta$ | markup for the price range |

# List of Acronyms

# 1. Introduction

The advancing digitalization enables firms to set prices for their products and services by software programs. Pricing algorithms are easy to implement, especially on e-commerce platforms, and their usage will likely increase.[1] This paper focuses on the fear that these algorithms facilitate collusion in oligopolistic markets.

Enhancements in machine learning and artificial intelligence enable the algorithms to act more autonomously than their rule-based predecessors. They learn pricing strategies without any previous knowledge by active experimentation and little to no guidance from their creators. This autonomous nature fuels concerns that the algorithms learn strategies with a reward-punishment scheme. If competing firms keep their prices at a collusive level, they are rewarded. If a firm undercuts prices, the competing firms' algorithms punish the defection by lowering their own prices. Using these independently learned strategies, the algorithms coordinate prices above the competitive level - collusion occurs (Harrington 2019).

Explicit communication between the algorithms is not necessary for this. The algorithms do not even have to be designed with the intention to facilitate price coordination. Instead, they arrive at this anti-competitive outcome on their own. Collusion shifts from agreements in "smoke-filled hotel rooms" to algorithms that coordinate their prices above competitive levels without communicating. Such an unspoken agreement is called tacit collusion (Ezrachi and Stucke 2017; Ezrachi and Stucke 2018).

Without communication, humans can only cooperate in simple situations. Therefore, regulation focuses on communication leading to collusion rather than collusion as such. For example, US competition law does not prohibit playing a collusive strategy as long as this is not explicitly communicated (Harrington 2019). The current regulatory approach becomes insufficient when self-learning pricing algorithms establish collusion without communication in market situations where humans cannot do so.

Law scholars were the first who raised concerns about collusion by algorithms (Mehra 2016; Ezrachi and Stucke 2017). The topic was picked up by the media (Priluck 2015; Lynch 2017) and is now on the agenda of various competition authorities (OECD 2017; Federal Trade Commission 2018; UK Competition and Markets Authority 2018; Monopolkommission 2018). However, there is no case known where tacit collusion

[1] In their study of around 1600 best-seller products on Amazon, Chen, Mislove, and Wilson (2016) find that more than 500 sellers use pricing algorithms. Amazon, for example, provides an Application Programming Interface (API) that allows retailers to receive information on the prices of competitor's products up to 30 times per hour and to adjust their prices accordingly (Amazon 2020). Third-party providers such as Feedvisor or PROS offer pricing algorithms for smaller companies that cannot afford to develop an algorithm on their own.

by pricing algorithms was detected. This lack of evidence and the experimental literature that identifies communication as key for collusion provoke skepticism about the topic's practical relevance for regulatory authorities (Kühn and Tadelis 2017; Schwalbe 2019).

The risk of tacit collusion by algorithms is difficult to access, both empirically and theoretically. Collusion is hard to empirically detect from market outcomes, except for cases where rich data is available.[2] That is aggravated by the fact that firms keep details about their potential use of pricing software secret. Theoretical results are hard to obtain because interactions among firms in pricing games generate extremely complex stochastic dynamic systems. The only theoretical approach is contained in the study of Salcedo (2015). He finds that the use of pricing algorithms inevitably leads to collusion. However, his results rest on the controversial assumption that algorithms stay fixed for a short period and can be completely decoded by rivals during this time.

Due to these difficulties, the current research focuses on an experimental approach. In these experiments, pricing algorithms repeatedly interact with each other in a computer simulated marketplace. The algorithms should be representative of those used in practice. It is not known which algorithms are actually employed in real markets. However, the current breakthroughs in the field of artificial intelligence, see, e.g., Mnih et al. (2015) and Silver et al. (2016), suggest that algorithms based on Reinforcement Learning (RL) are the most appropriate choice.

In their pioneering work, Waltman and Kaymak (2008) model a sequential Cournot competition. The RL algorithms frequently learn strategies that reduce output and, thus, increase prices. This can be a sign of collusion. However, the reduced output can also occur by failing to learn an optimal strategy.[3] Calvano et al. (2020) are the first to examine the learned strategies in greater detail. They find that pricing algorithms systematically set supra-competitive prices in a sequential Bertrand competition. The algorithms achieve this outcome with reward-punishment schemes, and thus learned truly collusive strategies.[4]

[2] One exception is the analysis of retail gasoline prices by Byrne and De Roos (2019). The analysis uses a unique dataset that contains station-level prices for an urban market over 15 years. With this granular data at hand, they can generate insights into collusion anatomy and how to detect it.

[3] In a different simulation setting, the algorithms are myopic and have no memory of past prices. Thus, they cannot base their decision on past prices. Under these conditions, it is impossible to learn collusive strategies. Nonetheless, Waltman and Kaymak (2008) find that the output reduction is even higher, which must be the result of a failure to learn an optimal strategy.

[4] Other experimental studies use a market model of staggered prices in the style of Maskin and Tirole (1988). In this model, two firms alternate with their pricing decision and keep prices fixed for two consecutive periods. In the most recent paper with this market model, Klein (2019) finds that the pricing algorithms frequently set supra-competitive prices. However, the assumption of a price-commitment seems to be controversial since pricing algorithms can adjust prices very fast. Additionally, these short-term price commitments proved to facilitate collusion (Maskin

All previous studies use Q-learning, one of the simplest RL algorithms available, to model pricing algorithms. Even though Q-learning is suited to present a proof-of-concept of algorithmic collusion, it is only partly representative for algorithms used in practice. The inability to use powerful approximation methods makes Q-learning too slow to be applied in real world scenarios.[5] Additionally, Q-learning's limited computational resources restrict the previous studies to markets with two to four firms. It remains unclear if pricing algorithms can collude with less time for learning and in markets with more competitors.

This paper approaches more realistic pricing algorithms to answer the following questions: Are state-of-the-art RL algorithms able to collude with limited time for learning? Does collusion by self-learning algorithms sustain in wide oligopolies with possibly many firms? In real-world markets, algorithms will be developed independently by each firm. To account for heterogeneity among algorithms, I additionally examine how such differences influence collusion.

I use an experimental approach with a simulated marketplace that builds on the sequential Bertrand competition of Calvano et al. (2020). In particular, I let Deep Q-Network (DQN) algorithms developed by Mnih et al. (2015) compete against each other. This algorithm is the straightforward enhancement of Q-learning to function approximation with the help of Deep Neuronal Networks (DNNs). Since its development, the algorithm became a popular choice and proved its qualities in various application domains. The sequential Bertrand competition could go on infinitely. DQN is designed to tackle tasks with a natural end, and thus needs adjustment to this continuing setting. Therefore, I restated the optimization problem of the DQN algorithm with an average reward formulation, as proposed by Sutton and Barto (2018). To the best of my knowledge, this has never been done before.

The results indicate that state-of-the-art pricing algorithms consistently learn collusive strategies. Algorithms powered by DNNs learn significantly faster than previously used algorithms. They start to increase prices after a time horizon that translates to approximately one month. With an increasing number of market participants, the level of collusion decreases. However, moving away from the purely model-free approach and incorporating knowledge about the algorithm's learning behavior seems to make collusion even in wide oligopolies possible. Analyzing the pricing strategies emphasizes that collusion heavily depends on joint learning. Consequently, heterogeneity among algorithms hinders collusion. However, this holds not true for asymmetries on the firm's level.

and Tirole 1988; Leufkens and Peeters 2011).

[5] In the study of Calvano et al. (2020), the Q-learning algorithms have, on average, one million time steps for learning. If we assume 30 updates per hour (the update frequency of the Amazon API), this translates to more than three years.

The rest of the paper is arranged as follows. The next section describes the setup of the computer simulation. The first part presents the economic environment in which the algorithms interact with each other. The second part contains a description of the pricing algorithms. The simple Q-learning algorithm used in Calvano et al. (2020) illustrates the basic principles of RL. With this knowledge, the functioning of the more sophisticated DQN algorithm becomes easily understandable. Section 3 demonstrates that DQN algorithms learn collusive strategies with limited time. The next two sections focus on i) wider oligopolies and ii) heterogeneity among pricing algorithms. The paper concludes with a discussion of the findings and their implication for policy-making.

# 2. Simulation Setup

## 2.1. Economic Environment

The Bertrand model is the natural choice to represent oligopolistic competition between pricing algorithms. Each firm uses an independent pricing algorithm. There are $n$ firms in the market. Each firm $i$, $i = 1, 2, .., n$, produces a single product with quality $g_i$ and marginal costs $c_i$. I assume no fixed costs. The model describes interactions among firms that set price $p_i$ for their product and consumers that choose quantities at this price, the demand $d_i$.

This one-shot game is repeated in each period. At the beginning of each period, the firms set prices for their product. The demand for each product is computed, and the firms earn their profit. Then, the simulation moves to the next period. Firm $i$'s profit $r_i$ is given by:

$$r_i(p_i, p_{j\in n\neq i}) \doteq (p_i - c_i) \times d_i(p_i, p_{j\in n\neq i}). \tag{1}$$

### 2.1.1. Demand

The well-known multinomial logit model describes the demand side of the simulated oligopolistic market. It allows for horizontal and vertical product differentiation. Besides, it easily extends to product factors beyond quality or more complex heterogeneity among consumers. This flexibility and the straightforward parameterization to model real-world markets make it popular in competition investigations.[6] In each

[6] Other demand models are conceivable. Calvano et al. (2020) show that the choice of the demand model does not influence the collusive behavior of self-learning algorithms.

period the demand for firm $i$'s product is:

$$d_i(p_i, p_{j\in n\neq i}) \doteq \frac{\mathrm{e}^{\frac{g_i-p_i}{\mu}}}{1+\mathrm{e}^{\frac{g_i-p_i}{\mu}}+\sum_{j\in n\neq i}\mathrm{e}^{\frac{g_j-p_j}{\mu}}}. \tag{2}$$

The product quality $g_i$ makes vertical product differentiation possible. $g_i - p_i$ reflects the consumers' utility from buying product $i$. $\mu > 0$ controls for horizontal product differentiation. As $\mu$ increases, the consumer tastes become more heterogeneous. Thus, demand is less responsive to changes in price or quality. For $\mu \to \infty$, the heterogeneity becomes so large that consumers are equally split between products (Anderson and De Palma 1992).

### 2.1.2. Collusion Measure

To compare the degree of collusion, we need a consistent measure across all simulations. The main collusion measure is the average profit gain $\Delta$, defined as:

$$\Delta \doteq \frac{\bar{r_i} - r_i^N}{r_i^M - r_i^N}. \tag{3}$$

$\bar{r_i}$ denotes the average profit of firm $i$ over a given number of periods. $r_i^N$ is the profit of firm $i$ in the static Bertrand-Nash equilibrium and $r_i^M$ is the monopoly profit. For $\Delta = 0$ the average profit correspond to the competitive outcome and for $\Delta = 1$ to the outcome under full collusion (Calvano et al. 2020).

$r_i^N$ is the profit that a firm $i$ receives when all firms play the Nash price of the one-shot game $p^N$. In a Nash equilibrium, no firm can increase her profit by unilaterally changing her price. Firm $i$ chooses $p_i$ to maximize her profit $r_i$, taking the other prices $p_{j\in n\neq i}$ as given. Thus, prices $p_{i\in n}^N$ constitute a Nash equilibrium if and only if:

$$p_i^N \doteq \arg\max_{p_i} r(p_i, p_{j\in n\neq i}^N) \text{ for all } i \in n. \tag{4}$$

To find the static Nash equilibrium prices I use the fixed-point theorem. We have a function $f(x)$ and look for a point $x^*$ such that $x^* = f(x^*)$. Let $o_i(p_{j\in n\neq i}) \doteq \arg\max_{p_i} r(p_i, p_{j\in n\neq i})$ denote firm $i$'s optimal response function. The optimal response function returns the price that maximizes the own profit subject to the competitors' prices. Defining the vector function $f$ as:

$$f(p_{i\in n}) \doteq \begin{pmatrix} o_1(p_{j\in n\neq 1}) \\ o_2(p_{j\in n\neq 2}) \\ \vdots \\ o_n(p_{j\in n\neq n}) \end{pmatrix}, \tag{5}$$

we are looking for the point $p^N_{i\in n}$ such that $p^N_{i\in n} = f(p^N_{i\in n})$:

$$\begin{pmatrix} p^N_1 \\ p^N_2 \\ \vdots \\ p^N_n \end{pmatrix} = \begin{pmatrix} o_1(p^N_{j\in n\neq 1}) \\ o_2(p^N_{j\in n\neq 2}) \\ \vdots \\ o_n(p^N_{j\in n\neq n}) \end{pmatrix}. \tag{6}$$

After rearranging, equation (6) can be solved with a root finding algorithm.[7]

The profit under full collusion $r^M_i$ is the profit that a firm $i$ receives when all firms in the market play the monopoly prices $p^M_{i\in n}$ of the one-shot game. These are the prices that maximize the joint profit of all firms in the market:

$$p^M_{i\in n} \doteq \arg\max_{p_{i\in n}} \sum_{i=1}^{n} r_i(p_{i\in n}). \tag{7}$$

The profit gain is a suitable measure for tacit collusion. With tacit collusion, firms can obtain supra-competitive profits per definition, where supra-competitive refers to the equilibrium situation (Ivaldi et al. 2003). The Nash profit $r^N$ of the one-shot game is a reasonable estimation of the equilibrium situation. With the monopoly profit $r^M$, the collusion measure is normalized between 0 and 1. This allows comparing collusion across different simulation settings.

### 2.1.3. Price Range

The pricing algorithms require a discrete price range to choose from. A reasonable price range includes the static Nash equilibrium prices $p^N_{i\in n}$ and the monopoly prices $p^M_{i\in n}$ of the one shot-game. Then, the price range is given by $m$ equally spaced points in the interval:

$$\mathcal{A} \doteq \left[\min(p^N_{i\in n}) - \eta, \max(p^M_{i\in n}) + \eta\right], \tag{8}$$

where I set $\eta \doteq 0.1[\max(p^M_{i\in n}) - \min(p^N_{i\in n})]$. This markup increases the price range by 10% in both directions (Calvano et al. 2020).

## 2.2. Pricing Algorithms

Repeated games, like the sequential Bertrand competitions, can be modeled as a Markov Decision Process (MDP). In a MDP, the decision-maker, called agent, interacts continually with the environment, that is, everything outside of the agent.

[7] Instead of computing each firm's optimal response function $o_i$, I directly look for the profit maximizing price with the help of a maximization algorithm. For both the root finding and the maximization algorithm I use the Python package SciPy from Virtanen et al. (2020).

In each period $t$, $t = 1, 2, ..., T$, the agent observes the environment's state, $S_t \in \mathcal{S}$. Based on the observed state, the agent selects an action, $A_t \in \mathcal{A}$. As a consequence of the action, the agent receives a numerical reward, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and the system moves on to the next state, $S_{t+1}$, according to the time-invariant state-transition probabilities $F$. They describe a probability distribution for each combination of $a \in \mathcal{A}$ and $s \in \mathcal{S}$:[8]

$$F(s', r|s, a) \doteq Pr\{S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a\}. \tag{9}$$

The goal of the agent is to select action $A_t$ in order to maximize the present value of future rewards:

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k \, R_{t+k+1}, \tag{10}$$

where $\gamma$ is a discount factor, $0 \leq \gamma < 1$. The agent's future rewards depend on her way of acting, called the policy $\pi(a|s) \doteq Pr\{A_t = a \mid S_t = s\}$. The policy describes with which probability an agent will chose $A_t = a$ if she is in state $S_t = s$ and, thus, $\pi$ defines a probability distribution over $a \in \mathcal{A}$ for each $s \in \mathcal{S}$ (Sutton and Barto 2018).[9]

The optimal action-value $q_*(s, a)$ denotes the maximum expected rewards achievable by following any policy $\pi$ after choosing action $a$ in state $s$:

$$q_*(s, a) \doteq \max_{\pi} \mathbb{E}_\pi[G_t|S_t = s, A_t = a]. \tag{11}$$

We can rewrite the optimal action-value without reference to a policy by using the Bellmann equation. The value of a state-action pair under an optimal policy equals the reward expected from transition to the next period plus the discounted action-value of the next state. The next state's action-value assumes that the agent will behave optimal, and again reflects all future rewards under an optimal policy. With this recursive property the optimal action-values become:

$$q_*(s, a) = \mathbb{E}[R_{t+1} + \gamma \max_{a} q_*(S_{t+1}, a)|S_t = s, A_t = a]. \tag{12}$$

The Bellmann equation (12) reflects all long-term returns as a value that is locally available. If the agent knows the action-values, she can behave optimally. The agent observes the current state $s$ and chooses action $a$ for which $q_*(s, a)$ is maximal

[8] Capital letters denote random variables, whereas lower case letters are used for values of random variables. $s'$ is shorthand for next period's state.

[9] The name policy originates from the terminology of the RL literature. It corresponds to the term strategy from the game theory domain. Both terms are used synonymously.

(Sutton and Barto 2018). Thus, she follows the optimal policy $\pi_*$:

$$\pi_*(a \mid s) = \arg\max_a q_*(s, a). \tag{13}$$

The MDP framework can easily be adapted to the sequential Bertrand competition. In the simulation, each firm is represented by one independent agent. The state of the environment at period $t$ are the prices that the agents, or firms, played in the previous time step $t-1$. The set of possible actions $\mathcal{A}$ are the prices that an agent can set for her product, i.e., the price range determined by equation (8). After each agent played her action, the environment moves to the next state $S_{t+1}$. The environment determines the demand for each product by equation (2). Then, each agent receives her reward signal $R_{t+1}$, the profit calculated by equation (1).

The Bellmann optimality equation (12) describes a system of $|S| \times |A|$ nonlinear equations, one for each state-action pair, in the same number of unknowns. If the state-transition probabilities (9) are known, standard methods can solve the system of equations. However, in many real-world applications, the state-transition probabilities are unknown. In this case the action-values $q_*(s, a)$ must be estimated. Agents based on RL estimate the action-values by trial-and-error interactions with the environment (Sutton and Barto 2018).

In the sequential Bertrand competition, the state-transition probabilities are, in fact, unknown. The transition from one state to the other state depends not only on the own action but also on all other agents' actions. Thus, the optimal policy of one agent depends on the policy of all other agents. The environment becomes non-stationary, and the Markov property does not hold (Laurent, Matignon, and Fort-Piat 2011). Even though RL algorithms are designed initially to tackle MDPs with only one agent and time-invariant state-transition probabilities, they can be applied to environments with multiple agents.[10]

### 2.2.1. Q-Learning

One of the simplest and best understood RL algorithm is Q-learning developed by Watkins (1989). It learns a Q-value $Q(s, a)$ that directly approximates the optimal action-values $q_*(s, a)$ by the following updating rule:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]. \tag{14}$$

[10] General convergence proofs for single-agent RL do not apply here. Nonetheless, empirical results show that RL algorithms do perform well in strategic multi-agent environments (Buşoniu, Babuška, and De Schutter 2008; Bloembergen et al. 2015; Leibo et al. 2017).

After initializing $Q(s, a)$ with arbitrary values for each state-action pair, the value of a state-action pair is updated every time it is visited according to equation (14). The quantity in the square brackets constitutes an error term. It measures the difference between the current estimate of the action-value and an updated estimate, the target value $Y_t = R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$. This target value comprises the observed reward and the discounted value of being in the next state $Q(S_{t+1}, a)$. The latter assumes that the agent will behave optimally in the next state and includes all expected future rewards.[11]

The learning rate $\alpha$, $0 \leq \alpha \leq 1$, specifies how much weight is given to the new knowledge compared to the current estimation. The tabular storage of the Q-values is often referred to as the Q-matrix. This is a matrix with $|\mathcal{S}| \times |\mathcal{A}|$ dimensions, one cell for each possible state-action pair (Sutton and Barto 2018).

The Q-value is only updated for state-action pairs who are visited. Therefore, all state-action pairs have to be visited sufficiently often in order to approximate the optimal action-value $q_*(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. Instructing the agent to experiment, i.e., randomly choose an action that may appear sub-optimal based on her current knowledge, assures sufficient exploration. This exploration of the whole state-action space comes at the cost of not exploiting the current knowledge.

A straightforward procedure to ensure exploration is a $\varepsilon$-greedy policy. With probability $\varepsilon$, $0 \leq \varepsilon \leq 1$, the agent will randomly select any of the possible actions. In all other cases the agent behaves optimal or greedy according to her current knowledge:

$$a = \begin{cases} \arg\max_a Q(s, a) & \text{with probability } 1 - \varepsilon \\ U\{\mathcal{A}\} & \text{with probability } \varepsilon \end{cases}, \tag{15}$$

where $U\{\mathcal{A}\}$ denotes a sample from the discrete uniform distribution over the set of actions $\mathcal{A}$. Algorithm 1 shows the pseudo-code for Q-learning. It illustrates that the algorithm estimates $q_*(s, a)$ and, thus, the optimal policy $\pi_*(a \mid s)$ by following the non-optimal $\varepsilon$-greedy policy. Therefore, Q-learning falls into the category of off-policy methods (Sutton and Barto 2018).

### 2.2.2. Deep Q-Network

Q-learning estimates the optimal action-values for each state-action pair separately. Without generalization, the learning capabilities of this approach are not efficient enough for real-world applications. Thus, it is common to use a function approximator with weights $\theta$ to estimate the action-values $Q(s, a, \theta) \approx q_*(s, a)$. Since the pioneering work of Mnih et al. (2015), nonlinear function approximation based on DNNs, called

[11] It is interesting to note that Q-learning estimates $Q(S_t, A_t)$ based on another estimate $Q(S_{t+1}, a)$. This technique is called bootstrapping.

**Algorithm 1** Q-Learning

1: Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}$, arbitrarily
2: Initialize $S_1$ arbitrarily
3: **for** $t = 1, T$ **do**
4: With probability $\varepsilon_t$ play a random action $A_t$
Otherwise play $A_t = \arg\max_a Q(S_t, a)$
5: Observe $R_{t+1}$, $S_{t+1}$
6: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]$

Q-network, are widely used.

Using function approximation has two advantages. First, the experience from one state-action pair improves the estimate for nearby state-action pairs by generalization. This will lead to faster learning. Second, it frees us from the curse of dimensionality. The tabular representation of action-values used by Q-learning exponentially increases in the number agents. Thus, memory consumption quickly reaches its limit.

The Q-network can be structured in various ways. Since the action-values assign a value for each state-action pair, some approaches use the state and action as input to the DNN. The main drawback of this approach is that a separate forward pass is required to compute the Q-value for each action. Instead, I only use the state as input to the network and one output unit for each possible action. The output of the DNN corresponds to the Q-value of each action at the input state. The main advantage of this approach is that only one forward pass is required to compute the Q-values of all actions in a given state (Mnih et al. 2015).[12]

Figure 1 shows the structure of the Q-network. Input are the agents' previously played prices, the state of the environment. Therefore, the number of input nodes corresponds to the number of agents in the market. The Q-network has two fully-connected hidden layers with 32 nodes each, followed by rectified linear activation functions. The output layer is fully-connected as well, and has one unit per action. Thus, the number of output units equals the number of possible prices $m$. Here, the network uses a linear activation function.

The size of the network is a somewhat arbitrary decision. Theoretically, one large hidden layer can approximate any function. In practice, deep networks with more than one hidden layer proved to perform better than shallow networks. With an increasing number of layers or nodes, the network can learn more and more abstract representations. However, larger networks need more time and training samples to find good solutions (Goodfellow, Bengio, and Courville 2016).

[12] For example, if an agent wants to play the greedy action, she chooses the action for which the Q-Network predicts the highest value, $A_t = \arg\max_a Q(S_t, a, \theta)$. Only one forward pass is needed.

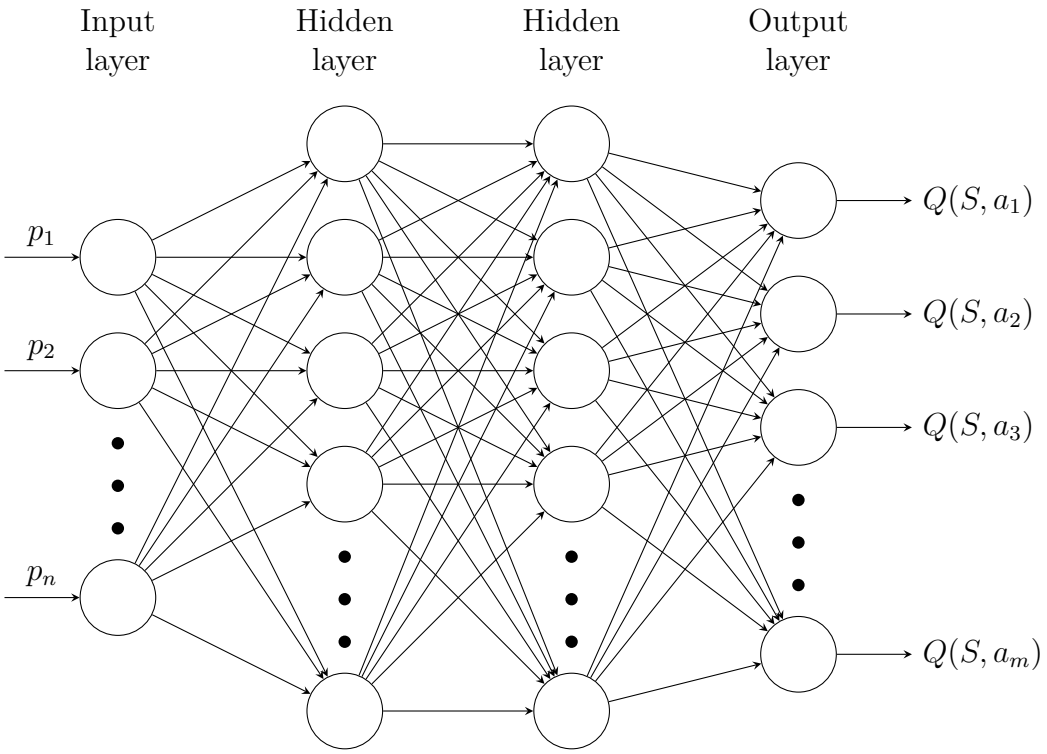



Figure 1: Schematic illustration of the Q-network.

**Learning** With function approximation, we seek to improve the weights $\theta$ instead of direct estimates of the action-values. There are more state-action pairs than weights by definition. In contrast to Q-learning, we cannot get each action-value exactly correct. Making one state-action pair's estimate more accurate means making others' less accurate. Thus, we need a measure-of-fit to evaluate our approximation. A meaningful measure of the difference between the approximated value and the true target value is the Mean Squared Error (MSE).

We can apply this error measure to the training of the Q-network. At each iteration, we adjust the weights $\theta$ to reduce the MSE in the Bellmann equation (12). The true target value, the optimal action-value $R_{t+1} + \gamma \max_a q_*(S_{t+1}, a)$ is not known. Thus, we approximate it by $Y_t = R_{t+1} + \gamma \max_a \hat{Q}(S_{t+1}, a, \hat{\theta})$ using the Q-network with weights $\hat{\theta}$ from a previous iteration.[13] The loss function we want to minimize is then:

$$L(\theta) \doteq \left[ R_{t+1} + \gamma \max_a \hat{Q}(S_{t+1}, a, \hat{\theta}) - Q(S_t, A_t, \theta) \right]^2 . \tag{16}$$

Reducing the error completely at each step is impracticable. As described above, we need to balance the fit between action-values. At each learning step, we train the Q-network with only a small sample. Instead of reducing the error for this sample completely, the weights can be moved only a small step in the direction where the

[13] In contrast to supervised learning, the loss function's target values are not fixed before training but depend on previous weights. At each learning step, the weights $\hat{\theta}$ are held constant. Thus, the optimization problem is well-defined (Mnih et al. 2015).

error falls most rapidly (Sutton and Barto 2018).

Stochastic Gradient Descent (SGD) is the most popular method to implement this fitting procedure. At each learning step, the weights $\theta$ are adjusted by a small amount in the direction that reduces the error $L(\theta)$ specified in equation (16) the most:

$$\begin{aligned} \theta &\leftarrow \theta - \frac{1}{2}\alpha\Delta_\theta \left[R_{t+1} + \gamma \max_a \hat{Q}(S_{t+1}, a, \hat{\theta}) - Q(S_t, A_t, \theta)\right]^2 \\ \theta &\leftarrow \theta + \alpha \left[R_{t+1} + \gamma \max_a \hat{Q}(S_{t+1}, a, \hat{\theta}) - Q(S_t, A_t, \theta)\right] \Delta_\theta Q(S_t, A_t, \theta), \end{aligned} \tag{17}$$

where $\alpha$ is a positive step-size parameter comparable with the one for Q-learning. $\Delta_\theta f(\theta)$ denotes the vector of partial derivatives (the gradient) with respect to the weight vector:

$$\Delta_\theta f(\theta) = \left(\frac{\partial f(\theta)}{\partial \theta_1}, \frac{\partial f(\theta)}{\partial \theta_2}, ..., \frac{\partial f(\theta)}{\partial \theta_d}\right)^\top. \tag{18}$$

The gradient indicates the direction in which the loss function $L(\theta)$ has the steepest ascent (Sutton and Barto 2018).[14]

SGD yields too little information about the loss function to train the Q-network efficiently. It only takes the first-order derivatives into account. These contain information how fast the loss declines in which direction. However, it contains no information about the curvature: whether the loss declines faster or slower as we move in a certain direction. Second-order derivatives contain this information. However, they are computationally expensive. Common optimization methods consider the gradients of past steps to guide the search. This is the idea behind Adam, which I use in the simulations (Goodfellow, Bengio, and Courville 2016).[15]

**Modifications of the algorithm** Algorithm 2 shows the pseudo-code for DQN following Mnih et al. (2015). Similar to Q-learning, the algorithm selects actions based on an $\varepsilon$-greedy policy (15). Off-policy learning, combined with function approximation, leads to a danger of instability and divergence. Additionally, the discounted problem formulation (10) is incompatible with function approximation (Sutton and Barto 2018). Compared to Q-learning, the algorithm is modified in three ways to address these challenges: i) the average reward formulation replaces the discounted formulation, ii) learning is performed with experience replay, and iii) a separate network with older weights calculates the target values.

The first modification addresses the problem formulation. Equation (11) implies

[14] The direction that reduces the error the most is the direction exactly opposite to the gradient. That is why we are subtracting the $\Delta_\theta L(\theta)$ vector from the weights vector.

[15] There are many different implementations. For each of the most common optimization methods, I tested how well the DNN was able to learn several strategies of increasing complexity. In this testing procedure, Adam performed best.

**Algorithm 2** Deep Q-Network

Initialize local network $Q(s, a, \theta)$ with arbitrary weights $\theta$
Initialize target network $\hat{Q}(s, a, \hat{\theta})$ with weights $\hat{\theta} = \theta$
Initialize average reward estimate $\bar{R}$ arbitrarily
Initialize $S_1$ arbitrarily
**for** $t = 1, T$ **do**
    With probability $\varepsilon_t$ play a random action $A_t$
    Otherwise play $A_t = \arg\max_a Q(S_t, a; \theta)$
    Observe $R_{t+1}$, $S_{t+1}$
    Store transition $S_t$, $A_t$, $R_{t+1}$, $S_{t+1}$ in $B$
    Sample random minibatch of transitions $(S_j, A_j, R_{j+1}, S_{j+1})$ from $B$
    Set $Y_j = R_{j+1} - \bar{R} + \max_a \hat{Q}(S_{j+1}, a, \hat{\theta})$
    Perform a gradient descent step on $[Y_j - Q(S_j, A_j, \theta)]^2$ with respect to $\theta$
    $\bar{R} \leftarrow \bar{R} + \lambda \left[ R_{t+1} - \bar{R} + \max_a \hat{Q}(S_{t+1}, a, \hat{\theta}) - \hat{Q}(S_t, A_t, \hat{\theta}) \right]$
    Every $C$ steps reset $\hat{Q} = Q$

that a policy is optimal, if, for every state-action pair, following this policy leads to a higher discounted sum of future rewards than any other policy. The optimal policy satisfies the following inequality:

$$q_{\pi_*}(a \mid s) \geq q_\pi(a \mid s) \text{ for all } a, s \text{ and } \pi. \tag{19}$$

These inequalities define a partial order on the set of possible policies.

Tabular methods store a separate estimate for each state-action pair. Thus, they can represent any possible policy. For example, Q-learning updates one state-action pair at a time, thus, slowly improving its policy in the direction of the optimal policy. Q-learning can tackle this problem formulation and find the policy that is at least as good as all other policies for each state-action pair.

With function approximation, not each possible policy can be represented since there are fewer weights than state-action pairs by definition. Thus, the optimal policy defined in (19) usually cannot be represented. Instead, the agent tries to learn the best representable policy. However, there is typically no representable policy that is universally better than all other representable policies. For some state-action pairs, one representable policy will be better, and for other state-action pairs, another representable policy. The partial order described in (19) is not transferable to representable policies and the problem formulation is not well-defined (Singh, Jaakkola, and Jordan 1994; Naik et al. 2019).

Instead of the partial order of policies, we should use an explicit optimization objective to compare any two policies. The agent chooses actions in order to maximize the present and future rewards. In episodic tasks, the number of periods is relatively small, and each episode has a natural end. Therefore, the sum of all discounted

future rewards (10) is a meaningful explicit objective.

The sequential Bertrand competition has no natural end and will potentially go on infinitely. We need to find an objective that is similar to the sum of rewards in episodic tasks. The basic principle of RL is that the agent tries to maximize the rewards not just in the present but over the whole lifetime. The agent should choose actions that cause her to visit states with high rewards more frequently (Naik et al. 2019). Since the time horizon is infinitely in continuing tasks, the average reward $r(\pi)$ is a good candidate for the optimization objective:

$$r(\pi) \doteq \lim_{h \leftarrow \infty} \frac{1}{h} \sum_{t=1}^{h} \mathbb{E}[R_t | A_{0:t-1} \sim \pi], \tag{20}$$

where the expectation is conditioned on the prior actions $A_0, A_1, ..., A_{t-1}$ taken according to the policy $\pi$. This measure can order the policies by a single number, their average reward $r(\pi)$. Each policy that attains the maximum value of $r(\pi)$ is an optimal policy (Sutton and Barto 2018).

In the average-reward setting the goal of the agent (10) must be restated as differences between rewards and the average reward:

$$G_t = R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \cdots . \tag{21}$$

The optimal action-values (12) can easily be adjusted to the average formulation by replacing the discount factor $\gamma$ with the difference between reward and the true average reward:

$$q_*(s, a) = \mathbb{E}[R_{t+1} - \max_{\pi} r(\pi) + \max_{a} q_*(S_{t+1}, a) | S_t = s, A_t = a]. \tag{22}$$

The loss function of our algorithms must reflect these changes as well:

$$L(\theta) = \left[ R_{t+1} - \bar{R} + \max_{a} \hat{Q}(S_{t+1}, a, \hat{\theta}) - Q(S_t, A_t, \theta) \right]^2, \tag{23}$$

where $\bar{R}$ is an estimate of the true average reward $r(\pi)$ at time $t$. This estimate is updated at the end of every period. It is pushed a small step, defined by $\lambda$, towards the true value as the learned policy converges to the optimal policy (Sutton and Barto 2018).

Figure 2 shows the rewards/profits of a DQN agent with discounted reward formulation in comparison to an agent with average reward formulation. They play against an agent with a hard-coded Tit-for-Tat policy.[16] Both agents are able

[16] The agent with the Tit-for-Tat strategy will play the lowest possible price if the opponent undercut her in the previous round. Otherwise, she will replicate the opponent's previous action. This policy encourages cooperation.

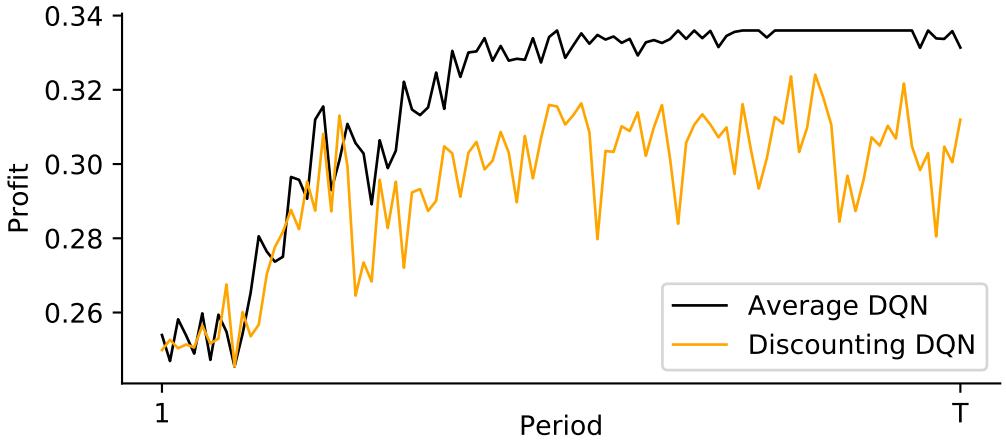



Figure 2: Learning behavior of DQN agents with discounted and average reward formulation against a Tit-for-Tat agent. The lines represent average values over 10 simulation runs. Each simulation lasts 1000 periods.

to increase their rewards/profits over time. However, the oscillating behavior of the agent with discounted rewards is a sign that the optimization problem is not well-defined. In contrast, the agent with average reward formulation learns smoothly and behaves almost always optimally at the end of the simulation runs.

The second modification is the use of experience replay. With experience replay we store the agents experience $E_t = S_t, A_t, R_{t+1}, S_{t+1}$ in the replay buffer $B = E_1, ..., E_\beta$. The replay buffer is limited to store $\beta$ experiences. If the replay buffer is full, the oldest memory is deleted to free space for the new experience. At each period $t$, the agent does not learn with the newest experience $E_t$. Instead, $\omega$ experiences are randomly sampled from the replay buffer. This minibatch is used to perform a gradient descent step, as described previously.

Training on a minibatch instead of a single observation has several advantages. First, each experience is used in possibly many learning steps. Second, consecutive samples often have a strong correlation. With random sampling, this correlation reduces and the variance of the updates declines. Third, unwanted feedback loops are avoided. For example, if the current best action is to set a high price, the next states will be dominated by samples with high prices. If the best action is to play low prices, the training samples will be dominated by samples with low prices. Thus, it is likely that the algorithm could get stuck in a poor local minimum (Mnih et al. 2015). Lastly, software that implements DNNs is highly optimized to larger training batches. Thus, training on these minibatches instead of a single observation significantly decreases computation time for learning.

The third modification, a separate network for the target value calculation, further improves the algorithm's stability. The approximated target $Y_t$ is calculated with the target network $\hat{Q}(s, a, \hat{\theta})$. The target network $\hat{Q}$ has the same structure as the

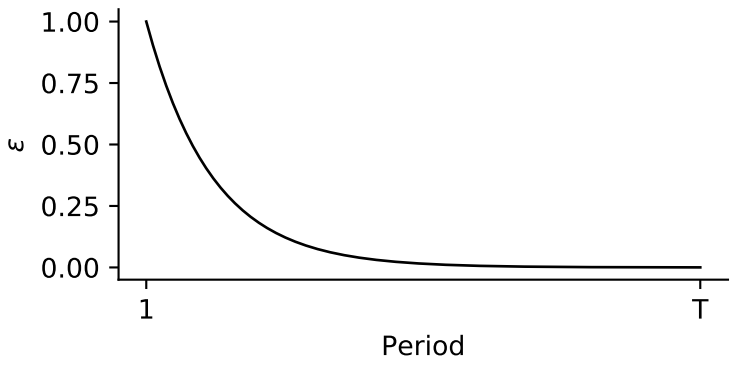



Figure 3: The $\varepsilon$-greedy policy with decreasing probability of exploration.

local network $Q$. The SGD update, the learning, is performed on the local network $Q(s, a, \theta)$ and the agent also uses this network to determine her actions. Each $C$ periods, the weights of the target network $\hat{\theta}$ are replaced by the weights of local network $\theta$.

Generating the targets with an older set of weights adds a delay between learning and the effect in the target $Y_t$. Without this delay, learning would immediately influence the learning target. Considerable risk of oscillation or divergence in the learning process would occur (Mnih et al. 2015).

### 2.2.3. Policy

Self-learning agents must explore the whole state-action space to learn optimal policies. As stated in section 2.2.1, this can be assured by following an $\varepsilon$-greedy behavior policy. Experimentation is especially helpful at the beginning when the agents have little knowledge about the environment or the behavior of the competitors. However, experimentation is costly and with increasing knowledge, the greedy action will more likely result in higher rewards. Therefore, the agents follow a policy with exponentially decreasing probability of experimentation $\varepsilon_t$:

$$\varepsilon_t = \left(0.015^{\frac{2}{T}}\right)^t. \tag{24}$$

The probability of experimentation starts around 1 in the first period to 0.015 halfway the run until 0.000225 in the last period $t = T$. Figure 3 shows $\varepsilon_t$ over the curse of the simulation. Q-learning and DQN learn with different pace. Thus, the simulations have varying length $T$ depending on the used algorithm. The $\varepsilon$-greedy policy dynamically adapts to different simulation lengths, and makes exploration comparable across simulations.

## 2.3. Technical Implementation

The experiment is implemented in Python. I choose an object-oriented approach, and the entities of the simulation are designed as separate objects, each with a standardized interface. Therefore, different agents, demand models, or policies can be combined with one another. In addition, new agents or more sophisticated demand models can be integrated straightforwardly. The environment is scalable to model markets with a varying number of firms from two upwards. The full source-code can be found on GitHub (`https://github.com/matthias-hettich/price_simulator`).

The simulations were deployed on the high performance computing cluster PALMA II of the WWU Münster. For simulations with tabular algorithms, e.g., Q-learning, this enabled the usage of up to 36 Central Processing Units (CPUs). The DNNs are implemented in TensorFlow 2.2.0 by Abadi et al. (2016). This software uses data flow graphs to build large-scale machine learning models and is optimized for calculations on Graphics Processing Units (GPUs). The PALMA II cluster allowed training of the DNNs on up to 8 GPUs in parallel.

# 3. Collusion with Limited Time

This section focuses mainly on duopolies and examines whether pricing algorithms can learn collusion with limited time. Each firm or agent $i$ has marginal costs $c_i = 1.0$ and product quality $g_i = 2.0$. The price sensitivity is set to $\mu = 0.25$, and the agents can choose from $m = 15$ prices. The agent's parameters differ between algorithms. For Q-learning the discount parameter is $\gamma = 0.95$ and the learning rate $\alpha = 0.125$. Agents based on DQN have several parameters. The most important parameters will be discussed in section 3.2. For an overview of all parameters, see appendix A.

## 3.1. The Need for Approximate Learning Methods

Q-learning agents frequently learn to collude, as shown by Calvano et al. (2020). Their results' replication serves as a benchmark to compare the collusive behavior of more sophisticated algorithms. One simulation run lasts $T = 1,000,000$ periods and the market size varies from two to four Q-learning agents. Figure 4 shows the average profit gain $\Delta$ in the last $25,000$ periods.

In a duopoly, the agents consistently reach average profit gains of $\Delta = 67\%$. If three agents are active in the market, the tendency to collude significantly decreases to $\Delta = 30\%$ and becomes even smaller in markets with four agents, $\Delta = 27\%$.

Calvano et al. (2020) show that Q-learning algorithms could reach average profit

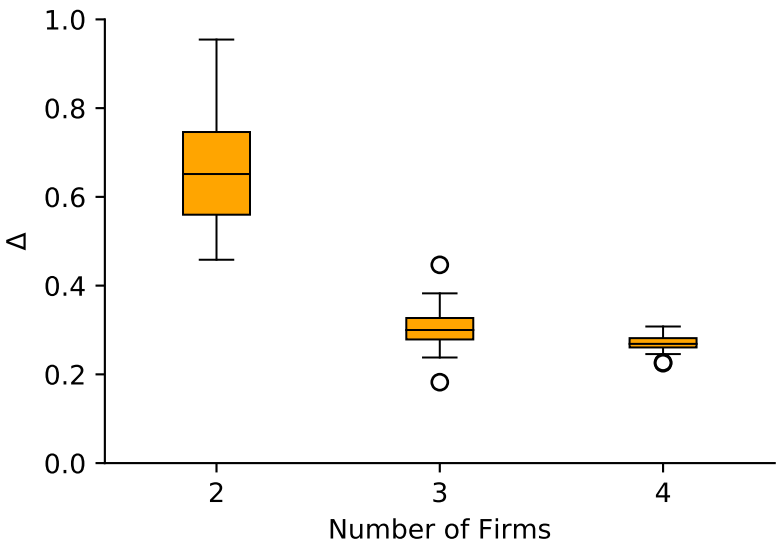


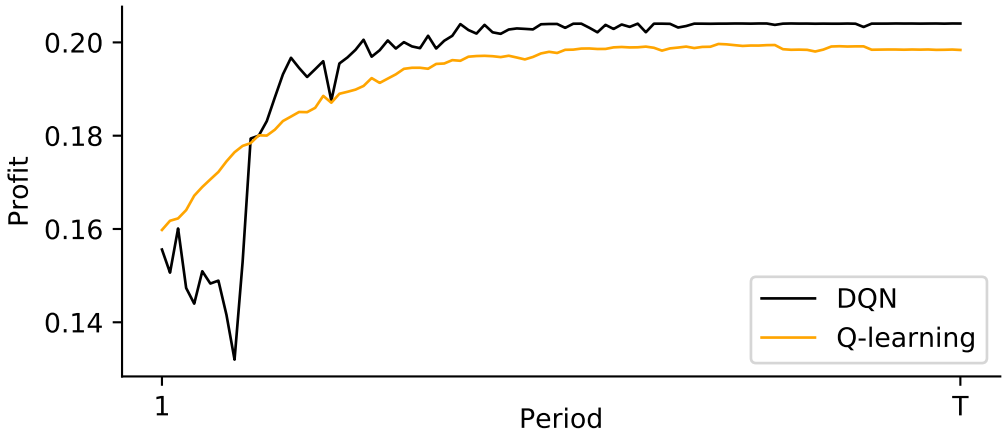

Figure 4: Average profit gains by Q-learning agents for different market sizes. The box-plot shows the results of 100 simulation runs. The box extends from the lower to the upper quartile of the average profit gains, with a line at the median. The whiskers show the range of the data. Values beyond the whiskers are considered outliers and are plotted as individual points.

gains of $\Delta = 56\%$ in a market with four agents. However, the algorithms had up to one billion periods to learn their strategies. On Amazon, the prices can be updated 30 times per hour. If we assume this update frequency, 1 million periods already translate to a time horizon of roughly three and a half years. Especially on e-commerce platforms, it is quite unrealistic that markets stay unaltered that long. Simulations with this time horizon will not bring us closer to assess the probability of collusion in real markets. Instead, we should focus on more efficient self-learning algorithms, i.e., DQN.

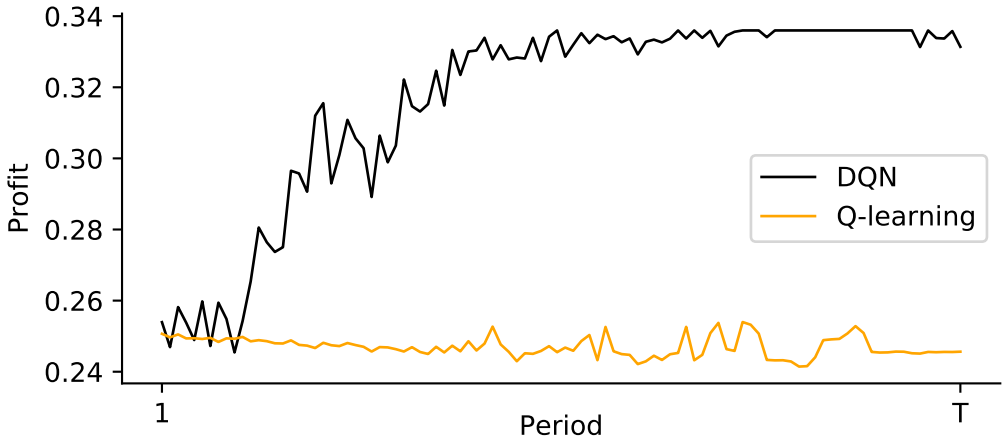
To see the superior learning capabilities of DQN over Q-learning, I let both play against an agent with a rule-based strategy. Each period, DQN trains on a minibatch of $\omega = 32$ experiences, whereas Q-learning trains on a single experience. The simulations with Q-learning last for $T = 32,000$ and with DQN for $T = 1,000$ periods. Thus, they learn on the same number of experiences.

The upper figure 5a compares the profit (reward) of the two algorithms against an Always-Defect agent. This agent follows a rule-based strategy: she always plays the lowest price possible. Q-learning and DQN agents are both able to increase profits when exploration becomes less frequent. DQN quickly gains higher profits than Q-learning. Whereas DQN almost always ends up at the optimal strategy, Q-learning does sometimes learn sub-optimal strategies.

The behavior of the rule-based agent does not change and the state-transition probabilities (9) are time-invariant. The environment fulfills the Markov property,




(a) Competition against Always-Defect agent




(b) Competition against Tit-for-Tat agent

Figure 5: Performance differences between Q-learning and DQN. The plot shows the profit of the Q-learning and DQN agents averaged over 10 simulation runs.

and Q-learning is theoretically guaranteed to find the optimal strategy. Therefore, the agent fail to learn the optimal strategy because experimentation was insufficient for her slow learning pace.

These differences in performance become more evident in the lower figure 5b. The rule-based opponent now follows a Tit-for-Tat strategy. She plays the lowest price if the other agent undercut her in the last period. Otherwise, she plays the opponent's last period's price. Tit-for-Tat is a cooperative strategy and enables higher profits than Always-Defect. Again, DQN quickly learns the optimal strategy and can increase profits when experimentation becomes less frequent. Contrary, Q-learning fails to learn the optimal strategy and ends up with low profits.

## 3.2. Duopolies with Deep Q-Networks

Let us now focus on the more efficient DQN algorithms to see if they can learn collusion at a reasonable time limit. The superior performance is achieved at the cost of higher complexity and, thus, more parameters. It is computationally not feasible to perform a systematic grid search to select values for all parameters. Besides, we do not want to optimize the algorithm to facilitate collusion. Instead, we would like to observe if such behavior occurs with as little prior knowledge and tuning as possible. Where feasible, I adopted standard values or performed an informal search.

Excluded from this are the learning rate $\alpha$ and the size of the replay buffer $\beta$. The learning rate is the most important parameter for training a DNN. Together with the size of the replay buffer, it determines the pace of learning. If the learning rate is too large, the SGD will overshoot and oscillate without finding local or global minima. If the learning rate is too low, the weights could be shifted too little, and it is more likely to end up in a bad local minimum (Goodfellow, Bengio, and Courville 2016).

The replay buffer size $\beta$ determines how quickly changes in the competitor policy are represented in the training data. A small replay buffer could lead to unstable learning behavior because the learning target will change fast. If the replay buffer is too large, the learning target is stable, but the algorithm will take a long time to adapt to changes in the competitor's strategy.

The significant influence of the two parameters on learning makes a systematic grid search for optimal values necessary. I considered replay buffer sizes $\beta$ from 500 to 10,000 samples and learning rates $\alpha$ from 0.00025 to 0.005. One simulation run lasts $T = 100,000$ periods. Figure 6 shows the average profit gain $\Delta$ of a duopoly with DQN agents.[17] The heat map shows the results for all combinations of the parameter.

For small replay buffer sizes, the algorithms are not able to learn collusive strategies. The algorithms have not enough time to learn the opponent's strategy and find a collusive response because the learning targets change too fast. Thus, they play prices near the one-shot game's optimal price, which results in average profit gains near the static Nash equilibrium. For larger replay buffer sizes, the algorithms frequently learn to play collusive strategies and end up with a high average profit gain. The learning rate and replay buffer size are not independent of one another. The results suggest that algorithms with smaller replay buffer must learn faster. For replay buffer size 5,000 or 10,000 and learning rates around 0.001, the average profit gains of DQN are comparable to the ones of Q-learning, see figure 4.

[17] In the following, average profit gain $\Delta$ is always calculated with the profits from the last 5,000 periods.

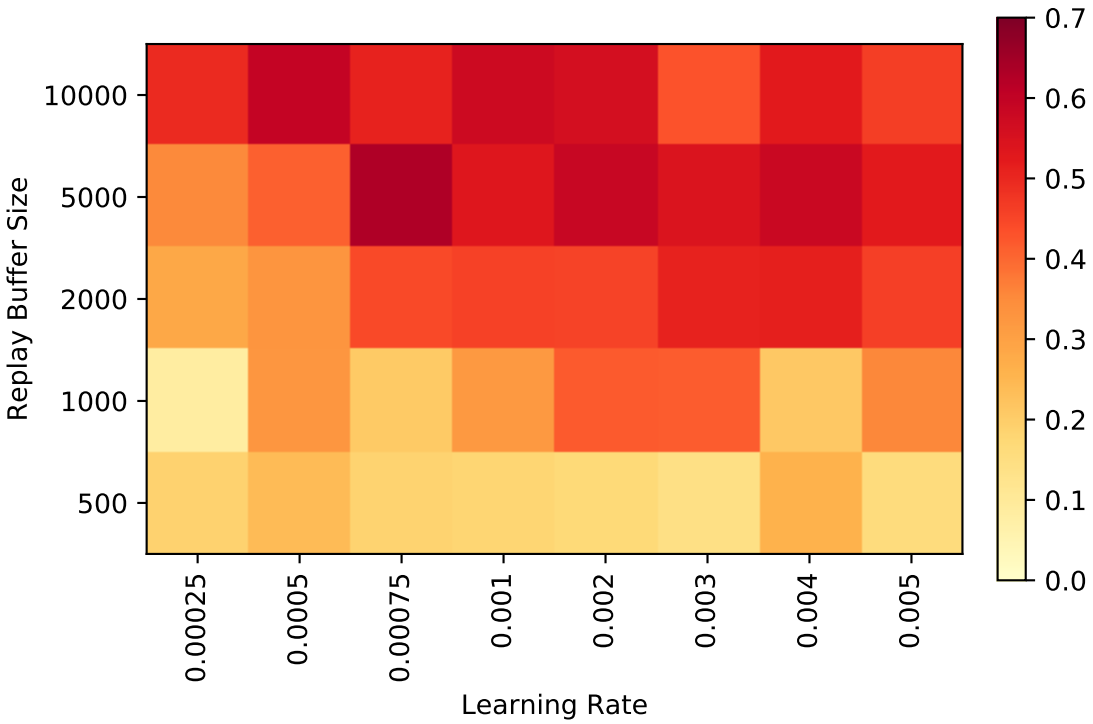



Figure 6: Average profit gains by DQN agents for different learning rates and replay buffer sizes. Each parameter combination is repeated 50 times.

To further investigate the emergence of collusion, I have to limit the analysis to a specific parameter setting. In the following, I will keep the replay buffer size fixed at $\beta = 5,000$ and the learning rate at $\alpha = 0.001$. However, the results are robust to reasonable changes in these parameters.

Figure 7 shows the development of the average profit gain over time.[18] At the beginning of the simulation, both agents experiment frequently. It is not possible to answer to a stable strategy of the opponent. Thus, playing a low price to undercut the opponent is the best an algorithm can do. The prices decrease, and the average profit gain comes closer to the static Nash level.

As the experimentation probability $\varepsilon$ decreases, the agent's strategies become more stable. Already after 20,000 periods, the agents start to learn strategies other than undercutting the opponent. They raise profits by increasing prices. If we again assume that pricing algorithms can update their prices 30 times per hour, this translates to approximately one month. After this turning point in the learning process, the agents steadily improve their strategies. In the last 5,000 periods of the simulation, they obtain average profit gains of $\Delta = 63\%$.

[18] For price movements and the evolution of profits and quantities see figure 21 in appendix B.

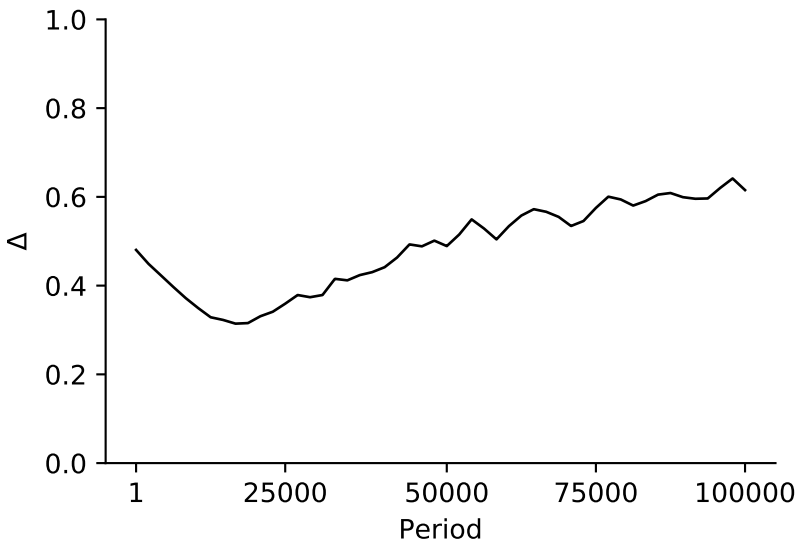



Figure 7: Average profit gain of the DQN agents in a Duopoly. The line represents average values over 50 simulation runs.

## 3.3. Learned Strategies

The previous section focused on market outcomes: the average profit gain compared to the static Nash equilibrium. Besides, it is crucial to examine how collusion occurs. Do the algorithms learn a strategy based on a reward-punishment scheme, or do they just fail to learn a best-response?

Strategies are challenging to describe analytically because they are very complex. In a duopoly with 15 prices, a strategy is a mapping from 225 possible states to 15 actions. Also, the learned strategies differ from simulation run to simulation run. One way to analyze such complex strategies is by looking at averages (Calvano et al. 2020). First, I will examine the response to price cuts to verify if the algorithms learned a reward-punishment scheme. Then I show the average pricing behavior for each possible state, which can be displayed for duopolies.

To see the reaction to a price cut of the opponent, I used the parameter setting from before ($\beta = 5,000$ and $\alpha = 0.001$). After the agents played for $T = 100,000$ periods against each other, learning is disabled. First, the agents play for 50 periods with fixed strategies and then I artificially force one agent to play the static Nash price. This causes the price cut of the defecting agent in period $t = 0$ in figure 8.

How does the non-defecting agent react to defection? She punishes the other agent by lowering her price in $t = 1$, the period after defection. The defecting agent expects this punishment and sets her price near the punishment price as well. Then both agents simultaneously start to slowly increase their prices. Already after five periods, the prices are near the pre-defection level. On average, the prices stay slightly below the long-run price after the punishment phase.

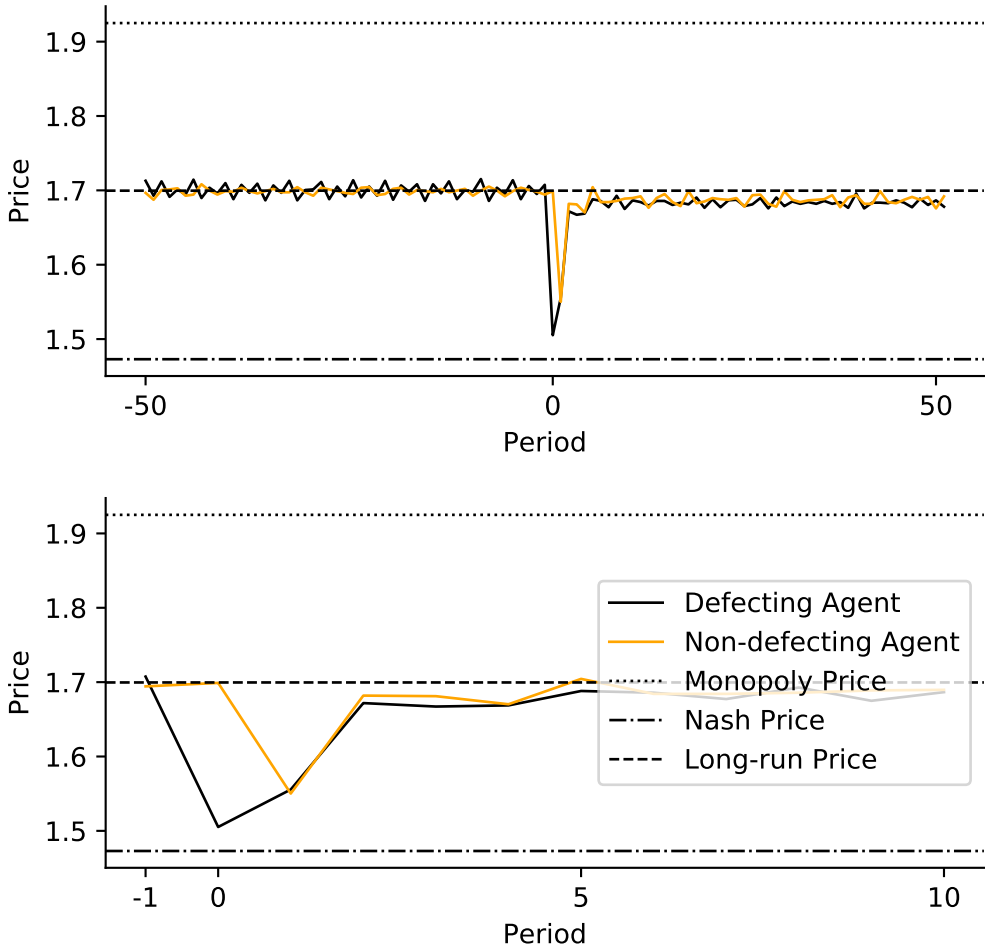



Figure 8: Price reaction after a defection of one agent to the static Nash equilibrium price. The lower figure zooms in to the post defection phase. The lines represent average values over 50 simulation runs.

The price reaction to defection seems to exhibit a reward-punishment scheme. However, a meaningful punishment must make defection unprofitable. Figure 9 shows the change in profit after a unilateral defection, both for the defecting and the non-defecting agent. This change is the difference between the average profit in the periods before defection, $t < 0$, and after defection, $t >= 0$. The change is simulated not just for defection to the static Nash equilibrium price, but for each possible price from 1.43 to 1.97. The long-run price was on average 1.7.

The figure reveals that defection to prices below the long-run price is seldom profitable for the defecting agent. The harsher the defection is, the greater is the loss in profit. Except for a defection to 1.62, the average profit change is negative for all price cuts. Price increases above the long-term price seem to have a positive effect on profits. However, large price increases do not significantly increase profits. In conclusion, the agents have no incentive to lower prices in most of the learned strategies. If at all, there is an incentive to raise prices slightly.

For the non-defecting agent, things are different. Overall, the profit losses are

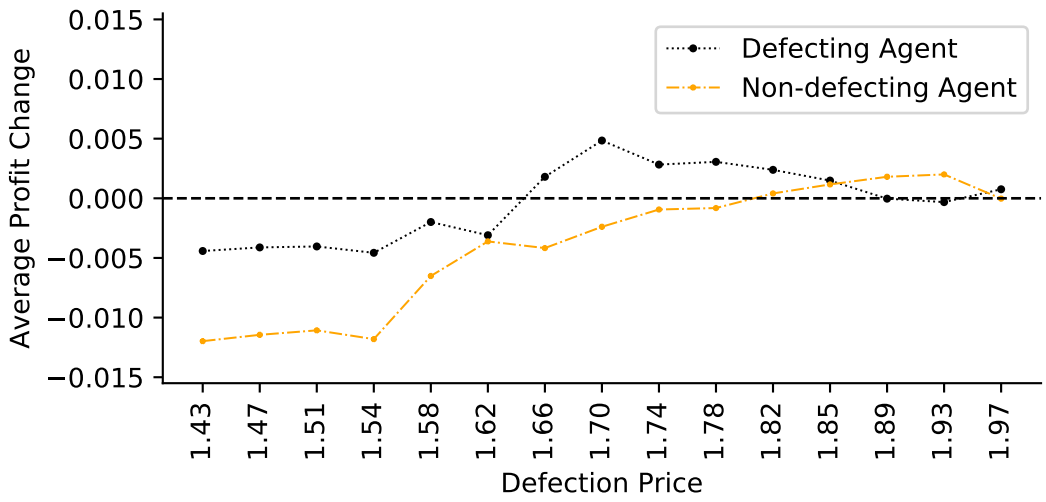



Figure 9: Change in profit before and after defection. Compared are the 50 periods before and the 51 periods after defection for all possible defection prices. The lines represent average values over 50 simulation runs.

higher than for the defecting agent. That is possibly due to the defection period, where the defecting agent can gain high profits. In addition, profits also decrease for price changes above the long-term price up to 1.78. Similar to the defecting agent, losses are greater, the harsher the defection is.

The analysis shows that defection to lower prices is mostly unprofitable due to an effective punishment-scheme. So far the focus was on deviations from the long-run price. Luckily, the learned strategies can be displayed for the whole state-space in markets with two agents. Figure 10 shows the optimal action of agent 1 in each of the $15 \times 15 = 225$ states.[19]

The first thing that catches the eye is the symmetry of the surface. The agent reacts similarly to previous prices no matter which agent played which price. The agent will not just punish the defection of the opponent but also anticipates the punishment if she defects herself. For prices in the medium range, the agent holds the price approximately constant. The concave shape of the surface at the top shows that the agent starts to undercut as prices approach the monopoly price.

Another remarkable behavior is the price increase when both agents played prices near the Nash price. This mutual understanding enables the agents to return to higher prices after a punishment. The learned strategy is reminiscent of the Pavlov (or win-stay-loose-shift) strategy described by D. Kraines and V. Kraines (1989). As long as both cooperate, prices stay at a collusive outcome. If one of the agents defects and unilaterally decreases her price, both agents stop cooperating and lower their prices. If both agents defect and play prices near the Nash equilibrium, both mutually return to cooperation and again play prices at a collusive level.

[19] The strategy for agent 2 has the same shape.

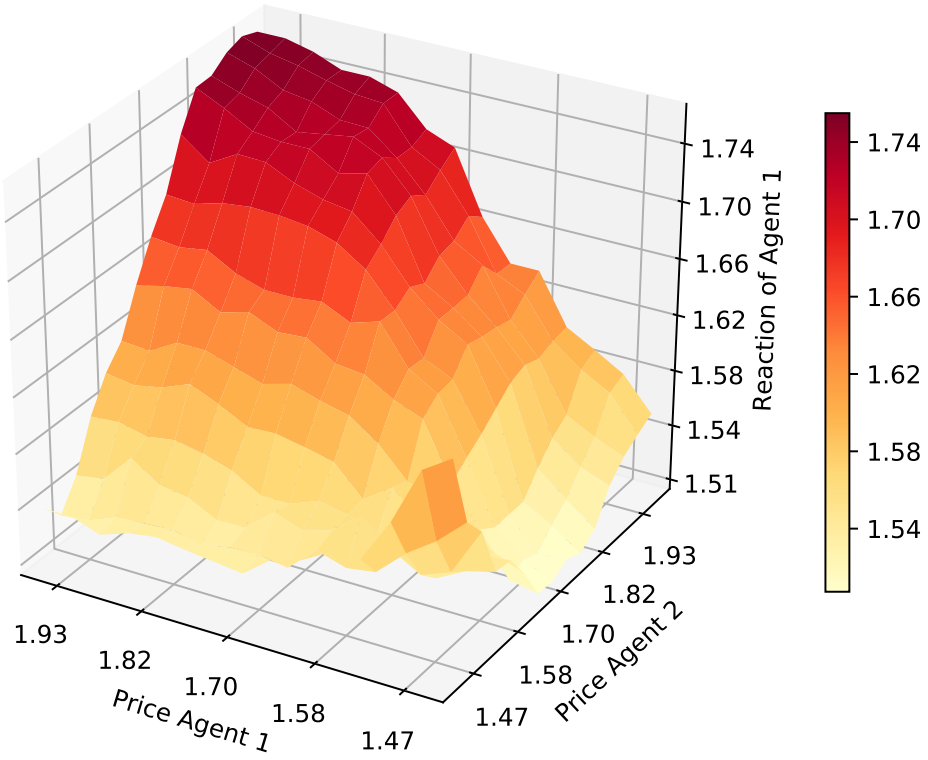


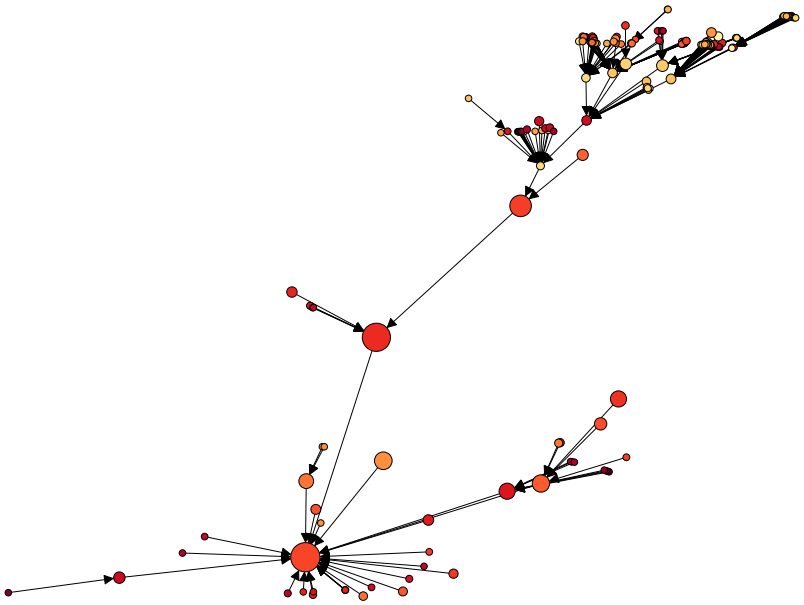

Figure 10: Learned strategy of the DQN agents in a duopoly. The figure displays the optimal action of agent 1 in each possible state according to $a = \arg\max_a Q(s, a, \theta)$. The state $s$ are the previously played prices of the two agents, the values on the x-axis and y-axis. The 3-d surface shows the average optimal action over 50 simulation runs.

The previous results show that some properties prevail in all of the learned strategies. Learning heavily depends on the experience that the agents encounter. This experience is unique for each simulation run and is based on the joint behavior of both agents. Evidently, there is much variation in the learned strategies that vanishes through the aggregation of results.

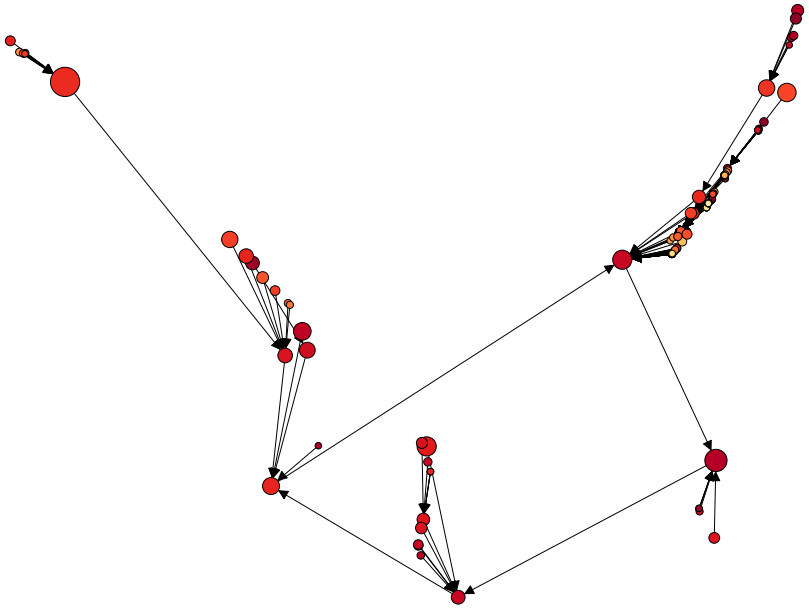
Looking at some examples will further deepen our understanding of the learned strategies. Figure 11 visualizes the joint strategy for two simulation runs. The nodes represent all possible states. The edges show the transition from one state to the next state according to the joint strategies of the agents.[20] The node color indicates the average profit gain in this state. The node size shows how often this state was visited in the last 5,000 periods of the simulation run.

The upper figure 11a reveals a few remarkable properties of the learned strategies. First, no matter from which state the agents start, they eventually end up in the terminal state. This is the large node at the bottom. Thus, the agents will re-start

[20] For each state we can calculate the optimal action that both agents will choose according to $a = \arg\max_a Q(s, a, \theta)$. The two optimal actions then constitute the next state.



(a) Strategy always ends in one terminal state



(b) Strategy always ends in a four period cycle

Figure 11: Exemplary directed graphs of the joint strategies.

cooperation after both unilateral and bilateral defection, which holds for all possible defection prices. Second, just a few central nodes serve as a gateway to the terminal state. Third, the paths toward the terminal state are relatively short. For most states, the terminal state will be reached after five periods. From the 50 simulation runs, 34% of the joint strategies end up in such a collusive terminal state.

The lower figure 11b has similar properties. Instead of one terminal state, the agents will always end up in a cyclic price movement between 4 states. All of these states are collusive, as indicated by the dark red color. 32% of the simulations contain such a cyclic movement. The cycles differ in length from two to five periods.

Another characteristic are separated graphs that do not have a connection to each other. Dependent on the start state, the agents will end up in different terminal states or cyclic movements. Separated graphs are found in 52% of the simulation runs, but often one graph contains just few states. For more details see table 2 in appendix B.

This section showed that self-learning pricing algorithms frequently set supra-competitive prices in a duopoly. This finding holds for simple Q-learning algorithms and more sophisticated algorithms based on function approximation. The analysis of these DQN algorithms reveal that they start to collude after a few thousand periods. That corresponds to roughly one month on common e-commerce platforms. The algorithms do not arrive at the anti-competitive outcome by failing to optimize. Instead, the learned strategies are based on an effective reward-punishment scheme that makes defection non-profitable. The strategies heavily depend on a mutual understanding to return to cooperation after punishment.

# 4. Collusion in Wide Oligopolies

Economic experiments showed that tacit collusion by human price-setters is strongly affected by the number of competitors. Implicit coordination of prices above the competitive level is frequently observed in duopolies, rarely in markets with three firms, and almost never in markets with four firms. In markets with more than four firms, tacit collusion seems unlikely unless the human subjects are allowed to communicate with each other (Potters and Suetens 2013).

Do the same effects apply to pricing algorithms, or can they collude even in wide oligopolies? To answer this question, I repeat the simulation of the previous section for markets with up to 10 firms. The parameters and the DQN algorithms remain unaltered in the baseline setting. Figure 12 shows the average profit gain for different market sizes. Similar to markets with Q-learning algorithms (see figure 4), collusion decreases in the number of market participants. In markets with three firms, the

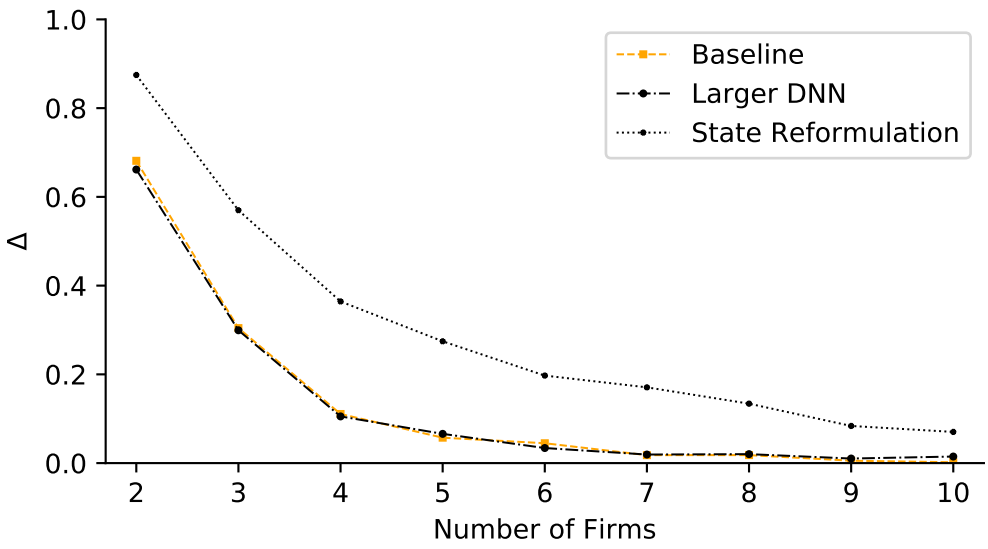



Figure 12: Average profit gains by DQN agents in wide oligopolies. The lines show average values for 10 simulation runs per setting.

average profit gain is $\Delta = 30\%$, and thus significantly higher than the static Nash equilibrium level. With more market participants, the average profit gains fall below $\Delta = 20\%$ and quickly approach zero.

Various factors could have hindered the algorithms from coordinating their prices above competitive levels in larger markets. Either i) the algorithms do not have the capacity to learn complex strategies in the given time or ii) the representation of the state is not suited to learn the competitors' strategy.

To see whether the former hinders the algorithms to learn collusion, I tested both a larger DNN and more time for learning. The larger DNNs have three hidden layers with 128 nodes each. The network size does not seem to facilitate collusion, see figure 12, and the results are almost identical compared to the baseline setting. Too low approximation capabilities do not qualify as a reason for failed collusion.

Collusion could also fail because the algorithms do not have enough time to learn each other's strategy plus the market demand. Longer time for learning could be especially important with larger networks. However, even in simulation runs with up to $T = 500,000$ periods, the collusion level does not increase. That holds for agents with both small and large DNNs.

In addition, I tested whether advancements in the algorithms, as such, could increase collusion in wide oligopolies. Double DQN (DDQN) developed by Van Hasselt, Guez, and Silver (2016) is the successor of DQN. The latter suffers from overestimation of the action-values under certain conditions. This overestimation is reduced by decomposing action selection and action evaluation in the target value

$Y_t = R_{t+1} - \bar{R} + \hat{Q}(S_{t+1}, \max_a Q(S_{t+1}, a, \theta), \hat{\theta})$.[21] Although DDQN proved to have a better performance in several application domains, agents based on this advanced algorithm do not show more collusive behavior than DQN.

The agents' learning capacity does not seem to be the barrier to collusion in wide oligopolies. Alternatively, I test the influence of the state representation on the learning of strategies. The state determines how an agent perceives the environment and, therefore, the other agents' behavior.

In the baseline setting of the simulation, the previous period's prices constitute the state. The examination of strategies in section 3.3 shows that it does not matter who played which price in the previous round. More important are the overall price level and whether an agent defected by either playing a higher or lower price. Equipped with this knowledge, a reasonable state representation should contain the average of the last period's prices. Defection can be measured by including the minimum and maximum value of last period's prices.

Figure 12 shows the results for simulations where each agent perceives the state of the environment according to this formulation. The average profit gains are significantly higher for all simulated market sizes. For three firms the profit gain is $\Delta = 57\%$, for four $\Delta = 36\%$ and for markets with 5 firms still $\Delta = 27\%$. With more firms, it falls below 20% and approaches zero, similar to the baseline setting. Somewhat surprising is the fact that the reshaped state representation also helped collusion in duopolies.

The preprocessing is based on the knowledge which information algorithms need in order to learn collusive strategies. Although the agent's DNNs receive in principle the same information, aggregation of the previous period's prices seems to reduce detrimental variation. That frees the DNNs from distinguishing between important price defection and small price fluctuations.

I use a straightforward preprocessing of the previous period's prices. In real-world applications, more advanced methods could be required. Possible candidates are a direct measure of the opponent's willingness to cooperate, some economic structure, or a strategy model. The simulation results indicate that such expert knowledge could enable collusion even in wide oligopolies.

# 5. Heterogeneous Pricing Algorithms

In a real market place, firms develop their pricing algorithms independently. They will end up with algorithms that differ in the state representation, learning pace,

[21] For a comparison see equation (23).

or learning method. Tacit collusion by algorithms heavily relies on joint learning. Section 3.3 shows that the mutual understanding to increase prices after defection is a defining characteristic of the strategies. Up until now, the algorithms in each simulation are identical. One might presume that this similarity facilitates collusion. Thus, I test whether heterogeneity among algorithms influences the ability to collude.

## 5.1. On-Policy vs. Off-Policy

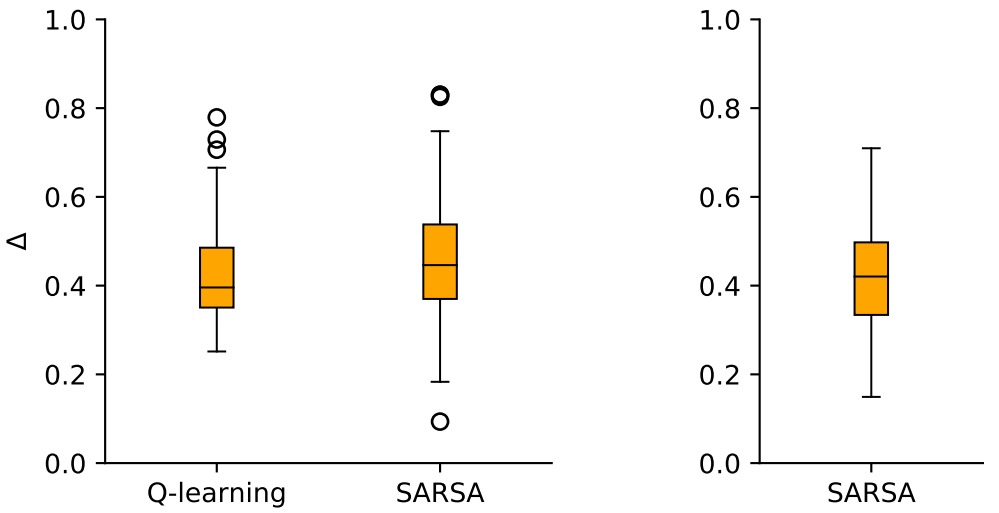
The developer of a RL based pricing algorithm faces various design decisions. Among the most fundamental decisions is the method of learning. Whether algorithms learn on-policy or off-policy is by no means clear. Off-policy algorithms, e.g., Q-learning, learn an optimal policy while following another policy, e.g., an $\varepsilon$-greedy policy. While this avoids the unwanted influence of the agent's behavior on learning the optimal policy, it introduces a danger of instability and divergence. This is especially true for learning with function approximation. Most of the modifications described in section 2.2.2 solely attack this danger. On the other hand, on-policy methods often proved to be more stable but learn a sub-optimal behavior policy.

The on-policy pendant to Q-learning is SARSA. The learning target in Q-learning is the observed reward plus the expected next period's Q-value under an optimal policy, $Y_t = R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$. With SARSA the learning target is the observed reward plus the observed next period's Q-value, $Y_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$. The next period's Q-value apparently depends on the action $A_{t+1}$ the agent played and, therefore, on the behavior policy. As long as the agent experiments, she will not learn the optimal policy. See appendix C for the full pseudo-code of SARSA.

To test how a different learning method influences the ability to collude, I simulated duopolies with both SARSA and Q-learning agents. The simulation runs last $T = 1,000,000$ periods. The parameters for SARSA and Q-learning remain unchanged compared to section 3 ($\alpha = 0.125$ and $\gamma = 0.95$). The average profit gain in a market with two SARSA agents is $\Delta = 42\%$, see the right figure 13b. SARSA seems to reach lower average profit gains in comparison to Q-learning.[22] The disadvantage of on-policy learning seem to outweigh the benefits of higher stability. Sometimes SARSA agents even end up at profit levels near the static Nash equilibrium.

The left figure 13a shows the results for a market with one Q-learning and one SARSA agent. It reveals that the Q-learning agent cannot reach a similar collusion level with SARSA instead of Q-learning as her collaborator. Even though the two different algorithms sometimes reach a high level of collusion up to 80%, this happens rarely and the average profit gain is $\Delta = 45\%$.

[22] In a duopoly with two Q-learning agents, the average profit gain is $\Delta = 67\%$, see figure 3.1.




(a) Q-learning against SARSA (b) SARSA against SARSA

Figure 13: Average profit gains in a duopoly with agents learning on-policy (SARSA) and off-policy (Q-learning). The box-plot shows the results of 50 simulation runs.

The results underline the importance of joint learning for tacit collusion. The Q-learning agents are not able to signal the less-performing SARSA agent to raise prices. Instead, the collusion level seems to be oriented towards the outcome of SARSA.

## 5.2. Fast vs. Slow Learner

The previous section shows that algorithms with less ability to learn collusive strategies will thwart superior algorithms. However, what happens if two algorithms play against each other that, in principle, can learn a similar level of collusion but learn at a different pace? To answer this question, I let a DQN agent play against a Q-learning agent. Each simulation run lasts $T = 100,000$ periods. Figure 14 shows the average profit gains.

The faster DQN agent can exploit the slower Q-learning agent. DQN is able to reach profits above the monopoly level, whereas Q-learning earns profits near the static Nash equilibrium. Figure 22 in appendix B reveals that DQN plays a highly competitive strategy. The Q-learning agent is not able to optimize her behavior fast enough. The DQN agent takes advantage of it and lowers her price. In this way, she drives the Q-learning agent out of the market and gains high rewards.[23]

The high profits of the DQN agent are not a sign of collusion. She fully exploits

[23] I repeated the simulation with a longer time horizon of $T = 1,000,000$. With more time, Q-learning can learn a reward-punishment scheme. Both agents start to cooperate and end up at similar collusive outcomes compared to duopolies with two Q-learning or two DQN agents.

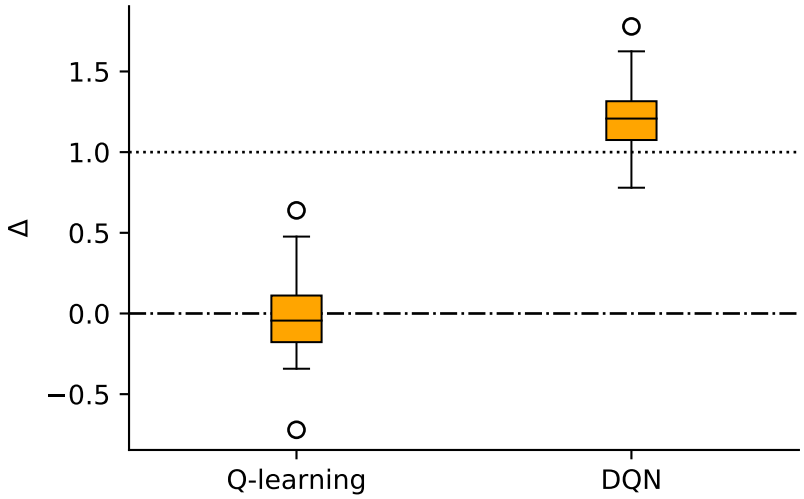



Figure 14: Average profit gains in a duopoly with a fast (DQN) and a slow (Q-learning) agent. The box-plot shows the results of 50 simulation runs.

the Q-learning agent. Again, this shows the importance of joint learning for tacit collusion. If both agents learn effective reward-punishment schemes, collusion emerges. However, cooperation is not an integral part of the self-learning algorithms. Instead, the algorithms fully take advantage of the chance to exploit the opponent and selfishly increase their profit.

## 5.3. Self-Learning vs. Rule-Based Strategy

Currently, pricing algorithms with a rule-based strategy are still in use. The managers or programmers define more or less exactly how the algorithm should set the price for their product. Self-learning algorithms are on the rise, but there will be a transition phase where both types of algorithms compete with one another. The question arises how rule-based algorithms influence collusion among algorithms.

Figure 15 shows the average profit gain in a market with two DQN agents and one rule-based agent. This agent plays one of the most prominent pricing strategies in the e-commerce sector: penetration pricing. She tries to undercut the competitors by setting the price one margin below their last period's prices.

Although this strategy is non-cooperative, the self-learning algorithms do not start a race to the bottom. The penetration agent cannot be punished and, thus, the self-learning agents let her receive profits near the monopoly level. In this way, they are able to reach average profit gains of $\Delta = 61\%$ themselves.[24] This is only slightly less compared to the duopoly case with two DQN agents.

[24] I repeated the simulation for another popular pricing strategy: premium pricing. The premium pricing agent performs worse than the penetration pricing agent. She will end up with profits near the Nash equilibrium level. The self-learning agents seem to be relatively unaffected and

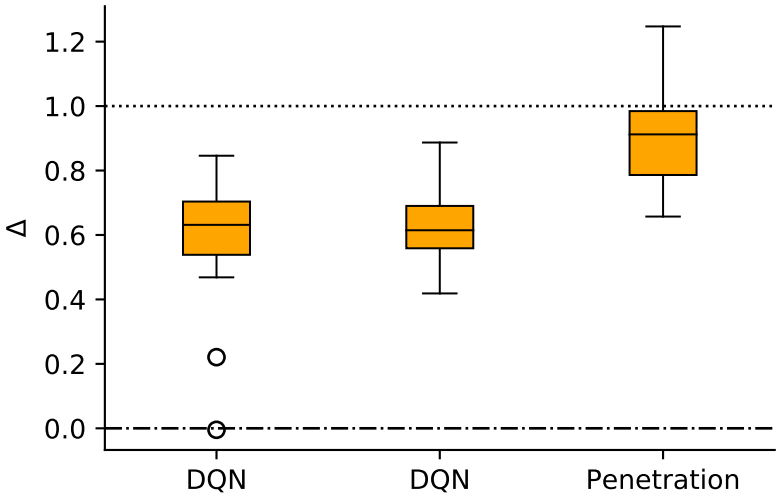



Figure 15: Effect of a rule-based agent on the average profit gain. The box-plot shows the results for 30 simulation runs.

The results illustrate an important property of a self-learning algorithm: she perceives the other firms as part of the environment. By interacting with the environment, the algorithm must learn the demand model plus the opponent's strategy, because both influence the state-transition probabilities (9). This task is daunting with self-learning opponents, because their strategy changes during the learning process and make the state-transition probabilities non-stationary. Contrary, a rule-based opponent behaves according to a time-invariant strategy. Even though this increases the complexity of the state-transition probabilities, it does not decrease their stationarity.[25]

With this insights, it becomes apparent that the coordination problem's complexity between two self-learning algorithms will not significantly increase with an additional rule-based agent. The previous section 4 shows that larger markets will decrease the likelihood of tacit collusion. However, this seems only valid if the additional agents are self-learning. Simple rule-based pricing algorithms (or human-price setters that do not frequently change prices) do not necessarily hinder tacit collusion.

## 5.4. Asymmetric Costs

While the previous sections focus on heterogeneity among algorithms, this section addresses heterogeneity on the firm's level. Tacit collusion under asymmetries is regarded as more complicated because the firms have to coordinate on two factors.

again can establish a highly collusive market outcome.

[25] Section 3.1 gives an impression how one self-learning performs against one rule-based algorithm. Since the opponent plays a time-invariant strategy, the environment is stationary from the self-learning agent's point-of-view. RL algorithms are designed to tackle stationary environments, and with enough exploration they will eventually always learn the optimal strategy.

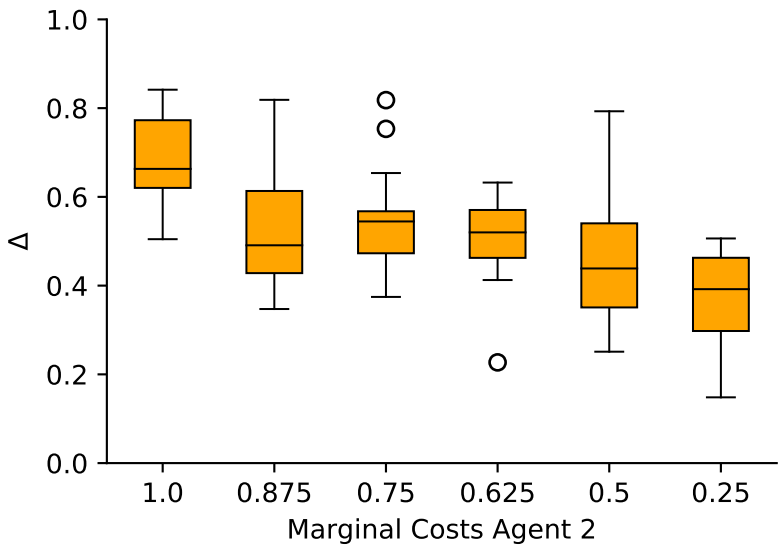



Figure 16: Average profit gains with asymmetric costs. The box-plot shows the results for 20 simulation runs per value of $c_2$.

Besides the overall price level, they must agree on the relative prices that determine how the profit is split between firms (Calvano et al. 2020). Does this two-sided optimization problem make collusion harder for the self-learning algorithms?

Figure 16 shows the average profit gains under cost asymmetries. The marginal costs of the second agent $c_2$ vary from 1.0 (symmetric cost structure) to 0.25. With increasing asymmetry the average profit gain decreases but remains at a high level of not less than $\Delta = 37\%$.

At least to some extent, the decrease in profit gain is due to the missing opportunity for side payments. The profit from collusion is disproportionally divided in favor of the less efficient agent 1. This becomes apparent from figure 17. For symmetric costs, the profit of agent 1 is near her monopoly profit. With increasing asymmetry, the profit of agent 1 decreases. However, the slope of her monopoly profit is steeper than the slope of her average profits. Joint profit maximization would require agent 1 to further lower her output and gain profits near the monopoly profits. Around marginal costs $c_2 = 0.5$, the monopoly profit drops below the static Nash profit. Here at the latest, it becomes clear that agent 1 has no incentive to do so without side payments.

The coordination problem of the algorithms does not necessarily get more challenging with asymmetric costs. Instead, they coordinate on a solution that does not maximize the joint profit. Heterogeneity among algorithms seems to actually hinder collusion, whereas asymmetries among firms instead shift the coordination problem's solution.

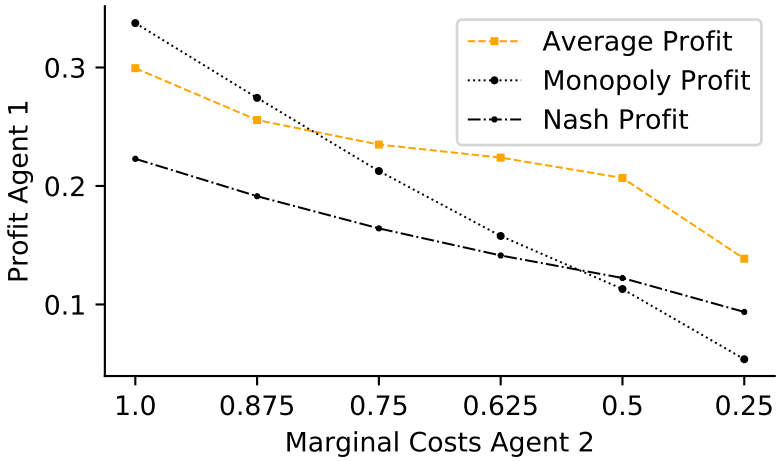



Figure 17: Shift of the maximization target under asymmetric costs.

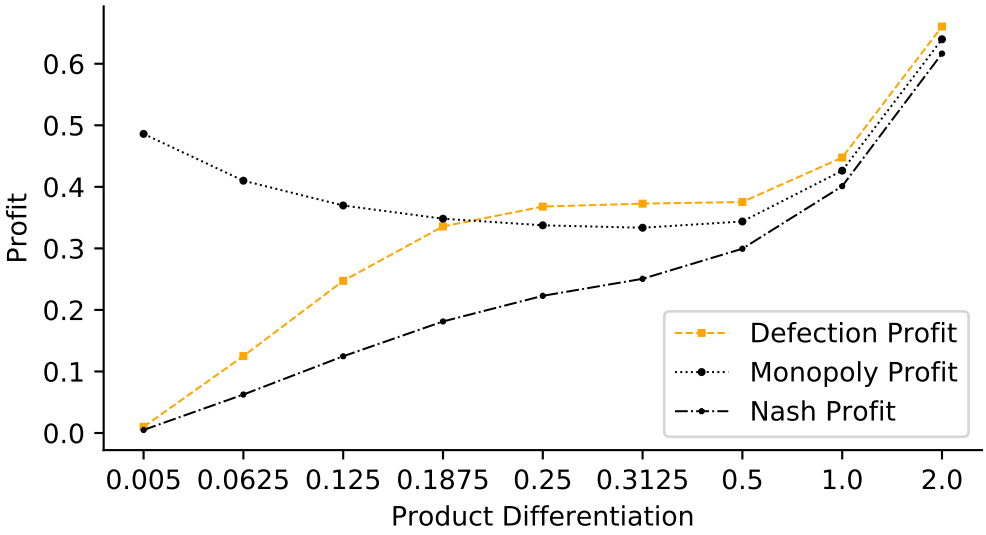



Figure 18: Profit from defection and possibility to punish with product differentiation.

## 5.5. Product Differentiation

Another interesting extension is the effect of horizontal product differentiation. With increasing $\mu$, the consumer tastes become more heterogeneous. The products are less substitutable and demand less responsive to changes in the price. How does product differentiation influence tacit collusion?.

Figure 18 shows the profit from cooperation (both agents play the monopoly price $p^M$) in comparison to the profit from competitive prices (both agents play the Nash price $p^N$). Besides, the orange line denotes the profit from defection. That is the profit an agent receives if she plays the Nash price $p^N$ and the other agent plays the monopoly price $p^M$. With decreasing $\mu$, the difference between monopoly and Nash profit becomes larger. Thus, deviation could be punished harsher with more homogeneous products. Additionally, higher product substitutability decreases the

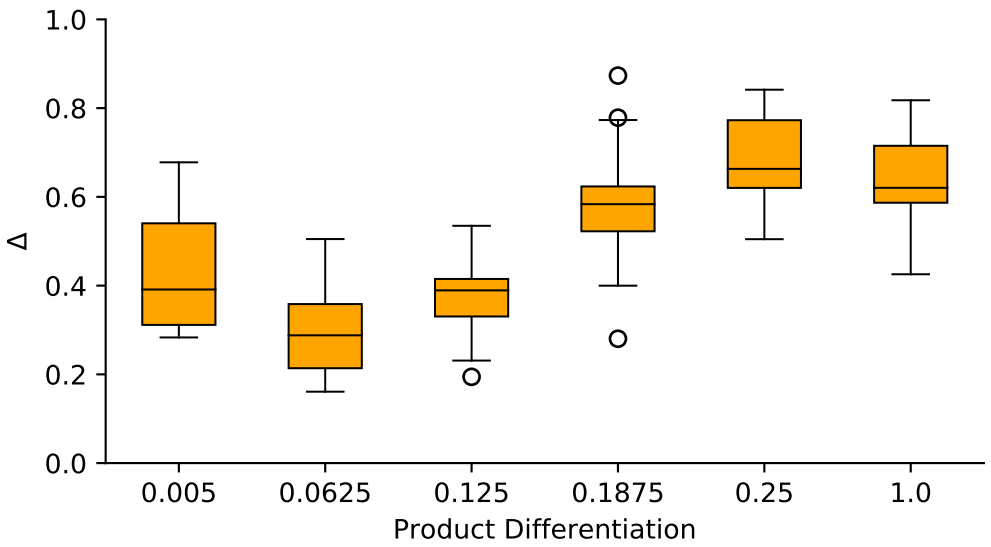



Figure 19: Average profit gains for different degrees of product differentiation. The box-plot shows the results for 20 simulation runs per value of $\mu$.

individual firm's profit from defection. It seems that more homogeneous products make tacit collusion easier to sustain.

Figure 19 shows the effect of setting $\mu$ from 0.005 (products are almost perfect substitutes) to 0.25 (the baseline) and up to 1.0. When the products are almost perfect substitutes, the algorithms gain supra-competitive profits of $\Delta = 43\%$. With higher values of $\mu$, the average profit gain increases. For $\mu = 0.25$ and higher, the average profit gains stay constant around $\Delta = 60\%$.

This somewhat surprising result could be explained by the incentive to return to cooperation. Collusive strategies rely on a reward-punishment scheme. Thus, the agents must learn to punish, but at the same time, they must learn to return to cooperation afterward. We have previously seen that defection can be punished less harsh with increasing $\mu$. Figure 20 reveals that, at the same time, profits of an agent that unilaterally raises her price increase in $\mu$. Thus, an agent has a higher incentive to increase her prices after a punishment to the Nash equilibrium.

Section 3.3 shows that a collusive strategy is based on a mutual return to cooperation. It can be suspected that the incentive to raise prices is essential for learning such strategies and possibly more important than a harsh punishment. In any case, the results show that the simulated market with higher product substitutability bear a lower risk of tacit collusion by pricing algorithms.

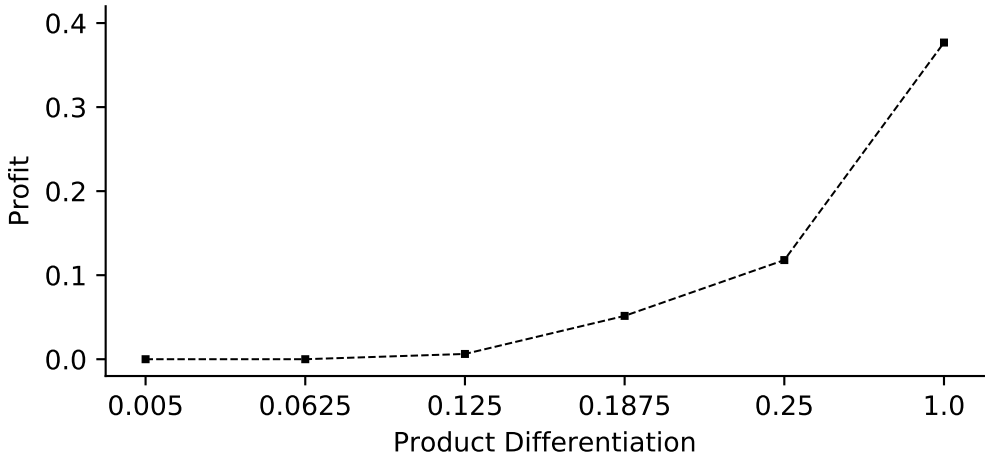



Figure 20: Incentive to raise prices after punishment. The line shows the profit of an agent that plays $p^M$ and her opponent $p^N$.

# 6. Conclusion

The simulations have shown that state-of-the-art RL algorithms systematically set supra-competitive prices in a duopoly. The learned strategies contain a reward-punishment scheme that enforces collusion. Defection gets punished by the opponent with a subsequent mutual return to cooperation. The algorithms started without prior knowledge and learned the strategies by experimentation. There is no instruction to collude in the software code, nor did the algorithms communicate with each other.

The learned strategies coincide with those of Calvano et al. (2020), but the algorithms powered by DNNs learn to collude significantly faster. They start to increase prices after approximately one month if the experimental setting is transferred to real-world e-commerce platforms.

The more efficient pricing algorithms made experiments in wide oligopolies with up to ten algorithms possible. With an increasing number of market participants, the level of collusion decreases. Whereas increasing the capacity of the DNNs cannot prevent this tendency, already a slight reformulation of the state representation increases the ability to collude across all market sizes.

Collusion heavily depends on mutual understanding. The strategies result from the algorithms' history with each other and often work solely in combination with the opponent's strategy. Consequently, heterogeneity among pricing algorithms hinders collusion. This is true for differences in the pace of learning and the learning method as such. In contrast, asymmetries among firms seem to change the algorithms' optimization target but do not necessarily hinder collusion. The same applies to additional firms that use algorithms with a rule-based strategy.

From a regulatory point of view, the results seem to be alarming. The current

regulatory approach relies on the assumption that collusion without communication is unlikely with human price-setters. We do not have a direct comparison of the ability to tacitly collude between humans and algorithms. However, the results suggest that pricing algorithms quickly learn to collude. Methods that learn more efficiently than DQN are already available today, and the advent of artificial intelligence will most probably further increase the risk of algorithmic collusion. Even wide oligopolies do not seem to prevent collusion unconditionally. Collusion becomes more difficult with more self-learning algorithms in the market, but this does not hold for more rule-based algorithms or human price-setters. In the light of these findings the false negatives rate could likely increase with such a lenient antitrust policy.

Heterogeneity among algorithms seems to hinder collusion effectively. Therefore, competition policy should prevent firms from using the same or similar algorithms. Especially pricing algorithms from third party providers may pose a high risk if they are widely used in one market. However, similarities could also occur between independently developed algorithms because the research field and the number of available algorithms are still relatively small.

So far, it was not possible to directly examine pricing strategies. By moving the price decision from the manager's mind to an algorithm, a direct examination of the strategy becomes feasible. When regulatory authorities detect potentially collusive behavior, the used algorithms could be tested in a simulated environment that replicates the monitored market. This approach would require tacit collusion to be unlawful. So far, tacit collusion was not considered unlawful because it was prone to a lot of false positives. The spread of algorithmic pricing could reduce the error rate and make this policy approach desirable (Harrington 2019; Calvano et al. 2020).

Future research should increase robustness and external validity of the findings. Several improvements bring both pricing algorithms and simulated market environment closer to real-world applications. First, prices should not be restricted to a limited set of values. Instead, the algorithms should choose from a continuous price spectrum. Since DQN cannot tackle a continuous action space, further experiments should focus on other algorithms, e.g., actor-critic methods.

Second, the model-free algorithms reach their limits in wide oligopolies. Reshaping the state representation seems to improve the algorithm's learning and further facilitate collusion. It would be interesting to incorporate additional prior knowledge into the algorithms, e.g., the market's economic structure. Generally speaking, it would be beneficial to find out which algorithms are deployed in practice in order to improve their external validity.

Third, the economic environment's complexity should increase. Additional product features beyond quality make the demand model less stylized. For e-commerce

platforms, this could include shipping costs, reviews, or position in the list of products. On the firm's side, additional variables like the stock of products could be incorporated into the pricing decision. It is particularly interesting how such external factors hinder the interpretation of price movements by competing firms. They must learn to differentiate between defection and a price change that is motivated by firm-specific factors which might pose a challenge to collusion (Calvano et al. 2020).

Lastly, additional factors that make collusion more difficult should be identified. Equipped with this knowledge, regulatory authorities will be able to implement effective measures against tacit collusion by pricing algorithms.

# A. Simulation Parameter

| Economic Environment | | |
|---|---|---|
| Parameter | Symbol | Default Value |
| Marginal costs | $c$ | 1.0 |
| Quality | $g$ | 2.0 |
| Price sensitivity | $\mu$ | 0.25 |
| Number of prices | $m$ | 15 |
| Q-learning | | |
| Parameter | Symbol | Default Value |
| Number of periods | $T$ | 1,000,000 |
| Learning rate | $\alpha$ | 0.125 |
| Discount factor | $\gamma$ | 0.95 |
| Deep Q-Network | | |
| Parameter | Symbol | Default Value |
| Number of periods | $T$ | 100,000 |
| Learning rate | $\alpha$ | 0.001 |
| Reward step size | $\lambda$ | 0.01 |
| Periods between target network updates | C | 100 |
| Size of the replay buffer | $\beta$ | 5,000 |
| Number of hidden layers | | 2 |
| Number of nodes per hidden layer | | 32 |
| Minibatch size | $\omega$ | 32 |
| The exponential decay rate for the 1st moment estimates of Adam | $\beta_1$ | 0.9 |
| The exponential decay rate for the 2nd moment estimates of Adam | $\beta_2$ | 0.999 |
| A small constant for numerical stability of Adam | $\epsilon$ | 1e-7 |

Table 1: Default parameter of the economic environment and the algorithms.

# B. Additional Simulation Material

## B.1. Limited Time: Duopoly with Deep Q-Networks

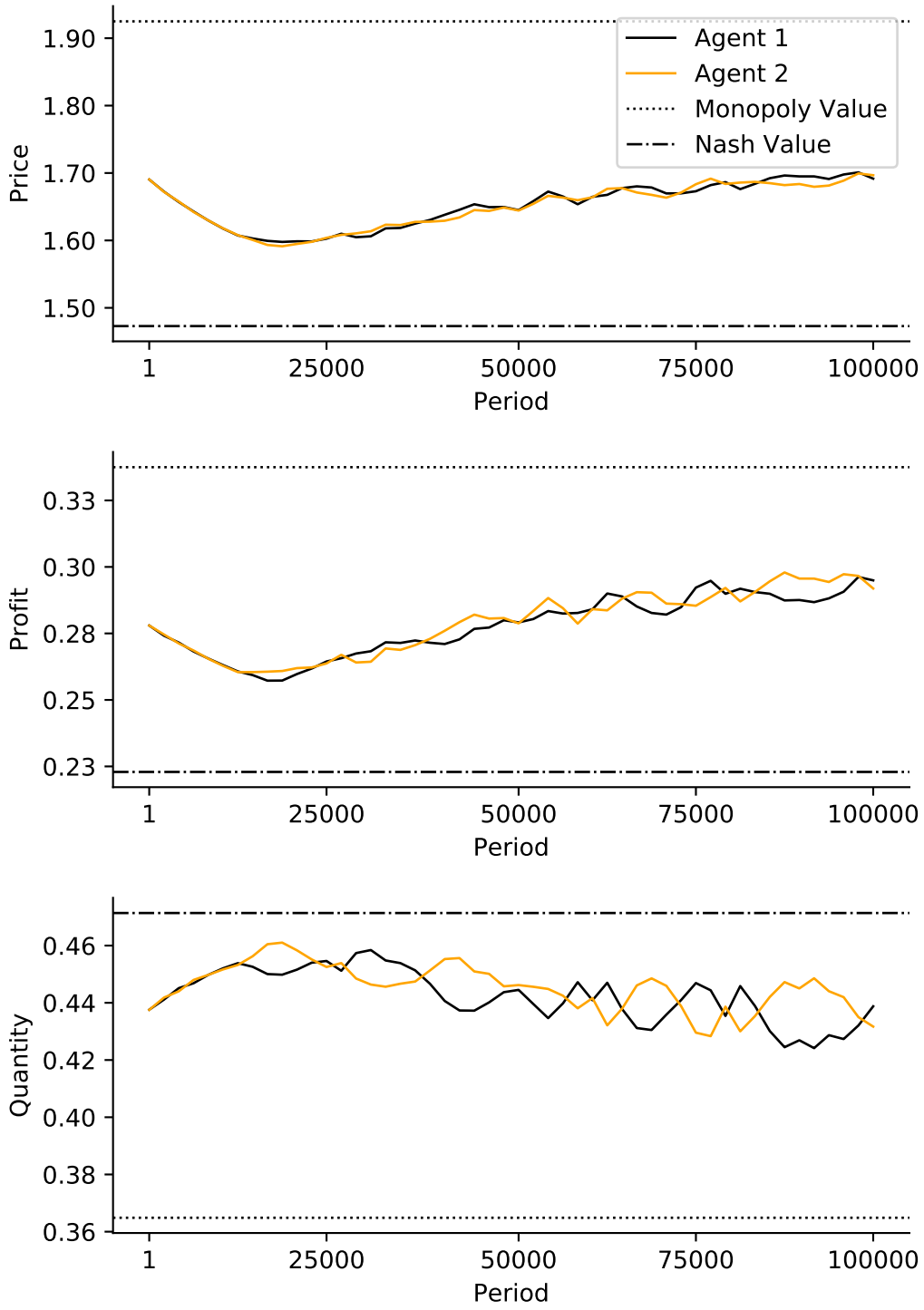



Figure 21: Prices, profits and quantities of the DQN agents in a duopoly. The lines represent average values over 50 simulation runs.

## B.2. Heterogeneity: Fast vs. Slow Learner

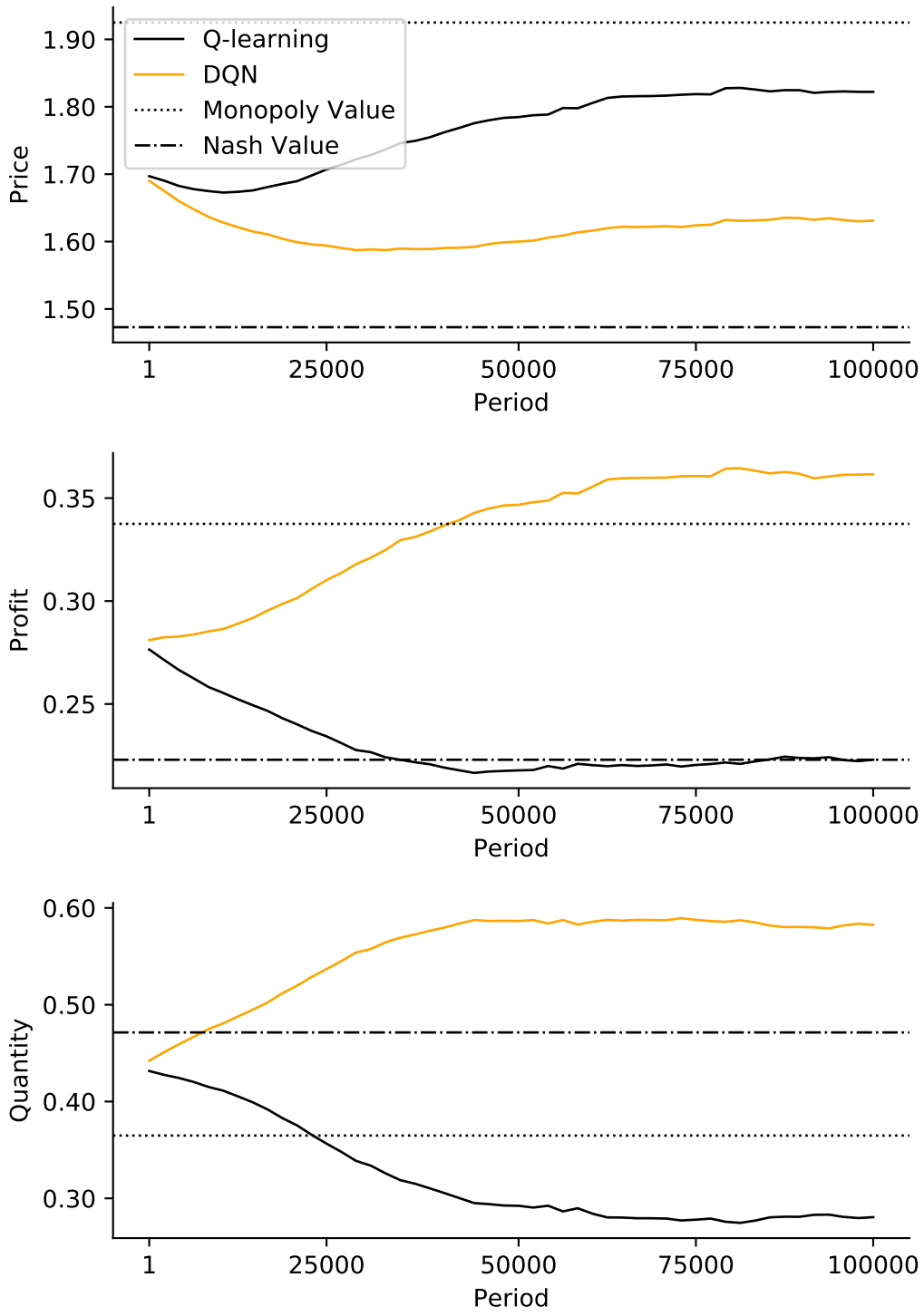



Figure 22: Prices, profits and quantities in a duopoly with a fast (DQN) and a slow (Q-learning) agent. The lines represent average values over 50 simulation runs.

## B.3. Strategy Characteristics

| Characteristic | Counts | Frequency |
|---|---|---|
| Strategy with a single terminal state | 17 | 34% |
| Strategy with a cycle | 16 | 32% |
| 2 periods cycle | 5 | 10% |
| 3 periods cycle | 4 | 8% |
| 4 periods cycle | 5 | 10% |
| 5 periods cycle | 2 | 4% |
| Strategy with separated parts | 26 | 52% |
| Multiple terminal states | 17 | 34% |
| Terminal states and cycles | 9 | 18% |

Table 2: Characteristics of the strategies learned by DQN agents. The table shows how often a characteristic is found in the joint strategies of 50 simulation runs.

# C. SARSA Algorithm

**Algorithm 3** SARSA

Initialize $Q(s,a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}$, arbitrarily
Initialize $S_1$ arbitrarily
With probability $\varepsilon_t$ play a random action $A_1$
Otherwise play $A_1 = \arg\max_a Q(S_1, a)$
**for** $t = 1, T$ **do**
    Observe $R_{t+1}$, $S_{t+1}$
    With probability $\varepsilon_t$ play a random action $A_{t+1}$
    Otherwise play $A_{t+1} = \arg\max_a Q(S_{t+1}, a)$
    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big]$

# References


Abadi, Martín et al. (2016). "TensorFlow: A System for Large-Scale Machine Learning". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*.

Amazon (2020). *Selling Partner API Documentation*. URL: `https://github.com/amzn/selling-partner-api-docs` (visited on 09/29/2020).

Anderson, Simon P. and André De Palma (1992). "The Logit as a Model of Product Differentiation". In: *Oxford Economic Papers* 44.1, pp. 51–67.

Bloembergen, Daan et al. (2015). "Evolutionary Dynamics of Multi-Agent Learning: A Survey". In: *Journal of Artificial Intelligence Research* 53, pp. 659–697.

Buşoniu, Lucian, Robert Babuška, and Bart De Schutter (2008). "A Comprehensive Survey of Multiagent Reinforcement Learning". In: *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews* 38.2, pp. 156–172.

Byrne, David P. and Nicolas De Roos (2019). "Learning to Coordinate: A Study in Retail Gasoline". In: *American Economic Review* 109.2, pp. 591–619.

Calvano, Emilio et al. (2020). "Artificial Intelligence, Algorithmic Pricing, and Collusion". In: *American Economic Review* 110.10, pp. 3267–3297.

Chen, Le, Alan Mislove, and Christo Wilson (2016). "An Empirical Analysis of Algorithmic Pricing on Amazon Marketplace". In: *25th International World Wide Web Conference*.

Ezrachi, Ariel and Maurice E. Stucke (2017). "Artificial Intelligence & Collusion: When Computers Inhibit Competition". In: *University of Illinois Law Review*, pp. 1775–1810.

– (2018). "Sustainable and Unchallenged Algorithmic Tacit Collusion". In: *University of Tennessee Legal Studies Research Paper* 366, pp. 1–39.

Federal Trade Commission (2018). *FTC Hearing 7: Competition and Consumer Protection*. URL: `https://www.ftc.gov/news-events/audio-video/video/ftc-hearing-7-nov-14-welcome-remarks-session-1-algorithmic-collusion` (visited on 09/30/2020).

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. Cambridge: MIT Press.

Harrington, Joseph E. (2019). "Developing Competition Law for Collusion by Autonomous Artificial Agents". In: *Journal of Competition Law & Economics* 14.3, pp. 331–363.

Ivaldi, Marc et al. (2003). *The Economics of Tacit Collusion*. Tech. rep. March.

Klein, Timo (2019). "Autonomous Algorithmic Collusion: Q-Learning Under Sequential Pricing". In: *Amsterdam Law School Research Paper* 2018-15, pp. 1–40.

Kraines, David and Vivian Kraines (1989). "Pavlov and the Prisoner's Dilemma". In: *Theory and Decision* 26, pp. 47–79.

Kühn, Kai-Uwe and Steve Tadelis (2017). *Algorithmic Collusion*. Tech. rep.

Laurent, Guillaume J., Laëtitia Matignon, and N. Le Fort-Piat (2011). "The World of Independent Learners is not Markovian". In: *International Journal of Knowledge-Based and Intelligent Engineering Systems* 15.1, pp. 55–64.

Leibo, Joel Z et al. (2017). "Multi-Agent Reinforcement Learning in Sequential Social Dilemmas". In: *Proceedings of the 16th International Joint Conference on Autonomous Agents and Multiagent Systems*.

Leufkens, Kasper and Ronald Peeters (2011). "Price Dynamics and Collusion under Short-Run Price Commitments". In: *International Journal of Industrial Organization* 29, pp. 134–153.

Lynch, David J (2017). *Policing the Digital Cartels - Financial Times*. URL: `https://www.ft.com/content/9de9fb80-cd23-11e6-864f-20dcb35cede2`.

Maskin, Eric and Jean Tirole (1988). "A Theory of Dynamic Oligopoly, II: Price Competition, Kinked Demand Curves, and Edgeworth Cycles". In: *Econometrica* 56.3, pp. 571–599.

Mehra, Salil K. (2016). "Antitrust and the Robo-Seller: Competition in the Time of Algorithms". In: *Minnesota Law Review* 100.4, pp. 1323–1375.

Mnih, Volodymyr et al. (2015). "Human-Level Control through Deep Reinforcement Learning". In: *Nature* 518.7540, pp. 529–533.

Monopolkommission (2018). *Wettbewerb 2018*. Tech. rep.

Naik, Abhishek et al. (2019). "Discounted Reinforcement Learning Is Not an Optimization Problem". In: *Optimization Foundations for Reinforcement Learning Workshop at NeurIPS 2019*.

OECD (2017). *Algorithms and Collusion: Competition Policy in the Digital Age*. Tech. rep.

Potters, Jan and Sigrid Suetens (2013). "Oligopoly Experiments in the Current Millennium". In: *Journal of Economic Surveys* 27.3, pp. 439–460.

Priluck, Jill (2015). *Can Algorithms Form Price-Fixing Cartels? - The New Yorker*. URL: `https://www.newyorker.com/business/currency/when-bots-collude`.

Salcedo, Bruno (2015). "Pricing Algorithms and Tacit Collusion".

Schwalbe, Ulrich (2019). "Algorithms, Machine Learning, and Collusion". In: *Journal of Competition Law & Economics* 14.4, pp. 568–607.

Silver, David et al. (2016). "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529.7587, pp. 484–489.

Singh, Satinder P., Tommi Jaakkola, and Michael I. Jordan (1994). "Learning Without State-Estimation in Partially Observable Markovian Decision Processes". In: *Machine Learning Proceedings 1994*.

Sutton, Richard S. and Andrew G. Barto (2018). *Reinforcement Learning: An Introduction*. 2. Edition. Cambridge and London: MIT Press, pp. 1–526.

UK Competition and Markets Authority (2018). "Pricing Algorithms: Economic Working Paper on the Use of Algorithms to Facilitate Collusion and Personalised Pricing".

Van Hasselt, Hado, Arthur Guez, and David Silver (2016). "Deep Reinforcement Learning with Double Q-Learning". In: *30th AAAI Conference on Artificial Intelligence*.

Virtanen, Pauli et al. (2020). "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python". In: *Nature Methods* 17.3, pp. 261–272.

Waltman, Ludo and Uzay Kaymak (2008). "Q-learning Agents in a Cournot Oligopoly Model". In: *Journal of Economic Dynamics and Control* 32, pp. 3275–3293.

Watkins, Christopher J. C. H. (1989). "Learning from Delayed Rewards". PhD thesis. Cambridge: King's College, pp. 1–234.

## Eidesstattliche Erklärung

….